**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ**

**Факультет комп'ютерних наук, фізики та математики**

**Кафедра комп'ютерних наук та програмної інженерії**

# Методи аналізу вразливостей апаратного забезпечення

**Кваліфікаційна робота (проєкт)**

на здобуття ступеня вищої освіти «магістр»

Виконав: студент 2 магістерського курсу 241м групи
Спеціальності: 121 Інженерія програмного забезпечення
Освітньо-професійної програми:
Інженерія програмного забезпечення
Гаврилів В. В.
Керівник: доктор фізико-математичних наук, професор
Песчаненко В.С
Рецензент: кандидатка технічних наук, доцентка кафедри програмних засобів і технологій, Херсонський національно-технічний університет
Захарченко Р. М.

Івано-Франківськ – 2024

# ЗМІСТ

**ВСТУП……3**

**РОЗДІЛ 1. Рівні аналізу вразливостей апаратного забезпечення……5**

1.1. Рівень проектування програмного забезпечення……5
1.2. Рівень проектування апаратного забезпечення……...7
1.3. Фізичні вразливостей програмно-апаратних архітектур……11
1.4. Захист інтегрованих систем від фізичних вразливостей……12
1.5. Сучасні технологічні засоби захисту програмного забезпечення від апаратних вразливостей……13
1.6. Проблеми захисту програмно-апаратних архітектур……..15
1.7. Сучасні технології та програмні засоби для вирішення проблем захисту програмно-апаратних архітектур……...19

**РОЗДІЛ 2. Інсерційне моделювання……23**
2.1. Системи переписування термів……23
2.2. Система алгебраїчного програмування……29
2.3. Транзиційні системи та алгебри поведінок……..31
2.4. Система інсерційного моделювання……32

**РОЗДІЛ 3. Модель для аналізу атак з впровадженням помилок……...44**
3.1. Загальна постановка задачі……44
3.2. Переписування двійкового коду……44
3.3. Інсерційна модель для задачі “винуватець та латник”……47

**ВИСНОВКИ……..48**

**СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ……49**

# ВСТУП

Програмні атаки на апаратне забезпечення, такі як атаки на бічні канали живлення, шляхом заміни вбудованого в мікроконтролери програмного забезпечення на подібне зловмисне, ставлять під загрозу безпеку вбудованих систем за низьку вартість. Захист вбудованих систем від таких атак передбачає захист апаратних блоків, програмного забезпечення та інтеграцію апаратного та програмного забезпечення.

Розробник апаратного забезпечення або розробник програмного забезпечення реалізує апаратне чи програмне забезпечення на мові високого рівня, яка пізніше оптимізується до кінцевого продукту за допомогою автоматизовані засоби. Складність у розробці програмно - апаратних систем полягає насамперед у великій вартості процесів їх тестування, оскільки вони вимагають фізичного виготовлення чіпу відповідної архітектури. Системи формальної верифікації програм дозволяють в свою чергу створення повної моделі усіх складових частин програмно апаратної системи.

**Актуальність** роботи полягає в необхідності розробки алгоритмів побудови моделей програмно-апаратних систем заснованих на мікроконтролерах. Ці моделі дозволять виявлення апаратних та програмних вразливостей цієї системи на рівні проектування, до передачі мікроконтролерів безпосередньо до виробництва. Програмне і апаратне забезпечення системи реалізують відповідні частини, використовуючі спеціалізовані мови високого рівня, тому застосування технологій формальної верифікації для контролю якості кінцевих продуктів є актуальною. А задача оцінки можливих вразливостей апаратної частини до безпосередньо її виготовлення може вважатись критичною.

**Метою** даної роботи є створення моделі для виявлення вразливостей для можливих атак на апаратне забезпечення, через програмну складову

системи у системі інсерційного моделювання, на етапі проектування кінцевої програмно - апаратної системи, до безпосередньої передачі її у виробництво.

Виходячи з мети дослідження було виділено наступні **задачі дослідження**:

1. Спроектувати найпростішу модель програмно апаратної системи, для демонстрації атаки на програмно-апаратну систему типу “винуватець-латник”.
2. Розробити інсерційну модель, з описом станів успішного завершення та тупикових станів для представлення частини можливої атаки на програмно-апаратну систему “винуватець”.
3. Розробити агента для створеної інсерційної моделі, який відображатиме поведінку частини програмно-апаратної системи, яка відповідає за протидію можливим атакам - “латник”.

**Об’єкт дослідження** — програмні атаки на програмно - апаратні системи.

**Предмет дослідження** — програмні атаки, що використовують вразливості програмно-апаратних систем, для завдання шкоди безпосередньо апаратному забезпеченню системи.

**Наукова новизна результатів дослідження** полягає в тому що в теперішній час не розроблені повноцінні методології побудови моделей для оцінки витоків на етапі проектування для автоматичного пошуку причини спостережуваного витоку в апаратному та програмному забезпеченні.

**Практичне значення результатів дослідження** полягає в тому, що розроблена інсерційна модель для оцінки та виявлення програмно-апаратної атаки типу “винуватець-латник”, може бути використана в подальших дослідженнях з даної теми, як базис для побудови моделей, що дозволять виявлення інших видів програмно-апаратних вразливостей.

# РОЗДІЛ 1. РІВНІ АНАЛІЗУ ВРАЗЛИВОСТЕЙ АПАРАТНОГО ЗАБЕЗПЕЧЕННЯ

## 1.1. Рівень проектування програмного забезпечення

Розробники програмного забезпечення зазвичай реалізують технічне завдання, розробляючи висхідний код користуючись об’єктно-орієнтованими мовами програмування високого рівня (C, Java, Python тощо) [27]. Висхідний код вважається найвищим рівнем абстракції конкретної задачі, за який відповідають безпосередньо програмісти. При створенні програмного коду можуть використовуватись готові бібліотеки, доступ до яких організовується через “API” (англ. “application programming interface - прикладний програмний інтерфейс”), що дозволяє “приховувати” їх реалізацію. Для запуску відповідного програмного забезпечення на мікроконтролері, необхідне перетворення висхідного коду в машинний код для формування бінарного файлу, що завантажується в пам'ять застосунка.

Для генерації бінарного файлу, висхідний код запускається засобами кросс-компілятору що відповідає конкретній цільовій архітектурі мікроконтролера. Робота кросс-компілятора організована в кілька циклів для оптимізації та генерації машинного коду. Для оптимізації процесу трансляції, висхідний код спочатку транслюється в проміжне представлення компілятора (IR). Після чого це нове представлення також здійснює відповідні заходи оптимізації, які застосовуються до різних частин висхідного коду [39].

Проміжне представлення компілятора — це спеціальна структура даних, яка представляє висхідний код і призначена для його оптимізації та перекладу у машинні команди процессора.

Машинні команди процессора - останній рівень абстракції в процесі розробки програмного забезпечення.

Команди процессора зберігаються в програмній пам'яті мікроконтролера [38]. Безпосередньо бінарний код буде отримано та виконано мікроконтролером. Це найменш придатний для читання людьми шар програмних абстракцій, оскільки він складається лише з бінарних значень. Для покращення читабельності машинних команд процесора спеціальні застосунки, які мають назву дизассемблери генерують висхідний код у цільовій мові програмування на основі двійкового файлу.

Проходи компілятора спрямовані переважно на оптимізацію бінарного коду для оптимізації швидкості роботи або розміру у пам'яті на різних рівнях (рисунок 1.1), тому у кінцевому програмному модулі на вищих рівнях проектування будуть відсутні будь-які заходи проти атак фізичного рівня.

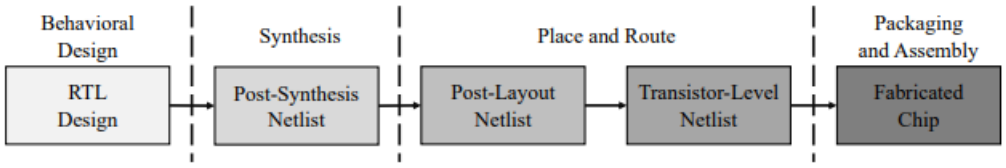


Рисунок 1.1. Рівні абстракції в цифровому апаратному забезпеченні

На (рисунок 1.2) представлено фрагмент коду, що ілюструє контрзахід на рівні вихідного коду.

Мета гілки “else” - створення постійного часу виконання незалежно від значення “try pwd”.

Однак ця гілка викликає мертвий код, який буде усунено на рівні проміжного представлення за допомогою оптимізаційних проходів компілятора. Так само контрзахід, реалізований на рівні проміжного представлення компілятора за допомогою оптимізаційних проходів, може бути видалений.

Задача переконання того, що висхідний код містить необхідні контрзаходи для протидії апаратним та програмним атакам є алгоритмічно складною.

## 1.2. Рівень проектування апаратного забезпечення

У цифровому циклі проектування апаратного забезпечення, після встановлення системних специфікацій, підготовлюється модель поведінки у вигляді схеми на спеціалізованих мовах опису апаратного забезпечення (англ. Hardware Description Language) [56]. Найбільш часто використовуваними мовами опису апаратного забезпечення є *Verilog*, *SystemVerilog* і *VHDL*. Таке моделювання поведінки системи виконується на рівні передачі реєстру (англ. Register Transfer Layer).

Програмне та апаратне забезпечення що працює на рівні передачі реєстру запускається інструментом автоматизації синтезу електронного проектування (англ. Electronic Design Automation), який генерує список підключень на шлюзовому рівні [58]. Інструмент синтезу відповідає за оптимізацію дизайну команд рівня передачі реєстру відповідно до області, обмеженням потужності та часу при визначенні конструкції в термінах заданої стандартної бібліотеки. Таким чином, в процесі синтезу на шлюзовому рівні можуть бути введені нові буфери для поєднання частини існуючої логіки.

Далі список мережевих підключень на шлюзовому рівні буде проаналізовано інструментом автоматизації синтезу електронного проектування для створення макета, після чого починається процес виготовлення який, складається з кількох кроків:

- підготовка плану мікросхеми;
- розміщення клітинок і макросів;

- синтез дерева годинника;
- маршрутизація сигналів.

Після процесу створення макету, запускаються процеси перевірки правил проектування для певної технології (англ. Design Rule Checking) і перевірки розміщення деталей самої схеми на макеті (англ. Layout Versus Schematic), для того щоб переконатися, що реалізована схема готова до виготовлення [44].

Усі вищезазначені процеси можуть бути розглянуті або у вигляді списку підключень безпосередньо на рівні самої схеми, або на нижчому рівні абстракції, що відображає транзистори та інформацію про з'єднання на рівні передачі реєстру, тобто на рівні транзисторів. Ці шари абстракції (тобто такі що передують безпосередньо запису логіки схеми фізично у кремнієвий кристал) призначені для підготовки макету для остаточного тестування перед виготовленням. Під час самого процесу виготовлення зміна схеми призведе до небажаних різних змін у характеристиках транзисторів. Отже, до кремнієві атрибути транзисторного рівня не зовсім збігаються з пост кремнієвими атрибутами. Виготовлена мікросхема буде потребувати детальної інструкції з експлуатації для подальшої роботи. Інтеграція мікросхеми в плату може сама по собі створити складніші розбіжності між моделлю до виробництва, виробничим ланцюгом і кінцевою робочою системою [46].

