Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікацій
(повне найменування факультету)

Кафедра інформаційної безпеки та наноелектроніки
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
магістр

(ступінь вищої освіти)

на тему Дослідження та моделювання методів віртуалізації квантових ресурсів для підвищення безпеки інформаційних систем

Виконав(ла): студент(ка) 2 курсу,
групи БК-814М
Спеціальності 125 Кібербезпека та захист інформації
(код і найменування спеціальності)
Освітня програма (спеціалізація)
Безпека інформаційних і комунікаційних систем
КРИВОШЕЄВ Д.М.
(ПРІЗВИЩЕ та ініціали)

Керівник НЕЛАСА Г.В.
(ПРІЗВИЩЕ та ініціали)
Рецензент САМОЙЛИК С.С.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Факультет інформаційної безпеки та електронних комунікації

Кафедра інформаційної безпеки та наноелектроніки

Ступінь вищої освіти: магістр

Спеціальність 125 Кібербезпека та захист інформації

Освітня програма (спеціалізація) Безпека інформаційних і комунікаційних систем

ЗАТВЕРДЖУЮ

Завідувач кафедри ІБтаН, к.ф-м.н.

Андрій КОРОТУН

“____” _____2025 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

КРИВОШЕЄВА Дениса Миколайовича

(ПРІЗВИЩЕ**,** ім’я**,** по батькові**)**

1. Тема проєкту (роботи): Дослідження та моделювання методів віртуалізації квантових ресурсів для підвищення безпеки інформаційних систем

керівник проєкту (роботи) канд. техн. наук, доцент НЕЛАСА Ганна Вікторівна

(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім’я, по батькові)

затверджені наказом закладу вищої освіти від « 26 » листопада 2025 року № 530

2. Строк подання студентом проєкту (роботи): 15.12.2025

3. Вихідні дані до проєкту (роботи): технології та архітектура сучасних практик розробки гіпервізорів для потреб інформаційної безпеки

4. ЗМІСТ розрахунково-пояснювальної записки (перелік питань, що їх потрібно розробити): Теоретичні відомості про віртуалізацію; Проєктування архітектури гіпервізору; Моделювання віртуалізації у квантових системах.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень, кількість слайдів, плакатів): Презентація у програмі Microsoft Power Point (14 слайдів)

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	Прийняв виконане завдання
1-3	НЕЛАСА Г.В., доцент каф. ІБтаН	05.09.2025	12.12.2025
нормконтроль	КОРОЛЬКОВ Р.Ю., доцент каф. ІБтаН		14.12.2025

7. Дата видачі завдання « 05 » вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Ознайомлення з темою та визначення ключових аспектів дипломного проєкту	1 тиждень	Виконано
2	Аналіз загальної відомостей щодо інформації про віртуалізацію.	1 тиждень	Виконано
3	Формулювання та написання вступу.	2 тиждень	Виконано
4	Проєктування моделі квантового гіпервізору.	3-5 тиждень	Виконано
5	Аналіз отриманих результатів та формування висновків дипломного проєкту.	5-6 тиждень	Виконано
6	Здача на перевірку та підпис кваліфікаційної роботи керівнику.	6 тиждень	Виконано
7	Проходження перевірки на плагіат та нормоконтроль кваліфікаційної роботи.	6 тиждень	Виконано
8	Допуск завідувачем кафедри до захисту кваліфікаційної роботи.	8 тиждень	
9	Захист кваліфікаційної роботи.	9 тиждень	

Студент(ка) _____ Денис КРИВОШЕЄВ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Ганна НЕЛАСА
(підпис) (Ім'я ПРІЗВИЩЕ)

АНОТАЦІЯ

Пояснювальна записка до магістерської роботи: 104 с., 5 табл., 32 рис., 1 дод., 45 джерел.

ВІРТУАЛІЗАЦІЯ, КВАНТОВА СИСТЕМА, ГІПЕРВІЗОР, КУБІТ, ВІРТУАЛЬНЕ СЕРЕДОВИЩЕ, ІЗОЛЯЦІЯ, ОПЕРАЦІЙНА СИСТЕМА, КЛАСИЧНА СИСТЕМА.

Об’єкт дослідження – процеси обробки, передавання та зберігання інформації у квантових обчислювальних системах.

Предмет дослідження – методи віртуалізації квантових ресурсів у квантових обчислювальних системах із забезпеченням ізоляції, контрольованого доступу та підвищення безпеки інформаційних процесів.

Мета роботи – дослідження та моделювання механізмів віртуалізації квантових ресурсів для підвищення рівня безпеки інформаційних систем на основі гіпервізорного підходу.

Наукова новизна - формування та обґрунтування моделі квантового гіпервізору, що демонструє принципи віртуалізації у квантових системах з урахуванням фізичних обмежень квантових середовищ.

Дипломний проєкт складається з трьох розділів. У першому розглядаються теоретичні відомості про віртуалізацію та основні проблеми у реалізаціях квантових гіпервізорів, в другому планування модельної реалізації та її особливості, в третьому розділі та опис програмної реалізації моделі.

ABSTRACT

Explanatory note to the master's thesis: 104 pages, 5 tables, 32 figures, 1 appendix, 45 sources.

VIRTUALISATION, QUANTUM SYSTEM, HYPERVISOR, QUBIT, VIRTUAL ENVIRONMENT, ISOLATION, OPERATING SYSTEM, CLASSICAL SYSTEM.

The object of research is the processes of processing, transmitting and storing information in quantum computing systems.

The subject of research is methods of virtualising quantum resources in quantum computing systems with isolation, controlled access and increased security of information processes.

The aim of the work is to study and model mechanisms for virtualising quantum resources to improve the security of information systems based on a hypervisor approach.

Scientific novelty is the formation and justification of a quantum hypervisor model that demonstrates the principles of virtualisation in quantum systems, taking into account the physical limitations of quantum environments.

The thesis project consists of three sections. The first section discusses theoretical information about virtualisation and the main problems in the implementation of quantum hypervisors, the second section discusses the planning of the model implementation and its features, and the third section describes the software implementation of the model.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ОС - Операційна система;

ПЗ - Програмне забезпечення;

MAC - Mandatory access control(Мандатне керування доступом);

UML - Unified Modeling Language(Уніфікована мова моделювання);

API - Application Programming Interface(Прикладний програмний інтерфейс);

VMI - Virtual Machine Introspection (Моніторинг гостьових систем);

SoC - Security Operations Center (Операційний центр безпеки);

VMCS - Virtual Machine Control Structure (Структура керування віртуальною машиною);

VMCB - Virtual Machine Control Block (Блок керування віртуальною машиною);

KVM - Kernel-based Virtual Machine (Віртуальна машина на основі ядра Linux);

ARM - Advanced RISC Machines (Розширені машини зменшеного набору команд).

ВСТУП

Стрімкий розвиток квантових технологій упродовж останнього десятиліття відкрив можливість створення обчислювальних платформ нового покоління. Квантові процесори, які раніше розглядалися як експериментальні лабораторні системи, сьогодні переходять до стадії практичного застосування у моделюванні, оптимізації, криптографії та аналізі складних систем. Разом з тим виникає принципово нова проблема - забезпечення можливості багатокористувацького, багатозадачного та ізольованого використання квантових апаратних ресурсів.

У класичних обчисленнях подібні завдання успішно вирішуються завдяки віртуалізації, однак у квантових системах пряме перенесення цих принципів неможливе через фундаментальні фізичні обмеження, такі як заборона копіювання квантового стану, чутливість кубітів до декогеренції та складна топологія зв’язків між ними.

Сучасні підходи до квантових обчислень, що базуються на хмарних платформах, надають лише одноразовий доступ до квантового пристрою без гарантії ізоляції, відтворюваності стану або можливості логічного розподілу ресурсів між незалежними задачами. Це створює потребу у нових архітектурах, здатних виконувати роль квантових гіпервізорів - систем, які забезпечують керування квантовими ресурсами на рівні, аналогічному гіпервізорам традиційних операційних систем.

У даній роботі досліджуються принципи побудови такої системи та пропонується модель квантового гіпервізору, що демонструє можливість створення віртуальних квантових середовищ, їх ізоляції, маршрутизації квантової заплутаності, управління політиками доступу та логічної міграції без порушення базових законів квантової механіки. Для цього було здійснено аналіз фундаментальних концепцій

квантових обчислень, топології фізичних кубітів, обмежень на виконання унітарних операцій та існуючих архітектур хмарних квантових сервісів.

Запропонована симуляційна модель квантового гіпервізору відображає основні принципи, за якими може працювати майбутня квантова платформа загального призначення, і демонструє можливі шляхи подолання поточних обмежень квантових систем через програмні рішення та логічні механізми. Робота має не лише навчальне значення, але й формує підґрунтя для подальших досліджень у галузі квантової віртуалізації та квантово-класичних гібридних систем.

1 ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО ВІРТУАЛІЗАЦІЮ

1.1 Абстрактний рівень віртуалізації у класичних системах

Більшість звичайних комп'ютерних систем мають можливості для віртуалізації. Віртуалізація - це спосіб організувати декілька незалежних систем в одному апаратному середовищі. Це може бути як системний рівень (апаратно повноцінна віртуалізація), так і програмний рівень (паравіртуалізація). У такому випадку віртуалізована система знає, що працює у віртуальному середовищі, і взаємодіє з апаратним забезпеченням через системне API. Проте в деяких випадках система емулюється повністю програмно, і це інколи називають контейнеризацією (віртуалізацією на рівні операційної системи) [1].

На системному рівні процес віртуалізації оркеструється так званим гіпервізором [2]. Він може бути як вбудованим апаратним модулем, так і дискретним апаратним рішенням. За допомогою цих пристроїв здійснюється розподіл системних ресурсів на ізольовані частини. До апаратних модулів можна віднести, наприклад, vTPM, які переважно працюють як мікросистеми. Це забезпечує швидшу обробку даних, але не дає широкого набору функцій. До програмних рішень можна віднести, зокрема, Oracle VirtualBox, який забезпечує можливість запуску повноцінних операційних систем, хоча й накладає певні обмеження. Зокрема, у гостьовому середовищі недоступні такі функції, як прямий доступ до дискретної графічної карти або доступ до BIOS.

Якщо описувати типи гіпервізорів, то їх поділяють на гіпервізори першого типу - ті, що працюють безпосередньо на апаратній основі, та гіпервізори другого типу - ті, що працюють поверх звичайної операційної системи [3]. Оскільки програмна віртуалізація (у випадку повної програмної емуляції) має значно нижчу продуктивність через відсутність прямого доступу до фізичних ресурсів, нині її прийнято поєднувати з апаратними модулями. Вони поширені на більшості систем

x86-64. Наприклад, у процесорів Intel це технологія Intel VT-x, а у процесорів AMD - AMD-V [4].

Архітектура ARM у цьому сенсі є відносно молодою, адже її масове поширення на ринку настільних систем почалося нещодавно. Хоча ARM також реалізує механізми віртуалізації, частина рішень усе ще є програмною, оскільки більшість ARM-пристроїв спочатку були орієнтовані на енергоефективні завдання - мобільні пристрої та IoT-системи. Проте зараз з'являються й апаратні засоби віртуалізації для широкого кола користувачів. Наприклад, у Qualcomm це Gunyah Type 1, а в рішень Apple - The Virtualization Framework. Завдяки орієнтації на енергоефективність ARM-системи часто використовують більш легкі механізми віртуалізації, такі як паравіртуалізація або контейнеризація. Однак існують і рішення для повної віртуалізації - такі як KVM for ARM або XEN/HYPER-V ARM Edition, що дозволяють віртуалізувати навіть x86-64-образи систем (Linux, BSD, Windows).