Будь-які зміни, що вносяться в логіку роботи чіпу, в тому числі і контрзаходи запобігання шкоди від встановлених вразливостей, які впроваджуються в дизайн апаратного забезпечення, неминуче повинні пройти усі вищезгадані кроки, що відповідають рівням абстракції. На основі результатів цього процесу, робиться висновок, щодо працездатності певного реалізованого контрзаходу в остаточному записаному апаратному чіпі. Це підкреслює важливість аналізу вразливостей на етапі розробки. Крім того,

вищі рівні абстракції обмежують складність змодельованих вразливостей [48]. Наприклад, на рівні передачі реєстру затримки сигналу не моделюється. Таким чином, будь-яка вразливість, викликана такими затримками, буде пропущена під час аналізу потоку виконання на рівні передачі реєстру.

Приклад процесу проектування обладнання, з урахуванням вищезгаданих кроків, наочно продемонстровано на процесі проектування архітектури мікрочіпу Picochip. Picochip був розроблений з метою забезпечення повністю робочої локальної платформи. Функціональні можливості цієї платформи дозволяють проведення експериментів з порівняння до-кремнієвих та пост-кремнієвих атрибутів оцінки витоку бічного каналу живлення.

На **першому етапі** загальна структура, показана на (рисунок 1.3), була змодельована у форматі рівнів передачі реєстру.

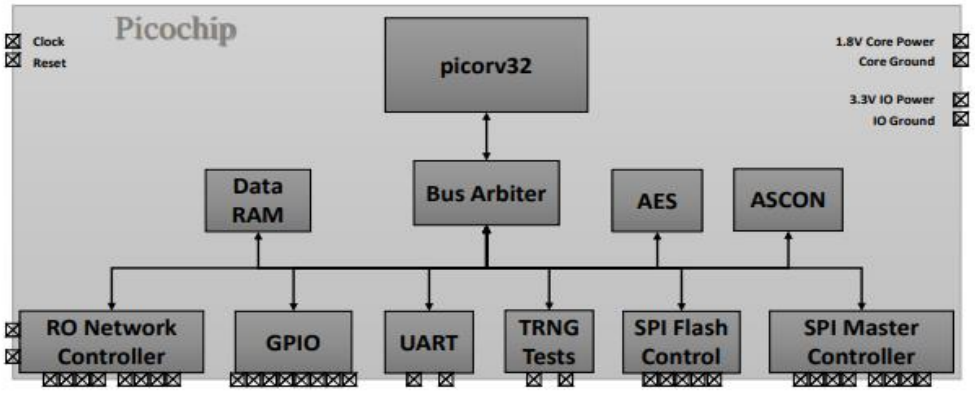


Рисунок 1.3. Блоксхема Picochip

Використання компілятора моделі Synopsys (Synopsys DC) дозволяє проводити синтез файлів, згенерованих на рівні передачі реєстру, на шлюзовому рівні “netlist” для стандартної бібліотеки клітинок “TSMC 180” нм і частотою 80 МГц. Розмір оперативної пам'яті процесора архітектури Picochip дорівнював 64kB.

На **другому етапі** місця та маршрут генеруються за допомогою технології Synopsys IC Compiler II (Synopsys ICC2). На цьому кроці на контактні панелі вводу-виводу, та контактні панелі логіки спочатку, що представляють із себе клітини розміру 3 мм × 5 мм були розміщені макроси пам’яті [47].

На **третьому кроці** створюється “годинникове” дерево для маршрутизації сигналів які надходять від системного годинника. Інші сигнали направляються далі, і додаються клітини-заповнювачі, для задоволення мінімальної відповідності вимогам щодо щільності металу ливарного цеху. Результат цього кроку показано на (рисунок 1.4).

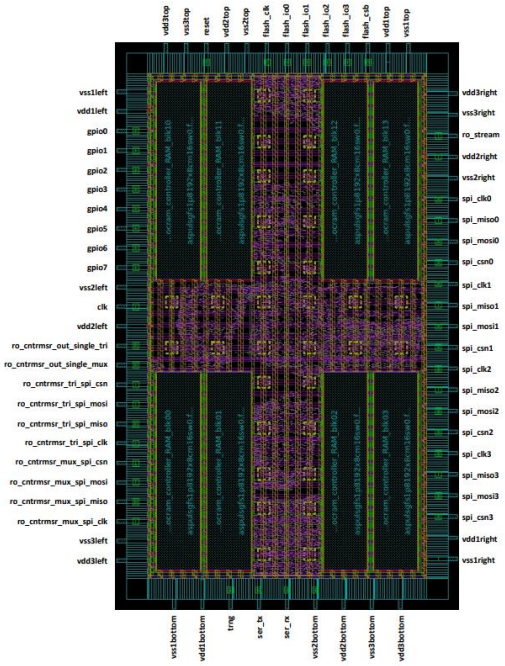


Рисунок 1.4. Розподіл шарів на Picochip

Після завершення останніх етапів перевірки, за допомогою програмного застосунка Mentor Graphics Calibre, файл з макетом мікрочіпа у форматі GDSII відправляється на виготовлення [49]. Після виготовлення чіпу він з’єднується (рисунок. 1.5) із контейнером, що забезпечує можливість його інтеграції до спеціально розробленої вже готової схеми.

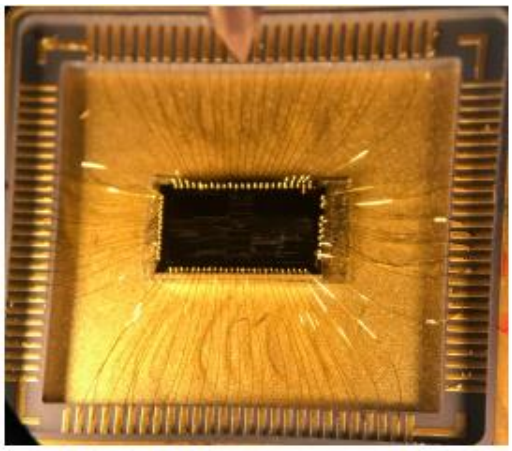

Рисунок 1.5. З’єднання виготовленого чіпу з пакетом

## 1.3. Фізичні атаки на апаратно-програмну архітектуру

Фізичні атаки підрозділяються на дві категорії: активні та пасивні [42]. При активних атаках, також відомих як ін'єкція помилки (англ. Fault Injection), зловмисник вносить зміни в атакований апаратний пристрій, для того щоб отримати будь-яку інформацію про його внутрішній стан або порушити його правильну поведінку. Несправності можуть бути введені в пристрої з використанням різних методів, таких як інжекція оптичної несправності [43], електромагнітна ін'єкція помилок (англ. Electromagnetic - Fault Injection), виклик збоїв напруги та годинника. В методі оптичної ін'єкції, помилка вводиться до пристрою шляхом впливу безпосередньо на кремнієвий кристал джерелом світла високої інтенсивності. Для застосування цього підходу потрібен декапсульований із контейнера чіп. З іншого боку, метод електромагнітної ін'єкція помилок дозволяє введення помилок напряму через контейнер, що усуває необхідність декапсуляції чіпа.

З іншого боку, зловмисник має можливість здійснювати пасивні атаки, також відомі, як аналіз бічних каналів (англ. Side Channel Analysis), без внесення будь-яких змін у пристрій фізично, просто шляхом спостереження та вимірювання поведінки пристрою на основі довільних відомих вхідних

даних. Джерелами інформації для методу аналізу бічних каналів є:

- споживана потужність пристрою [37];
- електромагнітне випромінювання [38];
- інформація про середній час витоку із пристрою.

В процесі здійснення фізичних атак на програмно-апаратні архітектури, навіть якщо саме апаратне забезпечення зазнає найбільшої шкоди від атаки, основною метою атаки все одно може бути програмне забезпечення. Наприклад, зловмисник може тимчасово підробити рівень напруги, яка доставляється до пристрою (введення помилки з перебоєм напруги), з метою спричинення пропуска виконання конкретної інструкції в програмному коді. В іншому прикладі, зловмисник може здійснити вимірювання споживаної потужності пристрою з урахуванням контрольованих входів, з метою отримання значень секретних даних, які використовуються в програмному коді. У програмно-апаратних системах виконання машинного коду програмного забезпечення розподіляється на групу транзисторів.

Отже, фізичні атаки на програмно-апаратні архітектури мають справу з поєднанням рівнів абстракції як у програмному, так і в апаратному компонентах.

## 1.4. Захист вбудованих апаратно-програмних архітектур

Атаки засобом аналізу потужності стороннього каналу мають на меті знайти секретні дані на пристрої шляхом внесення некоректних даних у енергоспоживання пристрою. Контрзаходи проти подібних атак розділяються на дві групи: приховування та маскування.

**Приховування:** мета контрзаходів приховування полягає в збільшенні енергоспоживання пристрою незалежно від внутрішніх даних шляхом рандомізації або вирівнювання. Прихована протидія може бути

реалізована як у часовому, так і в амплітудному вимірах [47]. У часовому вимірі атака реалізується шляхом перемішування процесів виконання операцій або блокуванням вирівнювання слідів потужності, і тому постобробка слідів потужності стане складною. В амплітудному вимірі атака реалізується через зменшення відношення сигналу до шуму (англ. Signal Noise Relation), шляхом внесення секретних даних до системи контролю енергоспоживання. В цьому випадку пошук правильного ключа стає алгоритмічно складним для супротивника. Простий приклад застосування методу приховування контрзаходів шляхом вирівнювання енергоспоживання полягає у паралельному виконанні дій, зворотніх кожній операції.

**Маскування:** мета контрзаходів маскування полягає в порушенні залежності енергоспоживання пристрою від основних секретних даних шляхом застосування генератора випадкових чисел до внутрішніх даних [48]. У логічному маскуванні на *першому етапі* для кожного входного біту даних, береться випадковий біт з рівномірного розподілу та виконується операція *XOR* із кожним бітом даних, для генерації замаскованих спільних ресурсів. На *другому етапі* усі операції налаштовуються для роботи з замаскованими даними і генерації замаскованого виходу.

## 1.5. Сучасні технологічні засоби захисту програмного забезпечення від апаратних вразливостей

Більшість сучасних наукових робіт з теми дослідження стосуються захисту програмного забезпечення від апаратних атак на вбудовані системи, працюючи над усуненням вразливостей у коді складання (тобто вихідному коді) програмного забезпечення або апаратних компонентів. Це збільшує важливість перехресного підходу до безпечних апаратно-програмних архітектур. Програмне забезпечення Rosita [67] знаходить вразливі місця для

аналізу бічних каналів живлення в апаратно-програмних архітектурах через моделювання витоку на збірці на рівні коду з використанням розширеної версії бібліотеки ELMO [58]. На наступному кроці система усуває знайдений витік, шляхом оновлення масок введення. Основне припущення в Rosita полягає в тому, що незважаючи на вже існуюче застосування програмним кодом технології маскування у якості контрзаходу, система всеодно не відповідає необхідним показникам безпеки. Система Rosita++ [66] розширює можливості оригінальної системи Rosita для розробки "маскових" програм вищого порядку.

Системи COCO [72] та PARAM [73] вирішують проблему витоку живлення з бічного каналу, що спостерігається в апаратно-програмній архітектурі на апаратному рівні. PARAM працює над симуляцією поведінки на рівні абстракції, тоді як COCO доводить безпеку замаскованого дизайну на рівні воріт після синтезу.

Оскільки витік із апаратно-програмних архітектур походить від апаратного забезпечення, ці підходи дозволяють з великою вірогідністю знайти причину витоку. Більш просунуті архітектури вимагають нових правил маскування в програмному забезпеченні та змін в апаратному забезпеченні для забезпечення безпечної реалізації. Подібні ситуації були вивчені для суперскалярної реалізації у програмному забезпеченні RISC-V [74] з використанням системи REBECCA [75] - інструменту, спочатку розробленого для формальної перевірки замаскованих обладнання.

Система ELMO [76] — емулятор живлення з точними інструкціями для пристроїв ARM Cortex-M0. Витік потужності оцінюється в ELMO на основі взаємодії між послідовними інструкціями.

Система ABBY [45] представляє собою автоматизовану методологію для створення моделей витоків на основі технологій машинного навчання.

Задача усіх вищезазначених систем (за винятком PARAM) полягає у

пошуку і виправленні помилок у замаскованих реалізаціях програмного забезпечення.

Система Althoff та подібні до неї [46] з іншого боку мають на меті представити підтримку архітектури для контрзаходів із періодичним приховуванням результатів власної роботи. Хоча система Althoff не обмежується маскованим програмним забезпеченням, вона застосовує механізм оцінки витоку на рівні передачі реєстру, що обмежує спостережуваний витік на моделі поведінки.

## 1.6. Проблеми захисту програмно-апаратних архітектур

Наявність вразливостей в апаратно-програмних архітектурах передбачає необхідність впровадження відповідних заходів протидії атакам на всіх можливих напрямках: на апаратному напрямку, і на програмному напрямку. Розробник програмного забезпечення створює висхідний код застосунка на мові програмування високого рівня, використовуючі велику кількість вже готових бібліотек які потенційно включають механізми захисту від програмних та апаратних атак.

Висхідний код програмного забезпечення буде проходити через численні алгоритми оптимізації та генерації коду, перед тим як потрапить до компілятора для створення двійкового файлу. Так само для розробника апаратного забезпечення, найбільш простим етапом для впровадження контрзаходів є етап передачі реєстру, який пройде через багато автоматизованих кроків під час синтезу та підготовки файлу формату GDSII до виготовлення.

Перехід від одного рівня абстракції до іншого здійснюється за допомогою автоматизованих інструментів (компіляторів, ланцюгів інструментів для генерації та верифікації частин програмного забезпечення та інструментів САПР для апаратного забезпечення). Ці інструменти в

переважній кількості розроблені з урахуванням необхідності оптимізації для області чи розміру коду та швидкості. Контрзаходи проти апаратно-програмних атак не входять до складу цих алгоритмів оптимізації, вони можуть бути змінені або повністю втрачені під час цих автоматизованих процесів. Через це оцінки вразливостей і введених контрзаходів проводяться на найнижчих можливих рівнях абстракції.

Крім того, забезпечення захисту окремої архітектури апаратного забезпечення та архітектури програмного забезпечення не гарантує, що їх інтеграція в повну апаратно-програмну архітектуру зможе забезпечити необхідний рівень безпеки. Важливо знайти вразливі місця системи та надати їм оцінку до її впровадження у робочі середовища.

Оцінки витоків на ранніх етапах конструювання можуть бути проведені шляхом створення прототипу або у процесі моделювання. Прототипування та моделювання можуть бути виконані, як тільки готова частина проекту, що працює на рівні передачі реєстру апаратних компонентів. Однак фізичний підпис реалізації проекту може відрізнятися від стандартної реалізації.

З іншого боку, оцінка витоків на основі моделювання може допомогти подолати цей недолік прототипування, шляхом збільшення часу оцінки. Оцінка витоків на основі моделювання надає певні переваги, а саме дозволяє проведення пошуку джерел для кожного поміченого витоку в апаратному забезпеченні компонент та програмному забезпеченні. Відсутність гібридних інструментів і методологій для оцінки вразливостей апаратного забезпечення, разом із відповідним програмним забезпеченням, ускладнює проведення оцінки витоків. Застосування виключно перевірок маскування програмного забезпечення не гарантує безпечного кінцевого дизайну програмного забезпечення.

Як згадували дослідники з Becker et al. [24], безпека більшості

перевірених захищених замаскованих застосунків закінчуються на рівні наявності витоку живлення з бічного каналу у фізичній реалізації. Традиційно у вбудованих системах мета програмного забезпечення - забезпечення гнучкості, а мета апаратного забезпечення - забезпечення безпеки та продуктивності системи.

Головним чином, у сучасній літературі розглядаються три виклики (англ. challenges), зображені на (рисунок 1.6.).

**Виклик 1**. Збереження контрзаходів на всіх рівнях абстракції архітектури. Відноситься до внутрішньої багаторівневої структури вбудованих систем під час впровадження контрзаходів. Будь-який захисний механізм, інтегрований на вищий рівень абстракції повинен бути збережений на найнижчому рівні абстракції (кінцева система).

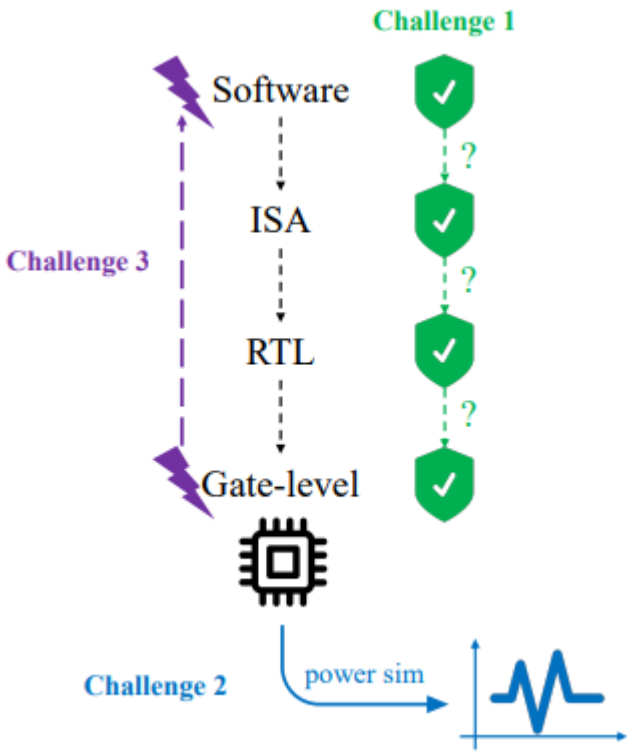


Рисунок 1.6. Виклики до захисту вбудованих систем від апаратних атак

**Виклик 2**. Симуляція та емуляція споживання енергії. Як наголошувалося раніше, для апаратно-програмної архітектури важливим є надання цілісної оцінки безпеки системи на всіх рівнях. Підходи на основі моделювання дозволяють зафіксувати повну інтеграцію апаратного та програмного забезпечення в подібні архітектури. Однією з проблем оцінки витоків на основі моделювання є моделювання потужності. *Емуляція корисного витоку*, особливо для комерційних пристроїв, для яких немає джерела дизайну апаратного забезпечення є найбажанішим підходом.

**Виклик 3**. Пошук причин спостережуваної вразливості. Одна із головних задач роботи із безпекою апаратно-програмних архітектур полягає в систематичному пошуку причин спостережуваних витоків в бічний канал, незалежно від наявності реалізації відповідних контрзаходів.

В літературі визначені наступні шляхи вирішення вищезазначених викликів:

1. Відображення важливості одночасного звернення до різних рівнів абстракції, для захисту апаратно - програмних архітектур. Для цього використовуються апаратно-програмні методи спільного проектування для захисту мікроархітектури від атак з використанням бічного каналу живлення. Цей шлях спрямований на вирішення проблеми першого виклику.

2. Розробка та виготовлення чіпа для порівняння попереднього кремнієвого бічного каналу на основі моделювання оцінки витоків. Для цього за допомогою фізичних пост-кремнієвих вимірювань, використовуються інструменти, що дозволяють моделювати енергоспоживання конструкції та показати методи зменшення потужності час моделювання без втрати точності оцінки витоку. Цей шлях спрямований на вирішення проблеми другого виклику.

3. Шлях представлення оцінки часу моделювання витоку з бічного каналу. Крім того, представлення методів точного визначення причин

витоку, що спостерігається як в апаратних засобах (на деталізаціях шлюзів), так і в програмному забезпеченні (при деталізації блоку виконання ан інструкція). Цей шлях спрямований на вирішення проблем третього виклику.

### 1.7. Сучасні технології та програмні засоби для вирішення проблем захисту програмно-апаратних архітектур

*Паралельне синхронне програмування* (англ. Parallel Synchronous Programming)(для розв'язання *виклику 1*). У типових вбудованих програмах точний час виконання програми не має значення, і цей факт є достатнім для дотримання кінцевого терміну в режимі реального часу. Однак сучасні застосунки спрямовані на інформаційну безпеку стали набагато більш чутливими до часу через ризик витоку синхронізації з бічного каналу. Хронометраж таких застосунків має бути точним і не залежати від вхідних даних.

Сучасним підходом до розв'язання проблем даного виклику є використання моделей синхронного програмного забезпечення, які виконується як $N$ паралельних потоків на процесорі з довжиною слова $N$. В даному підході кожен потік описується як однорозрядна синхрона машина з точним, без конфліктним процесом синхронізації, тоді як кожен з $N$ потоків все ще виконується як незалежна машина. Отриманий програмний застосунок підтримує детальне паралельне виконання. На відміну від попередніх робіт для отримання точної і періодично повторюваної синхронізації в програмному забезпеченні, таке рішення не потребує модифікації архітектури процесора або спеціальних методів планування інструкцій. Крім того, буде забезпечена паралельна та без конфліктна робота, що усуває проблему планування потоків.

Ключовою особливістю застосування даного підходу є те, що його

застосування не вимагає ніяких змін у апаратній архітектурі, проте використання технологій паралельного програмування веде до значного ускладнення програмної частини системи.

*Системи переписування для посилення.* (англ. rewrite to reinforce) *(*для розв'язання *виклику 1)* Атаки з впровадженням помилок є апаратними атаками, які можуть викликати помилки в програмному забезпеченні. Заходи протидії атакам на помилки можуть бути реалізовані на рівні апаратного забезпечення, програмного забезпечення або їх поєднанні. Програмні засоби протидії реалізовані вручну, не масштабуються та із значною вірогідністю можуть призвести до збільшення кількості помилок. Інструменти для вставки контрзаходів автоматично можуть бути орієнтовані на різні етапи життєвого циклу розробки.

Існують два можливі підходи до впровадження контрзаходів на бінарному рівні, наприкінці життєвого циклу розробки. Перший — це етап компілятора, де може бути реалізовано контрзахід, який автоматично зміцнює умовні гілки. Рішення застосовані на етапі компілятора, в подальшому застосовуються до проміжного представлення двійкового файлу LLVM, яке в подальшому буде скомпільовано у надійний двійковий файл за допомогою проходу компілятора. Другий підхід зосереджується на введенні контрзаходів безпосередньо в машинний код двійкового файлу за допомогою переписування.

Саме між результуючим двійковим кодом і проміжним представленням компілятора проводиться порівняння з точки зору безпеки та ефективності.

*Saidoyoki* (для вирішення виклику 2). Прогнозування рівня та можливості використання витоку бічних каналів із складної конструкції є обчислювально складним завданням. Saidoyoki - тестова платформа, яка дає функціональні можливості для оцінки витоків бічного каналу за двох різних

параметрів. Перший – це оцінка витоку побічних каналів кремнію, що вимагає використання засобів для швидкої оцінки витоку по бічному каналу базуючись на високорівневому описі проекту. Другий – це вимірювання та аналіз посткремнієвих витоків бічного каналу, і це вимірювання вимагає відповідне потрібне апаратне забезпечення.

Для створення середовища аналізу витоків побічних каналів, ця платформа надає можливість провести оцінки витоків в бокових каналах на етапі проектування (до кремнію), на етапі прототипування (після кремнію), а також під час вимірювання витоків в бічному каналі. Платформа Saidoyoki, незважаючи на те, що налаштування етапу прототипування забезпечує більше деталей для аналізу витоку в бокових каналах, у порівнянні з аналізом на етапі проектування. Оцінка витоку може бути важливим інструментом для аналізу безпеки сучасної апаратно-програмної системи.