Робота гіпервізора полягає у наступному. Фізичні кубіти поділяють на підгрупи, розбиваючи загальний фізичний простір на кілька qVM. Кожна qVM отримує виділену підмножину кубітів, яка максимально ізолюється шляхом формування буферної зони навколо кожної підсистеми. Це зменшує перехресні впливи та шум між логічно розділеними областями. У деяких випадках такий логічний простір можна розглядати як аналог оперативної пам'яті або процесорних ресурсів.

Таке розділення або ізоляція систем дозволяє не лише зручніше використовувати середовище, у якому працює необхідне програмне забезпечення, а й захищати інші системи від можливих критичних збоїв або, у випадку кібербезпеки, - від компрометації даних, якщо в одній із систем з'явиться шкідливе ПЗ. Це допомагає зберегти дані, проте залежно від характеру взаємодії між системами можуть існувати різні ризики.

Якщо узагальнити основні завдання гіпервізора, можна виділити такі [5]:

- ізоляція середовища - незалежність кожної віртуальної системи від інших;

- розподіл ресурсів системи на необхідні або жорстко виділені частини: кількість ядер процесора, обсяг оперативної пам’яті, обсяг фізичної пам’яті на жорсткому диску тощо;

- підтримка гарячої заміни систем (швидкого відновлення) або міграції з однієї системи на іншу з метою забезпечення безперервності бізнес-процесів.

У середині кожна віртуальна система повинна виконувати інструкції системи або перехоплювати їх за допомогою механізму trap-and-emulate. Вона також має містити власні апаратно-програмні драйвери для взаємодії з апаратними ресурсами, необхідні бібліотеки, плагіни тощо. Іншим важливим фактором є те, що система має розуміти, що вона ізольована, а її ресурси чітко зарезервовані (віртуальні процесорні ядра, віртуальна оперативна пам’ять тощо), і що вона не має доступу до всього фізичного обладнання хост-системи.

У деяких випадках віртуальні системи дозволяють динамічно змінювати обсяг необхідних апаратних ресурсів і повідомляти про це гіпервізор. Наприклад, якщо встановлено, що віртуальна система не може використовувати більше ніж 8 ГБ оперативної пам’яті, то в робочому стані вона може споживати лише 5 ГБ, залишаючи хост-системі достатньо ресурсів. Так само можна віртуально виділити, наприклад, 150 ГБ фізичної пам’яті з динамічним розширенням дискового простору. Проте в такому випадку необхідно стежити, щоб система не перевищила допустимий обсяг пам’яті, адже це може негативно вплинути на стабільність роботи віртуальної машини.

Проте існують певні відмінності між архітектурами. Наприклад, у системах x86-64 кільця доступу (привілеїв) мають такий вигляд [6]:

- Ring 0 - ядро системи;

- Ring 1, 2 - системні рівні, які практично не використовуються в сучасних рішеннях;

- Ring 3 - процеси користувачів;

- Root mode - рівень гіпервізора.

Root mode, у свою чергу, працює у режимах VMX/SVM root mode (для хост-системи) та VMX/SVM non-root mode (для гостьової системи). Це забезпечує те, що інструкції, потрібні для роботи віртуалізації, не впливають на роботу основної системи. Системами керування для цих режимів є структури VMCS для Intel та VMCB для AMD [7].

В архітектурі ARM ієрархія кілець безпеки або рівнів має іншу структуру [8]:

- EL0 - рівень користувача (аналог Ring 3);

- EL1 - рівень ядра системи;

- EL2 - рівень гіпервізора;

- EL3 - рівень безпеки.

Рівень EL3 поділяє систему на дві так звані зони за допомогою механізму TrustZone:

Secure World - ізольоване середовище для довіреного коду, наприклад криптографічних методів обробки або біометрично чутливих даних;

Normal World - середовище звичайної системної роботи та застосунків.

Архітектура ARM загалом забезпечує більш структуровану ієрархію, що створює безпечнішу схему взаємодії між системними ресурсами, а також більш ізольовану модель віртуалізації, оскільки віртуалізація та безпека фізично розділені між собою.

На відміну від структур VMCS та VMCB, ARM використовує Virtualization Extensions, які утворюють такі компоненти:

- HYP mode для EL2[8];

- Stage-2 translation - рівень трансляції адрес (аналог Extended Page Tables у Intel та Nested Page Tables у AMD);

- VGIC - механізм керування перериваннями між гіпервізором та гостьовими системами [9];

- Trap mechanisms - механізми перехоплення інструкцій і подій для контролю гіпервізора[8].

Також варто зазначити ізоляцію та віртуалізацію пристроїв введення-виведення за допомогою SMMU, аналогом якого в x86-64 є IOMMU. Це критично важлива функція з погляду безпеки, оскільки дозволяє таким пристроям, як «радіомодуль» або «application processor», працювати ізольовано.

Важливою відмінністю між архітектурами є також керування пам’яттю. У системах x86-64 воно організоване наступним чином - гостьове середовище (guest virtual) працює зі своїм віртуальним адресним простором, який відображається у guest physical. Далі гіпервізор транслює guest physical у host physical за допомогою механізмів EPT або NPT.

ARM-системи забезпечують проходження пам’яті шляхом двох стадій трансляції. На Stage-1 керування здійснює гостьове середовище, перетворюючи віртуальні адреси процесів у проміжні «віртуально-фізичні» адреси. На Stage-2 гіпервізор транслює ці адреси вже у реальні фізичні адреси. Усі ці операції в ARM супроводжуються оптимізацією енергоспоживання та кешування, а TLB організовано так, щоб мінімізувати кількість звернень і контекстних перемикань у системі.

Віртуалізація та правильне мультиплексування ресурсів у сучасних датацентрах і хмарних середовищах є не менш важливими, ніж саме апаратне забезпечення. Завдяки можливості швидкого запуску потрібних середовищ за шаблоном на одній фізичній системі відпадає потреба в постійному ручному налаштуванні обладнання.