Метод *використання середнього* (англ. Leverage the Average)(для вирішення виклику 2). Перевірка можливого витоку бічного каналу на етапі прототипування є корисним інструментом для виявлення вразливостей обладнання під час проектування, але цей процес вимагає багато трасування потужностей з високою роздільною здатністю та збільшення вартості моделювання потужності проекту.

*Кореляційний аналіз неспецифічної архітектури* (для вирішення виклику 3). У той час як оцінка витоку по бічному каналу традиційно проводиться для вже виготовленого чіпа, більш економічно ефективним було би проведення даної оцінки на етапі проектування чіпа. Дана оцінка проводиться із використанням методології ранжування шлюзів проекту відповідно до їх внеску в бічний канал витоку мікросхеми. Методологія спирається на логічний синтез, логічне моделювання, оцінка потужності та оцінка витоку на рівні шлюзів. Метрика рейтингу визначається, як специфічний тест, що проводиться співвіднесенням активностей на рівні

шлюза з моделлю витоку, або як неспецифічний тест що проводить оцінку активності на рівні шлюзів у відповідь на окремий тестовий вектор груп.

*Кореневий канал* (англ. Root Canal) (для вирішення виклику 3). Знайти першопричину витоку бічного каналу на основі живлення стає обчислювано складною задачею, у випадку застосування кількох рівнів абстракції дизайну. Хоча витік побічних каналів виникає в апаратному забезпеченні процесора, небезпечні наслідки можуть стати очевидними лише в криптографічному програмному забезпеченні, яке працює на конкретному процесорі. Аналіз витоку кореневого каналу, є методологією, яка обґрунтовує походження витоку з бічного каналу у програмному забезпеченні з точки зору базової мікро архітектури та загальної архітектури системи.

При моделюванні апаратного енергоспоживання на рівні шлюзів виконуються неспецифічні тести для виявлення логічних вентилів, які найбільше сприяють витоку по бічних каналах. Потім результати тестів розмічаються маркерами відповідниих дій у програмному забезпеченні. Отриманий аналіз може автоматично вказувати на нетривіальні причини витоків бічних каналів.

# РОЗДІЛ 2. СИСТЕМА АЛГЕБРАЇЧНОГО ПРОГРАМУВАННЯ ТА СИСТЕМА ІНСЕРЦІЙНЕ МОДЕЛЮВАННЯ

## 2.1. Визначення систем переписуючих правил

У комп'ютерних та математичних науках до технології правил переписування відносяться велика кількість методів заміни математичних під-виразів "під-термів" певної формули іншими виразами. Ця методологія реалізується за допомогою так званих "систем переписування" (часто цей термін "система переписування", використовується з такими визначеннями як механізм переписування або система редукцій). У переважній кількості такі системи представляють із себе набір об'єктів, та сукупність відповідних правил перетворення цих об'єктів.

Простіший варіант техніки переписування - недетерміноване переписування. В цьому випадку одне "переписуюче" правило застосовується до певного терма, шляхом який прописано в одній із великої кількості стратегій. Не забороняється застосування не одного а певної кількості правил. Тобто можна сказати, що система переписуючих правил не задає конкретний алгоритм зміни одного терма на інший, а визначає набір правил того яким чином певний математичний вираз перетворюється на інший [2]. Якщо розглянути у загальному визначенні системи переписування та певні алгоритми алгоритми, що мають назву стратегії переписування, то їх поєднання може бути визначено як комп'ютерна програма. В деяких джерелах літератури можна такоє знайти визначення "алгебраїчна програма". Технологія переписування математичних виразів широко використовується у сучасному програмуванні спрямованому на наукові дослідження, наприклад у системах автоматичного доведення математичних теорем.

"Абстрактна система переписування" (також у деяких літературних

джерелах зустрічаються такі визначення: “система редукції” або “абстрактна система перезапису”) — це формальне представлення, яке дозволяє охопити більшість властивостей подібних систем [3]. Найпростіший вигляд абстрактної системи переписуючих правил — це звичайна множина «об’єктів» на якій визначене бінарне відношення, що традиційно позначають як . Як правило у конкретних задачах таке просте визначення потребує додаткової конкретизації та уточнення, разом із індексацією бінарних відношень. Хоча у загальному вигляді абстрактна система переписуючих правил виглядає дуже примітивно, її зазвичай повністю достатньо для того щоб описати головні властивості будь яких систем , наприклад: нормальні форми математичних виразів, різноманітні випадки об’єднання.

Із літератури можна виділити декілька варіантів подібної формалізації, кожний із яких має власну особливість, що частково зумовлено еквівалентністю деяких визначень. Найчастіше, коли в літературних джерелах (підручниках, наукових статтях) зустрічається визначення “формалізація”, то мається на увазі формалізація визначена Жераром Юе в 1980.

“Абстрактна система редукції” (скор. АРС) — це найбільш просте (одновимірне) визначення специфікації множин об’єктів та правил, які можуть бути використані для їх перетворення [3]. У найбільш сучасних дослідженнях авторами зазвичай використовується термін “абстрактна система переписування”.(Основна увага в цьому випадку приділяється «зменшенню» виразу замість його повного «переписування», що зумовлює відхилення від звичного застосування терміну «переписування» в назвах програм та систем, що виступають у вигляді конкретних випадків абстрактної системи редукції [14]. Саме слово «зменшення» рідко можна зустріти в визначеннях спеціалізованих систем, лише у відносно застарілих

публікаціях зустрічаються випадки коли термін “система скорочення” виступає синонімом до терміну “абстрактна система редукції”.

“Абстрактна система редукції” визначається як множина, елементами якої виступають певні об’єкти. На цій множині “А” визначається бінарне відношення скорочення, яке також називають “відношення переписування”, що традиційно позначається символом “→”. Ця термінологія використовується у більшості літературних джерел, і якщо зустрічається слово «зменшення», то це може ввести в оману, оскільки результатом даної операції далеко не обов’язково буде саме “зменшення” кількості об’єктів.

Для більш повного розуміння слід зазначити що деякі підмножини переписуючих правил, тобто певні “підвідношення” відношення редукції →, можуть містити правила асоціативності та комутативності [4]. Також деякими авторами відношення редукції “→” визначається у вигляді проіндексованого об'єднання певної кількості відношень.

Якщо розглядати “абстрактну систему редукції” так само, як і системи нерозмічених переходів між станами, тобто у вигляді математичного об’єкта, і якщо саме відношення розглядати у вигляді проіндексованої множини, тоді можно сказати, що “абстрактна система редукції” - це звичайна система переходів, що мають відповідні індекси (“мітки”). У більшості досліджень головна увага приділяється саме системам переходів між станами та тому яким чином будуть інтерпретуватися мітки, що представляють із себе дії над виразами. У випадку абстрактних систем редукцій навпаки головна увага приділяється не переходам, а об’єктам що можуть бути перетворені (переписані) на подібні.

Одним із найважливіших понять теорії переписування є поняття нормальних форм виразів. За визначенням говорять, що знайдено нормальну форму відповідного об’єкта якщо його вже не можна більше переписати. В залежності від конкретної обраної системи переписуючих правил, одному

математичному виразу можуть відповідати як декілька нормальних форм, так і не відповідати взагалі жодна. На основі поняття нормальної форми алгебраїчного виразу будується велика кількість важливих властивостей систем переписуючих правил [7].

За визначенням, якщо "$(A,\rightarrow)$" - це якась абстрактна система переписуючих правил, то вираз "$x \in A$" вважається відповідною нормальною формою, у випадках якщо не існує виразу типу "$y \in A$" такого, що є можливим перехід "$x \rightarrow y$", тобто $x$ вважається "незмінним доданком".

Об'єкт "$a$" вважається "слабо нормалізованим", у випадку якщо може існувати як мінімум одна визначена множина переписуючих правил, що починається з "$a$", та у конечному результаті буде зведена до її нормальної форми. Кажуть, що системі переписуючих правил притаманна властивість "слабкої нормалізації", або така система переписуючих правил є "слабко нормалізованою" (англ. Weak Normalization), якщо кожна її складова частина є "слабко нормалізованою". Об'єкт $a$ "вважається сильно нормалізованим", якщо кожна його складова частина, яка починається з "$a$", буде зведена до нормальної форми скінченною кількістю перетворень. Абстрактна переписуюча система вважається "сильно нормалізовною", або "завершеною", або "недетермінованою" або їй притаманне відношення "сильної нормалізації" (англ. Strong Normalization), в тому випадку коли кожна її складова частина є "сильно нормалізованою".

Кажуть що системі переписуючих правил притаманна властивість "нормальна форма" (англ. Normal Form), в тому випадку коли для всіх її складових частин "$a$", відповідні "нормальні форми $b$" можуть бути виведеними з "$a$" через послідовність прямих та зворотних перетворень, лише в тому випадку коли вираз "$a$" може бути приведено до виразу "$b$". Системам переписуючих правил притаманна властивість "унікальної

нормальної форми” (англ. Unique Normal Form), в тому випадку коли для кожної відповідної нормальної форми “*a” та “b”*, форма “*a”* може бути отримана із форми “*b”* через застосування послідовності застосування правил переписуваня та систем зворотних перетворень, тільки в тому випадку коли форма “*a”* дорівнює формі “*b”*. Системам переписуючих правил притаманна властивість “унікальної нормальної форми” стосовно редукції виду “*(UN→)”*, в тому випадку коли для кожного алгебраїчного виразу, який можна звести до вигляду “нормальної форми” “*a* і *b”*, вираз “*a”* тотожно дорівнює виразу “*b”*. Серед властивостей “нормальна форма” та “унікальна нормальна форма”, особливо виділяється поняття “канонічна форма” [11].

У математиці та інформатиці “канонічна”, “нормальна” або “стандартна” форма математичного виразу - це загальновизнаний варіант математичного представлення будь-якого програмного об’єкта. Як правило така форма призначена для забезпечення найпростішого вигляду необхідного об’єкта, та для однозначної ідентифікації цього об’єкта. Відмінності між «канонічною» та «нормальною» формами залежать від конкретного представлення “поля”, або в окремих випадках “підполя”. Для більшості математичних об’єктів “поле”, можна визначити канонічну форму як відповідне унікальне представлення кожного конкретного об’єкта, ц той час як “нормальна форма” просто визначає форму виразу без дотримання будь яких вимог щодо унікальності.

Наприклад: канонічна форма звичайного натурального числа в десятковому виді визначається у вигляді кінцевої множини цифр, що йдуть строго послідовно одне за одним, та не містять “нуль”. Взагалі, для будь якої множини об’єктів, на котрій визначено математичне відношення “еквівалентності”, спосіб побудови канонічної форми полягає у виборі необхідного конкретного об’єкта із практично кожного класу. Наприклад:

“Жорданова нормальна форма” — це канонічна форма відношення “подібності”, визначена на кільці “матриць”.

“Форма рядкового ешелону” — це канонічна форма за якої еквівалентно рівними вважаються матриці разом із їх лівими добутками з оберненими матрицями [3].

В “комп'ютерних науках”, а саме в “комп’ютерній алгебрі”, для представлення математичних об’єктів в комп’ютерній пам’яті, існує безліч варіантів визначення, перетворення та представлення одного і того ж об’єкта.В такому випадку “канонічна форма” виступає в якості “унікального ключа”, тобто об’єкта із унікальним представленням (у сукупності із процесом канонізації, тобто таким що забезпечує приведення будь якого виразу до його “канонічної форми”). У цьому випадку, коли необхідно перевірити рівність двох об'єктів, це може бути легко перевірено простим порівнянням “канонічних форм” цих об'єктів.