З бізнесово-технічної точки зору такий підхід дає змогу обслуговувати більше клієнтів, оскільки кілька клієнтів можуть працювати на одній системі без потреби у виділенні надлишкових ресурсів, які часто простоювали б за підходу «один клієнт - одна система». При віртуалізації модель перетворюється на «один клієнт - одне середовище», помножене на кількість клієнтів і пропорційне доступним ресурсам.

Не слід забувати й про те, що окрім системної віртуалізації існують й інші її типи:

- віртуалізація фізичної пам’яті (storage virtualization) - коли пам’ять віртуалізують для спільного використання між віртуальними середовищами;

- мережева віртуалізація (network virtualization) - віртуалізація мережевої інфраструктури для взаємодії між середовищами, наприклад VXLAN;

- віртуалізація віддалених робочих столів (VDI) - запуск середовищ через віддалений клієнт із можливістю дистанційного керування, що полегшує адміністрування та підвищує рівень безпеки.

У випадку мережевої віртуалізації також варто враховувати можливість атак типу «втеча з віртуального середовища» або контейнера (VM escape). Такі атаки можуть бути реалізовані через вразливості в програмному коді гіпервізора або через мережеву інфраструктуру. У подібному сценарії зловмисник може навіть отримати доступ до хост-системи, якщо вона використовується як оркестрантом середовищ.

Однак подібні атаки є дуже складними в реалізації, тому загалом системи віртуалізації вважаються одним із найнадійніших методів захисту інформації в апаратних середовищах. У наш час важко уявити повсякденну роботу бізнес-систем без можливостей віртуалізації, адже її зазвичай відносять до категорії «необхідних» засобів - на одному рівні з системами резервного копіювання та антивірусним захистом.

1.2 Низький рівень абстракції віртуалізації

Важливими початковими елементами на програмному рівні у Linux є технології VMX для Intel [10] та AMD-V або SVM для AMD [11], які є апаратними розширеннями процесора. Їхня основна задача - забезпечити створення віртуальних машин без необхідності емулювати небезпечні або привілейовані інструкції.

Далі йдуть монітори віртуальних машин (VMM), найближчим прикладом яких є KVM. Основні завдання VMM полягають у:

- створенні та ініціалізації структур VMCS або VMCB;

- запуску гостьового коду через VMLAUNCH або VMRESUME;

- обробленні VMEXIT у випадку виконання забороненої інструкції в non-root режимі;

- оновленні регістрів, пам’яті або таблиць EPT/NPT;

- поверненні керування гостьовій системі.

До появи VMX або SVM віртуалізація здійснювалася за допомогою механізму trap-and-emulate, коли гіпервізор перехоплював привілейовані інструкції й емулював їхню логіку програмно, що приводило до суттєвого падіння продуктивності.

Проте апаратні рішення VMX та SVM вирішили цю проблему, запровадивши два окремі рівні виконання - root (гіпервізор/VMM) і non-root (гість), що значно пришвидшило та спростило роботу систем віртуалізації.

Таблиця 1.1 - Основні метод-алгоритми для VMX та SMV

Метод або інструкція	Рівень	Завдання
1	2	3
VMXON/VMXOFF	ASM(INTEL)	Увімкнення або вимкнення VMX операції на логічному процесорі
VMLAUNCH/ VMRESUME	ASM(INTEL)	Виконання VM entry для запуску гостьового контексту
VMREAD/VMWRITE	ASM(INTEL)	Читання або запис інформації

Кінець таблиці 1.1

1	2	3
VMRUN/VMLOAD/ VMSAVE/VMMCALL	ASM(AMD)	Аналогічні операції для входу або виходу чи збереження стану
EPT	MMU/HW	Трансляція фізичного гостя у фізичний хост без тіньової сторінки таблиць
NPT/RVI(AMD)	MMU/HW	Аналог EPT
VM exit handling loop	Hypervisor(KVM/QEMU)	Перехоплює виходи, емуляцію, та оновлення VMCS/VMCB та повертання у гостя
KVM_CREATE_VM, KVM_CREATE_VCPU	Kernel API	Створення віртуальної машини та віртуального процесору у організаторі середовища через ioctls

VMCS від Intel у цьому контексті - це інформаційна структура, яка зберігає стан віртуальної машини. Вона містить:

- Guest-State Area - поля стану гостьової системи, де зберігаються регістри, які буде завантажено під час переходу до гостя;

- Host-State Area - поля стану хост-системи, що містять регістри, які процесор відновить після виходу з гостьового режиму (VM-exit);

- Control Fields - поля керування, що визначають події, які спричиняють VM-exit, а також спосіб обробки механізмів EPT, MSR та інших параметрів.

GUEST STATE AREA				
CR0	CR3		CR4	
DR7				
RSP	RIP		RFLAGS	
CS	Selector	Base Address	Segment Limit	Access Right
SS	Selector	Base Address	Segment Limit	Access Right
DS	Selector	Base Address	Segment Limit	Access Right
ES	Selector	Base Address	Segment Limit	Access Right
FS	Selector	Base Address	Segment Limit	Access Right
GS	Selector	Base Address	Segment Limit	Access Right
LDTR	Selector	Base Address	Segment Limit	Access Right
TR	Selector	Base Address	Segment Limit	Access Right
GDTR	Selector	Base Address	Segment Limit	Access Right
IDTR	Selector	Base Address	Segment Limit	Access Right
IA32_DEBUGCTL	IA32_SYSENTER_CS	IA32_SYSENTER_ESP	IA32_SYSENTER_EIP	
IA32_PERF_GLOBAL_CTRL	IA32_PAT	IA32_EFER	IA32_BNDCFGS	
SMBASE				
Activity state	Interruptibility state			
Pending debug exceptions				
VMCS link pointer				
VMX-preemption timer value				
Page-directory-pointer-table entries	PDPTE0	PDPTE1	PDPTE2	PDPTE3
Guest interrupt status				
PML index				

HOST STATE AREA			
CR0	CR3	CR4	
RSP	RIP		
CS	Selector		
SS	Selector		
DS	Selector		
ES	Selector		
FS	Selector	Base Address	
GS	Selector	Base Address	
TR	Selector	Base Address	
GDTR	Base Address		
IDTR	Base Address		
IA32_SYSENTER_CS	IA32_SYSENTER_ESP	IA32_SYSENTER_EIP	
IA32_PERF_GLOBAL_CTRL	IA32_PAT	IA32_EFER	

Рисунок 1.1 - Структури Guest-State Area та Host-State Area [12]

Покроково та на низькому рівні запуск гостьової системи на процесорах Intel відбувається так. Спершу перевіряється наявність підтримки VT-x і встановлюється біт CR4.VMXE=1. Далі готується VMXON-region, у який записується VMCS revision ID, після чого виконується інструкція VMXON. Потім створюється VMCS-region для vCPU, заповнюються та через VMWRITE записуються поля на кшталт RIP, RSP, CR0/CR3/CR4 й інші необхідні регістри. Якщо застосовується EPT, у VMCS вказується відповідний EPT pointer і налаштовуються таблиці EPT. Після цього процесор запускає гостьову систему за допомогою VMLAUNCH або VMRESUME та переходить у режим VM non-root. Коли відбувається VM-exit, процесор повертається до режиму VM root, гіпервізор зчитує з VMCS причину виходу, обробляє її і знову передає керування гостю через VMRESUME.

Аналогом VMCS у AMD є VMCB, який описує стан гостьового процесу і складається з двох частин:

- Control Area - керує перериваннями, налаштуваннями NPT, регістрами MSR, таймером TSC тощо;

- State Save Area - містить регістри RIP, RSP, RFLAGS, а також сегменти та регістри CR0/CR3/CR4.

```
1   //Linux Struct
2   struct __attribute__ ((__packed__)) vmcb_fmt_t
3   {
4       control_area_64_t control_area;
5       save_state_64_t   save_state;
6   };
7
8   // Windows Struct
9   struct vmcb_fmt_t
10  {
11      control_area_64_t control_area;
12      save_state_64_t   save_state;
13      uint8_t reserved[RESERVED_SIZE];
14  };
15
```

Рисунок 1.2 - Приклад програмного виклику VMCB блоків у системах Windows та Linux [13]

У випадку AMD алгоритм запуску гостьової системи виглядає приблизно так:

- перевіряється підтримка SVM та біти у CPUID, наприклад EAX = 8000_0001h, ECX bit 2;

- увімкнення SVM;

- підготовка VMCB та заповнення відповідних регістрів;

- ініціалізація середовища гостя;

- перехід процесора у режим гостя;

- виклик VM-exit у випадку виконання забороненої інструкції або переривання;

- процесор заповнює поля VMCB.EXITCODE, EXITINFO1, EXITINFO2 та повертає керування гіпервізору;

Рухаючись вгору на системному рівні Linux від VMX або SVM, формується алгоритм високого рівня.

Для Intel він виглядає наступним чином [14]:

- KVM виконує ioctl(KVM_CREATE_VM), створюючи структуру kvm;

- для кожного vCPU виконується ioctl(KVM_CREATE_VCPU);

- KVM заповнює поля VMCS, зокрема Guest RIP, BIOS Bootloader, Host RIP;

- виклик внутрішніх функцій ядра через vmx_cpu_run();

- vmx_cpu_run() виконує інструкцію VMLAUNCH;

- під час VM-exit відбувається автоматичний розподіл ресурсів і перехід у root mode.

Для AMD алгоритм подібний, але з невеликими відмінностями. Спочатку створюється область пам’яті VMCB, далі заповнюються поля Guest RIP, Guest RSP і маска перехоплення подій (Control intercept), які визначають, що саме спричиняє VM-exit. Під час виконання svm_vcpu_run використовується інструкція VMRUN, яка передає керування гостьовій системі.

В ARM-архітектурі основними регістрами для раніше згаданих рівнів привілеїв EL є:

- HCR_EL2 - керує гіпервізором, задаючи обмеження для гостя та визначаючи режими, які він може використовувати;

- VTTBR_EL2 - вказує на таблицю пам’яті для гостя на рівні Stage-2;

- VTCR_EL2 - регістр контролю трансляції Stage-2.

Також варто відзначити інструкцію ERET, яка є аналогом VMLAUNCH та VMRESUME. Через це в ARM немає окремого аналога VMCS - всі необхідні дані зберігаються безпосередньо в системних регістрах EL2.

На низькому рівні алгоритм роботи виглядає так. Процесор запускає гіпервізор у EL2 (root mode), після чого гіпервізор встановлює HCR_EL2, виставляючи біт

VM=1 для активації віртуалізації, а також задає інструкції та події, які мають оброблятися в EL2. Далі вказуються таблиці Stage-2 у VTTBR_EL2, що визначають віртуальну пам’ять гостя. Потім записуються гостьові регістри, зокрема ELR_EL2, SPSR_EL2, SP_EL1 та TTBR0_EL1. Інструкція ERET переносить виконання з EL2 у EL1, фактично запускаючи гостьову систему в режимі VM.

Коли відбувається VM-exit, генерується trap у EL2 у відповідь на заборонену або перехоплену інструкцію гостя. Гіпервізор зчитує причину виходу через ESR_EL2 та FAR_EL2, після чого повернення до гостя знову виконується інструкцією ERET, яка повертає процесор у EL1 і продовжує виконання гостьового коду.