Навіть враховуючи таку сильну властивість “канонічних форм”, вони як правило залежать від довільних, або непередбачуваних умов (“наприклад, упорядкування змінних”), що може доставити певні складнощі для процесів, які відповідають за перевірку рівності об’єктів. Така негативна риса хумовлюється незалежними обчисленнями. Через це в “комп’ютерній алгебрі” нормальні форми вважаються “слабким поняттям”: “нормальна форма — це таке представлення, яке представляє нуль однозначно”. Таке поняття дозволяє провести перевірку рівності, шляхом розміщення різниці нормальних форм цих об’єктів[13].

Під “канонічною формою” природним (канонічним) способом також може визначатись “диференціальна форма”.

Вирішення задачі розпізнавання “канонічних форм” є дуже вигідним для практичного програмування. Вирішення цієї задачі розглядається у вигляді алгоритму, який визначає шляхи переходу із “*s”* в “*S”* до відповідної

“канонічної форми” “s*”. Як правило “канонічні форми” як правило використовуються для спрощення та підвищення ефективності роботи із відповідними “класами еквівалентності”. Наприклад; для задач “модульної арифметики” у вигляді “канонічні форми” для класів залишків від ділення як правило береться “найменше невід’ємне ціле число”. Операції над класами визначаються через об’єднання таких представлень, із подальшим зведенням їх результатів до його “найменшого не від’ємного залишку”. Вимоги унікальності пом’якшуються у деяких особливих випадках, що дозволяє “канонічним формам” бути унікальними у точності до “найбільш тонкого відношення еквівалентності”, наприклад шляхом припущення можливості зміни порядку слідування математичних виразів ( у випадку якщо “природній порядок” математичних виразів не визначено) [15].

“Канонічна форма” може бути звичайним припущенням або бути визначеною відповідною “складною теоремою”. “Наприклад”, традиційна форма полінома - запис у вигляді що складається із доданків в спадних степенях. Більш звичною формою запису є , а не , не зважаючи на еквівалентність цих двох форм. В іншому випадку, існування “жорданової канонічної форми” для поля матриць є “складною теоремою”.

## 2.2. Система Алгебраїчного Програмування

Парадигма “алгебраїчного програмування” визначається як програмування що засноване на застосуванні систем переписування алгебраїчних виразів. “Алгебраїчне програмування” - це розширення “функціонального програмування”, що застосовується для алгоритмічного та програмного розв’язання типових та нетипових задач “комп'ютерної алгебри” (таких як “проблема слів у певних алгебрах”, алгоритми “поповнення Кнута-Бендикса” або “Бухбергер”а). Також “алгебраїчне програмування” певною мірою застосовуються до розв’язання задач, що

пов'язані з “операційною семантикою” мов програмування. До цих задач відносяться задача “виконання алгебраїчних специфікації компонентів програмного забезпечення”, задача “визначення операційних семантик мов програмування”, задачі пов'язані з розробкою “інтерпретаторів” та “прототипів компонентів програмного забезпечення” та ін.) [4].

У відмінності від “традиційного підходу”, що зосереджується на простому застосуванні “канонічних систем правил переписування” з «очевидними» стратегіями, в Системі Алгебраїчного Програмування реалізовано інструментальні засоби для об’єднання будь-яких систем “переписуючих правил” та різноманітних “стратегій переписування”.

Цей підхід надає дуже великі можливості програмам, що застосовують механізми “переписування”, через те що значно зростає “гнучкість” та “виразність”. До “системи алгебраїчного програмування” інтегруються та поєднуються цілих чотири головні “парадигми програмування” у такий спосіб, що головна частина алгебраїчного застосунка може бути виконана та реалізована у вигляді як “системи правил переписування” так і “імперативної” реалізації алгоритма. У свою чергу підходи “функціонального програмування” використовуються для визначення “стратегій переписування”. Парадигма “логічного програмування” також може бути реалізована на основі систем переписуючих правил, оскільки “система алгебраїчного програмування” має вбудовані алгоритми та процедури “уніфікації”. [2]

“Алгебраїчне програмування” може бути визначене як “програмування, що засноване на переписуванні”. “Алгебраїчне програмування” розширює функціональні можливості звичайного імперативного програмування та може бути застосоване до розв’язування задач комп’ютерної алгебри, в операційних семантиках мов програмування (виконанні алгебраїчних специфікацій програмних компонентів, визначення

операційної семантики мов програмування, розробка інтерпретаторів і прототипи програмних компонентів). [4]

“Система алгебраїчного програмування” розроблена на кафедрі 100,105 Глушкова Інститут кібернетики НАН України у 1987 р. “Система алгебраїчного програмування” може вважатись першою подібною системою переписування термів у якій застосування “систем переписуючих правил правил” та “стратегій переписування” виконувалось в окремих потоках.

Історично можна виділити три основних “масштабних” реаліза “системи алгебраїчного програмування” APS:

- “APS v.1” в 1987 році;
- “APS v.2” в 2004 році;
- “APS v.3” в 2009 році.

## 2.3. Транзиційні системи та алгебри поведінок

Загальноприйнятим засобом для опису динаміки систем у сучасній комп'ютерній науці є поняття *транзиційної системи*. Транзиційна система визначається безліччю станів та відношенням переходів. Зазвичай це поняття збагачується додаванням різних додаткових структур, найважливішою з яких є розмітка переходів (розмічені транзиційні системи, запроваджені Парком [19] для опису поведінки автоматів на нескінченних словах). У парадагмі інсерційного моделювання, у якості базового поняття можна взяти визначення “атрибутної транзиційної системи” [11], яке формально визначається як п'ятірка

$$“< S , A , U , T , j > \qquad (1)”$$

“де $S$ – множина станів”, “$A$ – множина дій, які використовуються для розмітки переходів”, “$U$ – множина атрибутних розміток, які використовуються для розмітки станів”, “$T$ – відношення переходів”. Така базова частка структури повністю відповідає базовому визначенню

“розміченої транзиційної системи” (із скритими переходами).

Функція $j : S \rightarrow U$ називається функцією розмітки станів. Зазвичай $U$ визначається як множина $U = D^R$ відображення множини $R$ атрибутів у множину даних $D$ (область значень атрибутів), або у випадку з типізованими даними. Для символьного моделювання у якості атрибутних розміток $U \subseteq L(R)$, використовуються формули певної логічної мови $L(R)$, де $R$ - множина атрибутів.

Зазвичай це інтерпретована чи не інтерпретована мова першого порядку, можливо розширена деякими модальностями темпоральної логіки. У разі символьного моделювання, розмітки станів зазвичай розглядаються з точністю до деякої певної еквівалентності.

Транзиційні системи можуть також бути налаштовувані шляхом виділення відповідних конкретних “множин станів” із загальної “множини станів $S$”. Серед них найважливішими є наступні множини:

- множина “початкових” станів;
- множина “заключних” станів;
- множина “невизначених” станів.

Останні використовуються теорією у якості інструмента визначення відношення апроксимації та побудови нескінченних систем у вигляді кінцевих меж.

Еквівалентність транзиційних систем. Як і в теорії автоматів, стани транзиційних систем розглядаються з точністю до деякої еквівалентності. У спектрі різних еквівалентностей, що розглядалися в літературі [20], найважливішими є трасова та бісимуляційна еквівалентності (найсильніша та найслабша в спектрі, відповідно).

## 2.4. “Система Інсерційного моделювання”

“Інсерційне моделювання” — це один із підходів теорії “проектування

складних систем", в основі якої лежить елементарна взаємодія "агентів" та "середовищ". З технічної сторони цей підхід базується на "алгебрі процесів". Головне призначення "інсерційного моделювання" - проведення процесу "уніфікації" різноманітних моделей взаємодіючих систем, та проведення процесу "уніфікації" відомих моделей обчислень (таких як "CCS", "CSP", "π-calculus", "mobile ambients" тощо). Останніми роками "інсерційне моделювання" успішно використовувалось для проведення "глибоких" перевірок "специфікацій вимог" для "розподілених паралельних систем" із повністю різних предметних областей, до яких входили:

- телекомунікаційні системи;
- задачі телематики;
- задачі, і яких застосовуються розподілені та паралельні алгоритми [17].

На базі "інсерційних програм" компанією "Kiev VRS-група" було створено "систему верифікації" "VRS", що використовувалась у проектах перевірки програмного забезпечення, та генерації тестових випадків для компанії "Motorola".

Системою "інсерційного моделювання" використовуються головні функціональні можливості підходу моделювання у "базових протоколах" для створення формальної "специфікації користувацьких та системних вимог" до "розподілених" та "паралельних" апаратно-програмних систем. Базові протоколи – це параметризовані MSC (діаграми послідовності повідомлень) з передумовами та постумовами, які інтерпретуються відповідно до станів середовища із зануреними у нього агентами [19].

Семантично "базовий протокол" визначається у вигляді програмного виразу "*x*" *(a < u > b )* певного типу динамічної логіки. Даний оператор "x" має певний конкретний перелік параметрів, вирази "*a"* і "*b"* — визначають "передумову" та "постумову" досліджуваного процесу, а "*u"* – власно

досліджуваний процес, що задається шляхом визначення діаграми “послідовності повідомлень”. “Передумова” та “постумова” визначаються як формули “багатосортової мови першого порядку”, яка має назву “базова мова”. “Базова мова” необхідна для процесу визначення та розмічення властивостей та “станів” системи, що представляються у вигляді поєднання (“композиції”) середовищ та агентів. “Атрибути середовища” - це “змінна частина” системи, яка змінюється протягом часу та розвивається. Вона представляється у вигляді “функціональних” та “предикатних” символів основної мови. В конкретному процесі “*u”* описується кінцева поведінка відповідного конкретного середовища до у яке “занурено” (“вставлено”) певну кількість агентів. Випадок із фіксованою кількістю параметрів конкретного “базового протоколу”, називається “екземпляром базового протокола” [1].

В процесі опису середовища, необхідно визначити “сигнатуру базової мови” та задати можливі обмеження. Цей опис може бути інтерпретовано різними варіантами, так певна конкретна частина опису може бути інтерпретована спочатку, у випадку коли вона представляє числові функції та предикати, або “конструктори станів агентів”. Опис також може містити відповідні конструктори для конкретних поведінок “агентів”, які з часом “занурюються” у дане середовище. Застосування набору “базових протоколів” надає можливості для явного визначення вимог до “поведінки системи”, а також для неявного визначення “функції занурення” (англ. insertion function) конкретного середовища [18]. Формальне вираження вимог має наступний вигляд: у випадку “дійсності передумови" деякого створеного протоколу і початку процесу виконання цього протоколу, визначаємо що після успішного завершення модельованого процесу також буде досягнуто “дійсну постумову”.

Семантикою “базових протоколів” визначається велика безліч

найрізноманітніших конкретних реалізацій “базових протоколів”, які будуть повністю задовольняти будь-які формальні властивості. Взагалі таку реалізацію можна представити у вигляді звичайного “атрибута системи переходів” - розміченої конкретної системи переходів із визначеними “діями”, “станами” та “мітками” атрибутів.

Задача “інсерційного моделювання” - побудова моделей систем будьякої складності, а також дослідження механізмів та алгоритмів взаємодії “агентів” та “середовищ” у складних “багатоагентних системах” (розподілених або паралельних) [21]. В літературі можна зустріти багато неформальних визначень головних положень “парадигми інсерційного моделювання”. Узагальнюючі, ці положення можна визначити у наступному вигляді:

1. “Світ є ієрархія середовищ та агентів, занурених у ці середовища”.