На високому рівні виконуються ті самі функції, що й у прикладах Intel та AMD:

- гостьовий стан керується через регістри, створюється аналог EPT та керується вихід/вхід у EL2;

- QEMU викликає ioctl(KVM_RUN);

- ядро опрацьовує функцію kvm_arch_vcpu_ioctl_run() та викликає розширення віртуалізації хосту kvm_vcpu_run_vhe();

- KVM завантажує гостьовий стан у EL1-регістри через EL2, встановлює HCR_EL2, VTTBR_EL2 та виконує ERET для переходу у EL1;

- Якщо стався VM-exit, процесор генерує синхронний виняток у EL2, обробляє ESR_EL2 та повертає керування користувачу (у даному випадку - QEMU).

Слід також згадати про поняття модульної віртуалізації за допомогою графічного прискорювача як одного зі способів забезпечення високої продуктивності графічних обчислень замість використання звичайного процесора. Існують три основні підходи до цього рішення. Перший - пряме присвоєння (GPU passthrough), коли графічний прискорювач передається безпосередньо в гостьову систему. Другий - використання через віртуалізовані GPU (vGPU), що дає змогу кільком гостьовим системам спільно використовувати один фізичний GPU. Третій - віртуалізація на

рівні драйвера (API remoting), яка передбачає перехоплення графічних викликів і їхнє віддалене виконання на хості через відповідні драйвери.

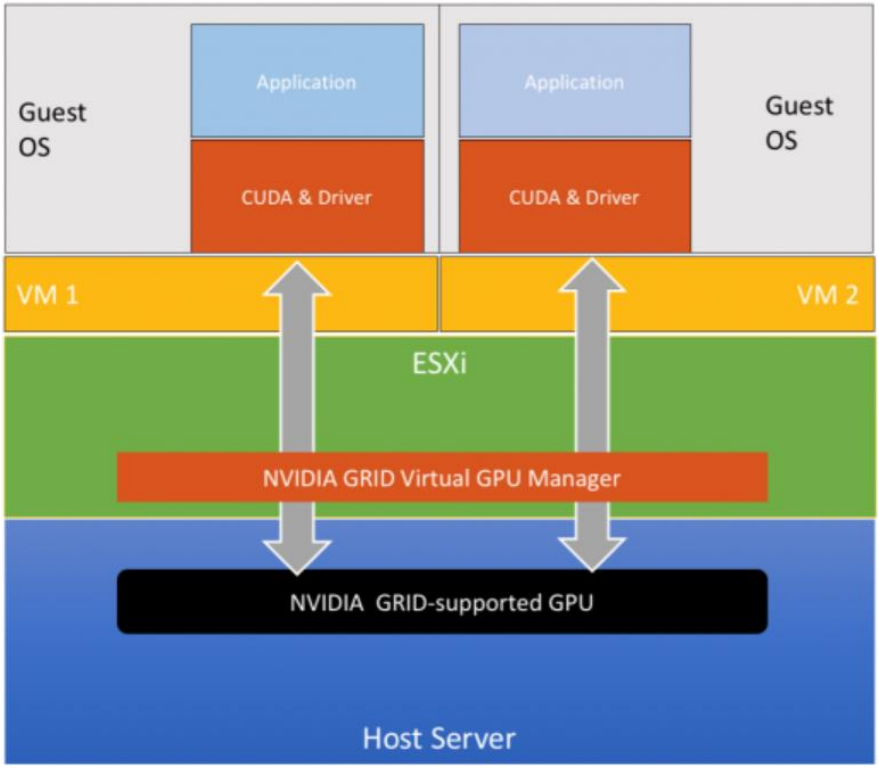

Рисунок 1.3 - Схема роботи віртуалізаціїї графічного прискорювача у vSphere [15]

Пряме присвоєння (GPU passthrough) [16] використовує фізичний графічний прискорювач або його значну частину для прикріплення до певної віртуальної машини, завдяки чому вона отримує майже повноцінну продуктивність. Водночас цей підхід забезпечує обмежену гнучкість у поділі ресурсів. Основною перевагою методу є простота - віртуальна машина отримує всю або майже всю потужність графічного прискорювача, що робить його хорошим вибором для задач із максимальною вимогою до продуктивності. Недоліки цього підходу включають принцип «одна віртуальна машина = один графічний прискорювач» та складнощі з міграцією чи забезпеченням відмовостійкості.

Рішення з поділом (vGPU) дозволяє декільком віртуальним машинам спільно використовувати один графічний прискорювач шляхом часової мультиплексії або виділення певних частин ресурсів графічного прискорювача. До таких ресурсів належать обсяг відеопам’яті, кількість обчислювальних блоків і гарантія якості обслуговування (QoS).

Віртуалізація на рівні драйвера (API remoting) передбачає більш поверхневий підхід, за якого віртуальна машина або контейнер надсилає запити на формування кадрів графічного прискорювача локально або через мережу, і прискорювач обробляє ці запити від імені віртуальної машини.

Для більш глибокого розуміння технічних аспектів виділяють кілька ключових компонентів. Перш за все це юніт управління пам’яттю вводу-виводу (IOMMU), який відповідає за трансляцію адрес і ізоляцію пристроїв у разі прямого доступу. IOMMU може ініціалізувати прямий доступ до пам’яті через DMA, проте графічний прискорювач з DMA здатен звертатися до пам’яті будь-якої віртуальної машини. Щоб запобігти цьому, IOMMU створює таблицю трансляцій DMA для кожного PCI-пристрою або віртуально-функціонального інтерфейсу (VF), у якій вказуються фізичні сторінки, доступні процесу на графічній карті.

Іншим підходом є передача PCI через одиночний root для віртуалізації вводу-виводу (SR-IOV), коли пристрій PCI створює кілька «вирізів», які можуть бути прикріплені до різних віртуальних машин. У цьому випадку графічний прискорювач має фізичну функцію (PF), яка є головною кінцевою точкою управління пристроєм, та кілька VF-інтерфейсів. Кожен VF виглядає як окремий графічний прискорювач для гостьових ОС і отримує унікальний PCI BDF (Bus Device Function), власний набір MMIO-регістрів, DMA-дескрипторів та контекстів пам’яті. Графічний прискорювач зазвичай має драйвери та контексти виконання, такі як командні черги, контексти пам’яті та запуск у ядрі. Тому під час віртуалізації графічного прискорювача необхідно, щоб гіпервізор або драйвери могли надати власний ізольований контекст, особливо у випадку підходу з поділом ресурсів. Наступним

кроком гіпервізор передає кожен VF-інтерфейс безпосередньо у віртуальну машину через device passthrough, внаслідок чого віртуальна машина бачить прикріплений графічний прискорювач як власний. На рівні модулів графічного прискорювача це реалізовано через AMD MxGPU для прискорювачів AMD та NVIDIA GRID vGPU для рішень NVIDIA.

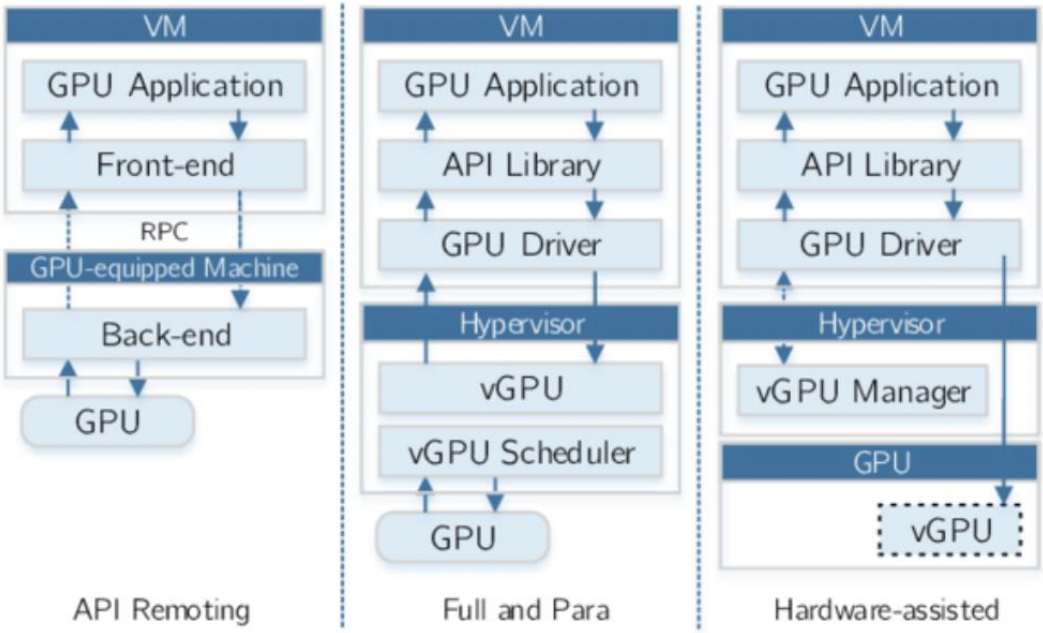

Рисунок 1.4 - Приклад реалізації для кожного рішення [17]

Прийнято вважати, що кожен графічний прискорювач має багаторівневу ієрархію пам’яті, яка складається з глобальної VRAM, локальної per-SM, shared-пам’яті та регістрової пам’яті. Коли для кожного гостя створюється окрема таблиця сторінок графічного прискорювача, саме MMU графічного прискорювача зберігає її у собі. У випадку, якщо інструкція або процес виходить за межі цього простору, викликається GPU fault, який передається у хост-драйвер, а далі - у гіпервізор. Ізоляція у такому підході відбувається шляхом перемикання контекстів у моменти поділу графічного прискорювача через time-slicing.

Графічний прискорювач збирає поточний набір регістрів, буферів команд, кешів текстур та інших ресурсів, а потім завантажує новий контекст іншої віртуальної машини. У випадку AMD чи NVIDIA це реалізується через механізми context preemption. Коли гіпервізор сигналізує графічному прискорювачу зробити перемикання контекстів, прискорювач завершує поточні wavefront або warp, зберігає стан у VRAM, проводить очищення кешів і виконує context_load() для нового контексту. Завдяки цим діям створюється жорстка ізоляція між контекстами, що запобігає залишкам даних у кешах, які могли б бути використані бічними каналами.

На рівні операційної системи керування ізоляцією через драйвери та гіпервізор зазвичай реалізується за допомогою KVM VFIO [18] у Linux або HYPER-V та паравіртуалізатора GPU-PV у Windows. У Linux драйвер VFIO PCI дозволяє безпечно передавати VF GPU у гостьову віртуальну машину. Він створює контейнер, у якому зібрані всі карти IOMMU для цієї віртуальної машини. Під час виконання VFIO виконується виклик ioctl (VFIO_SET_IOMMU), де створюється таблиця трансляції DMA для VF графічного прискорювача. Наступним кроком ioctl (VFIO_DEVICE_GET_REGION_INFO) передає віртуальній машині регістри MMIO GPU VF, які будуть відображені у її адресний простір.

Проводячи аналогії з процесорною віртуалізацією, можна сформувати таблицю паралелей або аналогів між компонентами GPU та CPU.

Таблиця 1.2 - Таблиця аналогового представлення викликів метод-алгоритмів у порівнянні з класичними методами

Рішення у звичайному процесорі	Аналог у графічному прискорювачі
1	2
VMCB або VMCS	GPU context descriptor
VMRUN або VMENTER	context load
VM-exit	GPU fault або context switch interrupt
Intercepts	DMA traps або command preemption points

Кінець таблиці 1.2

1	2
MSR з CR	MMIO-регістр
EAX або EDX регістри	shader або compute регістри

Якщо користувач хоче реалізувати пряму передачу графічного прискорювача до KVM через passthrough, KVM створює віртуальну машину через /dev/kvm. Під час призначення GPU VF KVM відключає графічний прискорювач від хост-системи (наприклад, через unbind у випадку NVIDIA), прив’язує його до VFIO PCI та передає гостьовій системі. Всередині KVM відбувається повна ініціалізація MMU GPU, таблиць сторінок та черг команд драйвером графічного прискорювача.