2. “Агенти та середовища є сутностями, що розвиваються протягом часу та мають поведінку яку можна спостерігати”.

3. “Занурення агента в середовище змінює поведінку цього середовища та породжує нове середовище, яке готове до занурення в нього нових агентів (якщо для них є місце в цьому середовищі)”.

4. “Середовище, яке розглядається як агент, може також бути занурене в середовище верхнього рівня”.

5. “Агенти можуть занурюватися в середовища середовищ верхнього рівня, а також породжуватись внутрішніми агентами, які вже є зануреними у середовище раніше”.

6. “Агенти та середовища можуть моделювати інші агенти та середовища на різних рівнях абстракції”.

Як правило у якості “агентів” та “середовищ”, визначаються повністю технічні та програмні системи. Проте сама парадигма не обмежується лише комп’ютерними науками та програмуванням. “Інсерційні модулі” можуть

бути створені на основі систем реального світу – фізичних, біологічних та соціальних. Взаємовідносини, що є найбільш цікавими для моделювання та дослідження – це у першу чергу інформаційно комунікаційні системи, повністю абстраговані від фізичного представлення [3]. Якщо переходити до чіткого математичного визначення, то під поняттям агента розуміється максимально абстрактний математичний об'єкт, задача якого визначити поведінку системи, яка змінюється протягом часу.

У термінах "інсерційного моделювання" агент визначається як "розмічена транзитивна система, стан якої визначається з точністю до бісимуляційної або трасової еквівалентності". Головна перевага поняття "транзитивної системи" у порівнянні з іншими моделями є розділення системи на "частини що спостерігається" (саме вона виражається засобами "алгебри поведінок"), та "прихованої частини" системи, яка визначається лише її внутрішніми станами [23].

Середовище - це агент, який має визначену "функцію занурення". Таким чином можна визначити, що до будь-якого "середовища *E"* може бути занурено будь-який "агент *A*" із відповідною безліччю дій. Через те що стани "транзитивних систем" визначаються та розглядаються з точністю до "бісимуляційної еквівалентності", вони можуть бути ототожнені з "поведінками". У цьому випадку можна говорити про безперервність "функції занурення". Саме "безперервність функції" занурення є головною вимогою до будь якого "середовища інсерційної моделі". Це припущення дозволяє зробити декілька дуже корисних висновків. По Перше той факт, що "функцію занурення" можна визначити у вигляді найменшої нерухомої точки системи функціональних рівнянь. Результатом роботи функції занурення "*Ins(e,u)"* є процес "занурення агента", що перебуває у стані "*u"*, до середовища, яке в свою чергу перебуває в стані "*e"*. Цей процес позначається як "*e[u]"*. Беручи до уваги той факт, що будь яке середовище

у свою чергу також є агентом, воно може бути занурене в середовище верхнього рівня, що призводить до створення багаторівневих середовищ [14].

Необхідність визначення “атрибутного середовища” випливає з того факту, що у випадку моделювання конкретних “практичних” моделей визначення агентів та середовищ - надто абстрактні, та ігнорують структуру станів самих “агентів та середовищ”. Також після досягнення термінального стану інформація про всі попередні стани середовища повністю втрачається, у випадку конкретної структури. Вся необхідна інформація може бути передана описами дій агентів та середовищ, проте більшості випадків це призведе до неприродного та занадто ускладненого вигляду. Для вирішення цієї проблеми, визначається поняття “атрибутної транзитивної системи”. “Атрибутні транзитивні системи” відрізняються від звичайних розмічених наявністю міток не лише на “переходах”, а також на “станах”. “Алгебра поведінок” для “атрибутних систем” відрізняється тим, що “поведінки” можуть бути розмічені “атрибутними мітками”. Щоб досягти цього вводиться додаткова операція крім префіксингу. “Операція розмітки” дозволяє всю необхідну інформацію щодо стану середовища зберігати та передавати через її відповідну розмітку. “Операція розмітки” також дозволяє існування великої кількості “термінальних констант”, які відповідають:

- стану успішного завершення;
- стану глухого кута (deadlock);
- будь якому іншому необхідному стану.

Подібне вдосконалення стає можливим через те що усі переходи до цих станів можуть мати різні розмітки. “Атрибутні середовища” визначаються та будуються з урахуванням певної визначеної логічної бази. Логічна база атрибутного середовища” включає:

- набори типів змінних (“цілі”, “речові”, “перераховані”, “символьні”,

“поведінки” та ін.), які можуть бути проаналізовані на відповідних диапазонах значень;

- символи для позначення констант із відповідних діапазонів;
- набори “функціональних” та “предикатних” символів, що мають конкретний тип.

Частина “логічної бази” може бути інтерпретована (наприклад, “арифметичні операції”, “нерівності”, “рівність для всіх типів” та ін.)[25]. “Функціональні” та “предикатні” символи, що не можуть бути інтерпретовані мають назву “атрибути”. “Неінтерпретовані функціональні символи" 0-ї арності мають назву “прості атрибути”, решта – “функціональні атрибути” (“неінтерпретований предикатний символ розглядається як функціональний з бінарною областю значень”). “Функціональні символи” як правило застосовуються для визначення структур даних, таких як масиви, списки, дерева.

Логічна база “атрибутного” середовища є базовою мовою, зазвичай “першого порядку”, у деяких випадках, з “кванторами”. В особливих випадках склад логічної базу збагачується шляхом додавання модальностей темпоральної логіки. Під “атрибутним виразом” мається на увазі “простий атрибут” або “атрибутний вираз”, до складу якого входять “прості атрибути”. У випадку якщо всі складові частини “атрибутного виразу” є константами, то весь “атрибутний вираз” називається “константим”. Ядро “атрибутного середовища” складають формули “базової мови”. “Атрибутні середовища” поділяються на два класи:

- конкретні атрибутні середовища;
- символьні атрибутні середовища.

“Специфікація програм та систем” (програмних та технічних) виконується засобами “темпоральної”, “динамічної” та деяких інших видів “модальних логік” [18]. Проте, для більшості випадків зазначені

математичні механізми застосовуються у випадках, із визначеною докладною моделлю досліджуваної системи та розв’язують задачу “перевірки властивостей цієї системи, заданих у логічній формі” (англ. “model checking”).

Інший підхід до визначення “вимог та специфікацій систем” полягає у визначенні “локальних поведінкових властивостей” описуваної системи. Математичне представлення говорить про властивості “відносин переходів транзитивної системи”, а у випадках представлення системи у вигляді “поєднання” (“композиції”) “середовищ та агентів”, мається на увазі визначення “функції занурення”. Досліджувана поведінка системи має назву “процес”. Його атрибути - це передумови та постумови. Ці атрибути задають значення основних параметрів, і загальну “поведінку системи”. “Локальні властивості” розглядаються у термінах “темпоральниї логік”, що відображає задоволення стану (для відповідних значень параметрів) системи певним визначеним умовам. У цьому випадку поведінка може бути попередньо визначеною після остаточного успішного завершення процесу розмітки нових станів, мета якого - задоволення визначених умов. Цей процес є повністю аналогічним до подібних “функцій розмітки” визначених для трійок Хоара, що виражаються формулами “динамічної логіки”, та локальними властивостями досліджуваних систем.

“**Абстракції”**. У випадках необхідності дослідження складних розподілених систем, таких як “телекомунікаційні системи”, мережі типу “Інтернет”, багатокомпонентні багатопроцесорні системи, процес опису усіх станів стає занадто складним або взагалі неможливим. Традиційний підхід у таких випадках полягає у заміні станів досліджуваних систем відповідними “абстрактними об'єктами”. В методології “інсерційного моделювання”, для проведення процесів абстракції “великих систем” використовуються “атрибутні середовища”, які складаються із “станів” розмічених

математичними виразами “базової мови”. У випадках коли стани так само представляються математичними виразами, їх визначають як “символьні атрибутні моделі” [16]. Для дослідження співвідношень та взаємовідносин між абстрактними і конкретними “атрибутними інсерційними моделями” було введено процеси “абстракції” та “конкретизації” відповідних “атрибутних транзитивних систем”.

“**Верифікація”**. Реалізована як складова частина “системи VRS” “мова базових протоколів”, допускає використання атрибутів числових та символьних типів (“вільні терми”), масивів, списків та функціональних типів даних. забезпечує Доказ алгебраїчних виразів та тверджень у теорії першого порядку забезпечується дедуктивною системою, яка інтегрує теорії “цілочисельних та мовних лінійних нерівностей”, теорії “перелічуваних типів даних”, “вільних (неінтерпретованих) функціональних символів” та “теорію черг”. “Дедуктивна система” успішно виконує процеси “доведення” заданих тверджень, або спростовує відповідні набори та класи формул. Через таку особливість “дедуктивної системи” , під час процесу верифікації можуть бути отримані “стани відмови” для деяких окремих проміжних запитів. На практиці подібні “відмови” трапляються відносно рідко і не мають вагомого впливу на кінцевий результат дослідження [22].

У якості “мови процесів” використовується мова “MSC”, яка збагачена додаванням “інсерційної семантики”. У системі також допускається використання операторів “мови SDL” та відповідних представлень у вигляді “діаграм мови UML”.

Окремо слід зазначити частину “системи VRS” яка призначена для оцінки характеристик “повноти” та “несуперечності” досліджуваних “базових протоколів”:

- визначення “несуперечливості базових протоколів” наступне: “два базові протоколи називаються несуперечливими, якщо за будь-якої

конкретизації цих протоколів їх передумови не можуть бути одночасно істинними". "Стан суперечливості" двох протоколів визначається наступним чином: "для деяких станів вибір протоколу, який застосовується в цих станах, відбувається недетермінованим чином". Подібний "недетермінізм" може бути "небажаним", і в цьому випадку "інсерційна модуль" повідомить розробнику про суперечливість у вигляді відповідного попередження.

- визначення "повноти базових протоколів" наступне: Система базових протоколів називається повною, якщо для будь-якого конкретного стану існує хоча б один базовий протокол, що можна застосувати до цього стану" [27].
- визначення "неповноти базових протоколів" наступне: у досліджуваній системі існує можливість виникнення "тупикового стану" (deadlock).

У випадку доведення "несуперечливості" та "повноти" досліджуваної модулі перевірка закінчується. У випадку знаходження стану "суперечності" або "неповноти" який не був попередньо визначений розробником через те, що не має можливості зробити висновок відносно, досяжності відповідного стану, постає нова задача – "перевірка недосяжності" умови, яка відображає стан в якому порушуються вимоги "несуперечності" чи "повноти". Тому що "недосяжність" призводить до виникнення "інваріантності її заперечення", то стає можливим повторне застосування алгоритмів та засобів "статичного аналізу", з метою доведення конкретних необхідних властивостей. У випадку успішного виконання поставленої задачі процес моделювання та "перевірки моделі" вважається завершеним. В іншому випадку можливо застосувати механізми посилення певних властивостей або застосувати "динамічну верифікацію", "конкретну або символьну генерацію" трас, з метою доведення або спростування моливості досяжності відповідних

станів.

В процесі впровадження “інсерційних моделей” певного класу застосовується застосунок-інтерпретатор моделей, який має назву “інсерційна машина” [5]. До складу “інсерційної машини” входять три головних компонента:

1. “Модельний драйвер”. - задача якого полягає у “керуванні рухом моделі по дереву її поведінки, обчислюючи переходи з поточного у новий стан”.

2. “Розв’язувач” (англ. unfolder) поведінок агентів. Конкретний поточний стан моделей - це “алгебраїчного виразу у розширеній алгебрі поведінок”. Вхідною мовою допускається визначення та виконання “рекурсивних стратегій” для визначення поведінок “агентів” та відповідних структур даних детального визначення “поведінки” або “стану” конкретного середовища. “Анфолдер” поведінок використовує такі “визначення” побудови “розкладу” стану.

3. “Інтерактор середовища”. Відповідає за приведення стану середовища до “нормальної форми”, спираючись на “сигнатуру” та опис “функції занурення”. Після завершення “процесу приведення до нормальної форми” модулю “драйвер моделі” залишається лише обрати напрямок руху по дереву станів та здійснити відповідний перехід або зупинитися.

Існує два типи “інсерційних машин”: “інсерційні машини” реального часу або “інтерактивні інсерційні машини” та “аналітичні інсерційні машини”. “Інсерційні машини реального часу” виконуються у реальному чи віртуальному середовищі, шляхом взаємодії із “зовнішнім середовищем” у реальному часі. “Аналітичні інсерційні машини” призначені для проведення наступних процесів:

- аналіз моделей;
- дослідження властивостей моделей;

- розв'язання задач на інсерційних моделях тощо.

Так само як і “інсерційні машини” “модельні драйвери” також поділяються на “інтерактивні” та “аналітичні”. “Інтерактивний драйвер” після завершення процесу “нормалізації стану” обирає лише один напрямок для переходу, та виконує його, тобто передає свій стан та дії до зовнішнього середовища. “Інтерактивна машина” з точки зору “зовнішнього спостерігача” функціонує як “агент”, який було “занурено” у “зовнішнє середовище з функцією занурення”. Це і визначає відповідні “закони функціонування” відповідного середовища. “Зовнішнє середовище”, здатне змінювати “префікс поведінки” стану “внутрішнього середовища” відповідно до своєї “функції занурення”. “Інтерактивний драйвер” може мати дуже складну конфігурацію. До його складу можуть входити критерії “успішного функціонування”, що працюють як “інтелектуальна система”, збираючи інформацію про минуле, та створюючи відповідні моделі “зовнішнього середовища” з метою покращення існуючих алгоритмів прийняття рішень щодо вибору дій, та створення нових. Також до складу, інтерактивного драйвера може входити відповідний інтерфейс, що забезпечує процеси обміну фізичними сигналами із “зовнішнім середовищем” (“наприклад, прийом візуальної або акустичної інформації, інформації про стан у просторі тощо”) [12].

“Аналітична інсерційна машина”, у відмінності від інтерактивної, бере до уваги різні варіанти прийняття рішень стосовно конкретних дій, та має можливість повертатися до точок вибору, розглядаючи різні шляхи в “дереві поведінки системи”. До складу моделі системи може входити узагальнений опис “зовнішнього середовища” для досліджуваної системи. Головна задача “аналітичної машини” - здійснення пошуку необхідних станів, які мають відповідні необхідні властивості (“досяжність цільових станів”) або конкретні стани, у яких певні властивості не виконуються.

# РОЗДІЛ 3. ІНСЕРЦІЙНА МОДЕЛЬ АНАЛІЗУ ПРОГРАМНО - АПАРАТНИХ АТАК

## 3.1. Загальна постановка задачі.

Для складних архітектур, таких як x86-64, задача групування бітів в двійковий файл для формування повних інструкцій, є обчислювально складною (має алгоритмічну складність більшу за $O(n)$ ). Також обчислювально складною є задача групування інструкції для форми основні блоків на цьому рівні абстракції. Таким чином, маніпулювання бінарним файлом безпосередньо не є тривіальним. В даному науковому досліджені пропонується процедура, у якій використовується дизассемблер з відкритим кодом, інструменти бінарної маніпуляції, а також бінарні підйомники для здійснення більш керованого процесу бінарного зміцнення.

## 3.2. Переписування двійкового коду

Використання застосунків - дизассемблерів та робота над асемблерним (машинним) кодом у порівнянні з бінарним файлом, може допомогти знайти шаблони інструкцій і застосувати виправлення локально. Однак на рівні коду складання, виділення регістрів і використання пам'яті виправлено. Тому застосовуючи виправлення на цьому рівні вимагають особливої обережності, щоб не переписати виділені регістри, які використовуються.

Сприятливою властивістю цього рівня ієрархії є однозначна інформація про розподіл функціональних можливостей між архітектурними модулями системи. Ця властивість використовується у запропонованій методології та для побудови ітераційного процесу, який, використовуючи моделювання ефектів несправності може локально застосовувати контрзаходи лише до тих частин, яким вони потрібні.

У той час як прості та невеликі за масштабом виправлення можуть

бути застосовані на рівні збірки, складніші виправлення вимагають застосування на вищих рівнях абстракції. У цьому випадку рівень абстракції, який дозволяє модифікацію коду та застосування різних корисних типів аналізу. Оскільки Low Level Virtual Machine-IR знаходиться в форматі Single Static призначення (SSA) [75] і підтримує різні рівні ієрархії (а саме модуль, функція, базовий блок і інструкція), його обрано у якості високорівневої віртуальної машини. Підтримка різних рівнів ієрархії в Low Level Virtual Machine-IR полегшує виконання статичного аналізу в програмі. Крім того, участь у ланцюжку інструментів Level Virtual Machine-IR має перевагу відкритого коду, оскільки містить велику кількість активних учасників і добре підтримувану документацію. В даній роботі розглядається модель “винуватець та латник” (англ. “faulter and the patcher”). (рисунок 3.1.)

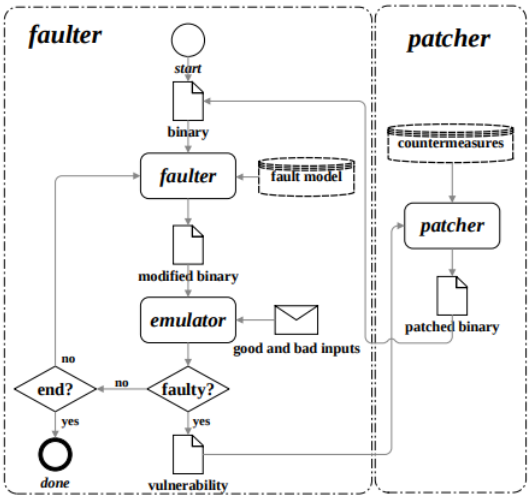


Рисунок 3.1. Загальна схема моделі “винуватець та латник”

**“Винуватець” (англ. faulter)**. Для нашої мети ін’єкції помилок – це вразливості, де зловмисник, тобто неавторизований користувач, може ініціювати поведінку в цільовому двійковому файлі, який слід зарезервувати тільки для авторизованих користувачів.

Наприклад, розглянемо перевірку пін-коду, яка отримує вхідні дані pin

і перевіряє правильність встановленого значення. Для правильного PIN-коду програма продовжується і може перейти до виконання деяких чутливих операцій. Зловмисник не знає правильного пін-коду, але може мати можливість пропускати інструкцію в цільовому двійковому файлі, з метою примушення застосунку до прийняття “вставлений штифт” як правильний, і тому виконати чутливі операції. Ці несправності позначені як «успішні помилки». Помилки, які не викликають небажаної поведінки або викликають програму до збою ігноруються.

На першому етапі обирається модель помилки, захист від якої необхідно надати бінарному файлу. Наприклад, на прикладі перевірки пін-коду визначаються два варіанти вхідних даних:

«хороший» вхід — це введення правильного пін-коду;

«поганий» — це будь-яке значення, крім правильного пін коду.

Під час запуску цільового двійкового файлу з «поганим» входом є можливість ефективно записати сліди всіх виконаних інструкцій. Для кожного зсуву в цій трасі, беручи за приклад «однобітну модель перевороту», драйвер моделі запускає модель введення\виведення цільового двійкового файлу.

Модель цільового двійкового файлу або перейде у стан аварійного завершення роботи (*deadlock state* у термінах інсерційного моделювання), виконується як неправильний вхід або поводиться інакше (як правильний вхід). Якщо поведінка моделі відповідає правильному входу, та виникає помилка, то записується стан моделі який їй відповідає. У цьому випадку необхідно записати відповідну трасу моделі у відповідний файл.

**“Латник” (англ. Patcher).** Список «успішних помилок», що надходять від “винуватця”, адресуються локально у відповідній частині моделі - патчері. Задача патчера полягає в заміні вразливих шаблонів

інструкцій на відомі захищені шаблони.

Після одного запуску моделі “faulter-patcher” буде отримано частково виправлений двійковий файл. Повторний запуск моделі для цього файлу може виявити додаткові вразливості на вже виправленому двійковому файлі оскільки в процесі виправлення було додано новий код і змінено. Для виправлення цих нових вразливостей проводиться повторний запуск латника доки не будуть досягненні певні фіксовані умови.

### 3.3. Інсерційна модель для задачі “винуватець та латник”

Першим кроком для опису інсерційної моделі для задачі “винуватець та латник”, є визначення кінцевих точок. Для даної моделі вони є очевидними:

- *Delta* - стан успішного завершення для даного прикладу означає, що модель апаратно-програмної системи знаходиться в такому стані, що не містить вразливостей (ідеальний випадок), або містить незначну кількість не критичних вразливостей.
- *bot* - стан при якому, кожен повторний запуск моделі не знаходить нових вразливостей, та не виконує операцій з виправлення, проте немає достатніх доказів для того, щоб стверджувати що вразливості повністю відсутні.
- *0 (deadlock)* - тупиковий стан, означає критичну вразливість. В більшості випадків подальше виконання моделі не матиме сенсу, поки ця вразливість не буде виправлена.

Початковим станом є базова архітектура апаратної частини, а функція занурення, в даному випадку складається із двох частин, перша виконує ін’єкцію помилки, а інша відповідає за процеси аналізу та виправлення.

## ВИСНОВКИ

За результатами проведеного дослідження було розроблено модель для аналізу найбільш розповсюдженого типу програмних атак на апаратну частину програмно-апаратних систем, заснованих на мікроконтролерах - атаки типу “винуватець-латник” (англ. faulter-patcher). Розроблено головні частини інсерційної моделі, що аналізує прототипи майбутніх мікросхем на наявність відповідних вразливостей та описані стани завершення та тупикові стани моделі.

Створена інсерційна модель дозволить повноцінне виявлення програмних та апаратних вразливостей проекту розроблюваного мікрочіпу на рівні проектування, до передачі мікроконтролерів безпосередньо до виробництва, та може бути використана у якості базисної основи для розробки моделей для аналізу, виявлення та реалізації механізмів запобігання та протидії іншим типам програмних атак на апаратну частину мікроконтролерів.

Напрямами подальшого дослідження за цією темою є розробка інсерційних моделей для аналізу атак, що є значно складнішими до виявлення, особливо атаки пов’язані із витоком з бічного каналу на основі живлення.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. C.A.R. Hoare. Communicating Sequential Processes. Prentice Hall, New York, 1985.

2. L. Aceto, Wan Fokking, and C. Verhoef. Structural operational semantics. In J. A. Bergstra, A. Ponce, and S. A. Smolka, editors, Handbook of Process Algebra, pages 197– 292. North-Holland, 2001.

3. Letichevsky and D. Gilbert. A Model for Interaction of Agents and Environments. In D. Bert, C. Choppy, P. Moses, editors. Recent Trends in Algebraic Development Techniques. Lecture Notes in Computer Science 1827, Springer, 1999.

4. Letichevsky, J. Kapitonova, A. Letichevsky Jr., V. Volkov, S. Baranov, V.Kotlyarov, T. Weigert. Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. Computer Networks, 47, 2005, 662-675.

5. M.A. Reniers. Message Sequence Chart: Syntax and Semantics. Eindhoven, University of Technology, 1998.

6. International Telecommunications Union. Recommendation Z.120 Annex B: Formal semantics of Message Sequence Charts, 1998.

7. A.A Letichevsky, J.V. Kapitonova, V.P.Kotlyarov, A.A.Letichevsky Jr, V.A.Volkov. Semantics of timed MSC language, Kibernetika and System Analysis, 2002.

8. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^20 states and beyond. Information and Computation, 98 (2): 142–170, 1992.

9. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In Proc. of ASE 2001, 382–386, November 2001.

10. S. Abramsky. A domain equation for bisimulation. Information and Computation, 92(2):161–218, 1991.

11. P. Aszel and N. Mendler. A final coalgebra theorem. In LNCS 389, pages 357–365. Springer-Verlag, 1989.
12. M. Roggenbach and M. Majster-Cederbaum. Towards a unified view of bisimulation: a comparative study. TCS, 238:81–130, 2000.
13. J. A. Goguen, S. W. Thetcher, E. G. Wagner, and J. B. Write. Initial algebra semantics and continuous algebras. J.ACM, (24):68–95, 1977.
14. A. Letichevsky. Algebras with approximation and recursive data structures. Kibernetika and System Analysis, (5):32–37, September-October 1987.
15. R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, oct 1991.
16. P. Azcel, J. Adamek, S. Milius, and J. Velebil. Infinite trees and comletely iterative theories: a coalgebraic view. TCS, 300(1-3):1–45, 2003.
17. A. Letichevsky and D. Gilbert. Interaction of agents and environments. In D. Bert and C. Choppy, editors, Resent trends in Algebraic Development technique, LNCS 1827. Springer-Verlag, 1999.
18. A. Letichevsky, J. Kapitonova, V. Kotlyarov, A. Letichevsky Jr, and V. Volkov. Semantics of timed msc language. Kibernetika and System Analysis, (4), 2002.
19. J. Rutten. Coalgebras and systems. TCS, 249, 2000.
20. R. Milner. A Calculus of Communicating Systems, volume 92 of Lecture Notes in Computer Science. Springer-Verlag, 1980.
21. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science, 96:73–155, 1992.
22. C. A. R. Hoare. Communicating Sequential Processes. Prentice Hall, 1985.
23. V. M. Glushkov and A. A. Letichevsky. Theory of algorithms and descrete processors. In J. T. Tou, editor, Advances in Information Systems Science, volume 1, pages 1–58. Plenum Press, 1969.
24. V. M. Glushkov. Automata theory and formal transformations of microprograms.

Kibernetika, (5), 1965.
25. R. J. Glabbeek. The linear time — branching time spectrum i. the semantics of concrete, sequential processes. In J. A. Bergstra, A. Ponce, and S. A. Smolka, editors, Handbook of Process Algebra, pages 3–99. North-Holland, 2001.
26. P. Azcel, J. Adamek, S. Milius, and J. Velebil. Infinite trees and comletely iterative theories: a coalgebraic view. TCS, 300(1-3):1–45, 2003.
27. Dr. Alan Kay on the Meaning of "Object-Oriented Programming"". 2003. Retrieved 11 February 2010.
28. Ross, Doug. "The first software engineering language". *LCS/AI Lab Timeline*. MIT Computer Science and Artificial Intelligence Laboratory. Retrieved 13 May 2010.
29. Gamma, Erich; Richard Helm; Ralph Johnson; John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. Bibcode:1995dper.book.....G. ISBN 978-0-201-63361-0.
30. Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley. Bibcode:1992oose.book.....J. ISBN 978-0-201-54435-0.
31. McCarthy, John; Abrahams, Paul W.; Edwards, Daniel J.; Hart, swapnil d.; Levin, Michael I. (1962). *LISP 1.5 Programmer's Manual*. MIT Press. p. 105. ISBN 978-0-262-13011-0. Object — a synonym for atomic symbol
32. Cardelli, Luca; Wegner, Peter (10 December 1985). "On understanding types, data abstraction, and polymorphism". *ACM Computing Surveys*. **17** (4): 471–523. doi:10.1145/6041.6042. ISSN 0360-0300.
33. "Difference Between Symmetric and Asymmetric Multiprocessing (With Comparison Chart)". 22 September 2016. Archived from the original on 18 July 2021. Retrieved 18 July 2021
34. EDN Staff (1 January 2000). "General Instrument's microprocessor aimed at minicomputer market". *EDN*. Archived from the original on 25 November 2022.

Retrieved 1 January 2023.
35. 16 Bit Microprocessor Handbook by Gerry Kane, Adam Osborne ISBN 0-07-931043-5 (0-07-931043-5)
36. "The Birth of the Microprocessor". Archived from the original on 4 October 2022. Retrieved 4 October 2022.
37. Markoff, John (20 June 1996). "For Texas Instruments, Some Bragging Rights". *The New York Times*. Archived from the original on 28 September 2022. Retrieved 4 October 2022.
38. Liptak, B. G. (2006). *Process Control and Optimization*. Instrument Engineers' Handbook. Vol. 2 (4th ed.). CRC Press. pp. 11–12. ISBN 978-0849310812 – via Google Books.
39. "1971: Microprocessor Integrates CPU Function onto a Single Chip". *The Silicon Engine*. Computer History Museum. Archived from the original on 12 August 2021. Retrieved 22 July 2019.
40. Kapil Anand, Matthew Smithson, Aparna Kotha, Khaled Elwazeer, and Rajeev Barua. Decompilation to compiler high ir in a binary rewriter. University of Maryland, Tech. Rep, 2010.
41. Thierno Barry, Damien Courouss´e, and Bruno Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, pages 1–6, 2016.
42. Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. Provable secure software masking in the real-world. In International Workshop on Constructive Side-Channel Analysis and Secure Design, pages 215–235. Springer, 2022.
43. Leonid Azriel, Julian Speith, Nils Albartus, Ran Ginosar, Avi Mendelson, and Christof Paar. A survey of algorithmic methods in IC reverse engineering. J. Cryptogr. Eng., 11(3):299–315, 2021.
44. Alberto Battistello, Jean-S´ebastien Coron, Emmanuel Prouff, and Rina Zeitoun.

Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In International Conference on Cryptographic Hardware and Embedded Systems, pages 23–39. Springer, 2016.

45. Omid Bazangani, Alexandre Iooss, Ileana Buhan, and Lejla Batina. Abby: Automating the creation of fine-grained leakage models. Cryptology ePrint Archive, 2021.

46. Omid Bazangani, Alexandre Iooss, Ileana Buhan, and Lejla Batina. ABBY: automating the creation of fine-grained leakage models. IACR Cryptol. ePrint Arch., page 1569, 2021.

47. G. T. Becker et al. Test vector leakage assessment (tvla) methodology in practice. In International Cryptographic Module Conference, 2013.

48.Eli Biham. A fast new DES implementation in software. In International Workshop on Fast Software Encryption, pages 260–272. Springer, 1997.

49. Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina K¨onighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 321–353. Springer, 2018.

50. Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In International conference on the theory and applications of cryptographic techniques, pages 37–51. Springer, 1997.

51. Jean-Baptiste Br´ejon, Karine Heydemann, Emmanuelle Encrenaz, Quentin Meunier, and Son-Tuan Vu. Fault attack vulnerability assessment of binary code. In Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems, pages 13– 18, 2019.

52. Paul Caspi, Stavros Tripakis, and Pascal Raymond. Synchronous programming., 2007.

53. Y. Fei et al. A statistics-based fundamental model for side-channel attack

analysis. IACR Cryptol. ePrint Arch., 2014:152, 2014.
54. Research Institute for Secure Systems (RISCEC/AIST). Evaluation environment for side-channel attacks. URL:[“https://www.risec.aist.go.jp/project/sasebo/”]
55. Ilias Giechaskiel, Ken Eguro, and Kasper B. Rasmussen. Leakier wires: Exploiting fpga long wires for covert- and side-channel attacks. ACM Trans. Reconfigurable Technol. Syst., 12(3), aug 2019.
56. Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco:{Co-Design} and {Co-Verification} of masked software implementations on {CPUs}. In 30th USENIX Security Symposium (USENIX Security 21), pages 1469– 1468, 2021.
57. Barbara Gigerl, Robert Primas, and Stefan Mangard. Secure and efficient software masking on superscalar pipelined processors. In International Conference on the Theory and Application of Cryptology and Information Security, pages 3–32. Springer, 2021.
58. Daniel Gruss, Cl´ementine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 279–299. Springer, 2016.
59. Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The led block cipher. In International workshop on cryptographic hardware and embedded systems, pages 326–341. Springer, 2011.
60. M. He et al. Rtl-psc: Automated power side-channel leakage assessment at registertransfer level. In 2019 IEEE 37th VLSI Test Symposium (VTS), pages 1–6. IEEE, 2019.
61. Miao Tony He, Jungmin Park, Adib Nahiyan, Apostol Vassilev, Yier Jin, and Mark Mohammad Tehranipoor. RTL-PSC: automated power side-channel leakage assessment at register-transfer level. In VTS, pages 1–6, 2019.
62. Pantea Kiaei, Archanaa S Krishnan, and Patrick Schaumont. Parallel

synchronous code generation for second round light weight candidates. NIST Lightweight Cryptography Workshop, 2020.
63. David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I, volume 12491 of Lecture Notes in Computer Science, pages 787–816. Springer, 2020.
64. Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Annual International Cryptology Conference, pages 104–113. Springer, 1996.
65. Stefan Mangard and Kai Schramm. Pinpointing the side-channel leakage of masked AES hardware implementations. In Louis Goubin and Mitsuru Matsui, editors, Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings, volume 4249 of Lecture Notes in Computer Science, pages 76–90. Springer, 2006.
66. Madura A Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. arXiv preprint arXiv:1912.05183, 2019.
67. Madura A Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. Rosita++: Automatic higher-order leakage elimination from cryptographic code. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 685–699, 2021.
68. Nader Sehatbakhsh, Baki Berkay Yilmaz, Alenka G. Zajic, and Milos Prvulovic. EMSim: A microarchitecture-level simulation tool for modeling electromagnetic sidechannel signals. In HPCA, pages 71–85, 2020.
69. Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. Design and evaluation of smallfloat simd extensions to the risc-v isa. In 2019

Design, Automation Test in Europe Conference Exhibition (DATE), pages 654–657, 2019.

70. A. Vasselle, H. Thiebeauld, Q. Maouhoub, A. Morisset, and S. Ermeneux. Laserinduced fault injection on smartphone bypassing the secure boot-extended version. IEEE Transactions on Computers, 69(10):1449–1459, 2020.

71. Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In NDSS, 2017.

72. Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on cpus. In Michael Bailey and Rachel Greenstadt, editors, 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021, pages 1469–1468. USENIX Association, 20216

73. Barbara Gigerl, Robert Primas, and Stefan Mangard. Secure and efficient software masking on superscalar pipelined processors. In International Conference on the Theory and Application of Cryptology and Information Security, pages 3–32. Springer, 2021.

74. Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and K Asanovic. Replicating and mitigating spectre attacks on an open source risc-v microarchitecture. In Third Workshop on Computer Architecture Research with RISC-V (CARRV 2019), 2019.

75. Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina K¨onighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 321–353. Springer, 2018.

76. David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering:'grey box'modelling for instruction

leakages. In 26th USENIX Security Symposium (USENIX Security 17), pages 199–216, 2017.