У випадках, коли графічний прискорювач не підтримує SR-IOV, може застосовуватися паравіртуалізація. Гостьове середовище здійснює виклики через API рівня OpenGL або Vulkan, але фактичне виконання цих викликів відбувається на хості. Прикладом такого підходу є VirGL для KVM, де гість передає GL-команди через virtio-gpu до хост-процесу, який використовує Mesa для рендерингу виводу. Такий механізм знижує продуктивність, проте забезпечує ізоляцію гостьового середовища, оскільки воно не має прямого доступу до пам’яті GPU. У Windows Hyper-V подібна логіка реалізована у технології GPU-PV, за якої гостьова система працює з віртуальним драйвером WDDM, що взаємодіє з диспетчером графічного прискорювача на хості та забезпечує кожній віртуальній машині частку графічних ресурсів завдяки часовому поділу.

Ізоляція підтримується також на вищих рівнях програмного забезпечення, зокрема в бібліотеках CUDA, ROCm, OpenCL, а також у менеджерах NVIDIA vGPU GRID та Intel GVT-g. Подібні механізми зазвичай застосовують під час виконання симуляцій і аналізу роботи програм, що використовують нейронні мережі та значні обчислювальні ресурси GPU. У середовищі CUDA ізоляцію забезпечує драйвер, оскільки кожен процес CUDA працює у власному контексті, а всі операції з

пам’яттю виконуються в межах його окремого адресного простору, що гарантує суворе розмежування контейнерів навіть у межах однієї віртуальної машини.

1.3 Абстракція на рівні ОС

На програмному рівні слід згадати sandboxing, який використовується, наприклад, у containerd для Kubernetes на Linux. Техніка sandboxing дозволяє обмежити або ізолювати доступ до системних ресурсів, права на виконання інструкцій із підвищеними привілеями через capability model, небезпечні системні виклики через seccomp, а також контролювати взаємодію з ядром через syscall interception. Це означає, що процес обмежується в межах ОС без необхідності створення окремої віртуальної машини.

Механізм seccomp фільтрує системні виклики (syscall) за допомогою Berkeley Packet Filter (BPF). У ядрі Linux основним захистом є механізми та методи, такі як clone(), unshare(), pivot_root(), mount namespaces, capabilities, cgroups та Linux Security Modules (LSM).

Створення ізольованого процесу в Linux передбачає створення нового простору імен (namespaces) для всіх типів системних об’єктів. Це дозволяє відокремити видимість таких ресурсів, як PID, UTS, NET, IPC, MNT, USER і CGROUP [19].

Таблиця 1.3 - Завдання та способи виклику просторів імен

Тип простору імен	Об’єкт ізоляції	Спосіб виклику
1	2	3
PID	Таблиця процесів	close(CLONE_PID)

Кінець таблиці 1.3

1	2	3
UTS	Ім’я хосту	close(CLONE_NEWUTS)
NET	Інтерфейс/си мережі	close(CLONE_NEWNET)
IPC	Механізми обміну інформації	close(CLONE_NEWIPC)
MNT	Приєднання файлової системи	close(CLONE_NEWNS)
USER	UID/GID простір	close(CLONE_NEWUSER)
CGROUP	Контроль ресурсів	close(CLONE_NEWGROUP)

Також варто нагадати про UID mapping, який робить можливим отримання прав суперкористувача в межах простору імен без наявності відповідних дозволів в основній системі. Важливим механізмом контролю ресурсів тут виступає cgroup. Він формує ієрархічну систему, у якій кожен елемент має свій виділений відрізок ресурсів. До контрольованого складу входять cpu, cpuset, io, memory та інші. Це дозволяє виділяти персоналізовані квоти та встановлювати необхідні ліміти.

Поняття суперкористувача в системі Linux не має суворого визначення, і механізм capabilities натомість розподіляє права на три основні набори - Permitted, Effective та Inheritable [20]:

- Permitted - набір успадкованих можливостей (прав), які може виконувати відповідний процес;

- Effective - не є набором, а бітом, який при встановленні дозволяє виконувати можливості, доступні у наборі Permitted, фактично під час роботи процесу. Наприклад, якщо в Permitted закладено CAP_NET_ADMIN, але цей біт не встановлено в Effective, процес не зможе виконувати мережеві операції;

- Inheritable - спадкові права, які можуть бути передані дочірнім процесам.

Через файлову систему chroot/pivot_root sandbox обмежує доступ до файлової системи контейнера.

Далі йде рівень із використанням OCI runtime, який створює відповідні runc/crun, що, у свою чергу, використовують namespaces, cgroups, seccomp, capabilities та rootfs. Інструмент runc вважається стандартним для запуску контейнерів, оскільки безпосередньо працює з системними викликами Linux, забезпечуючи максимально можливу продуктивність. Його відповідними аналогами є firejail, який легкий для старту без необхідності глибоких знань, та unshare, який ізолює та створює нові простори імен для швидкої ізоляції процесів або мережі.

Зазвичай, якщо контейнер запущений через runc, firejail або unshare, приклад роботи між ними виглядає наступним чином:

- виклик clone() для створення процесу з відповідними прапорами CLONE_NEW*;
- використання setns() у випадку необхідності підключити процес до існуючого простору імен;
- застосування cgroup API для обмеження ресурсів;
- задання uid_map/gid_map для мапування користувачів;
- обмеження прав через capabilities;
- фільтрування системних викликів за допомогою seccomp;
- запуск процесу.

Використання системних викликів Linux у поєднанні з механізмами namespaces, cgroups та seccomp формує основу для створення безпечних та продуктивних контейнерів. Це дозволяє не лише ефективно розподіляти ресурси, але й гарантувати відповідний рівень захисту від несанкціонованого доступу чи некоректної взаємодії між процесами. У результаті OCI runtime виступає ключовим елементом у сучасних системах віртуалізації, забезпечуючи баланс між продуктивністю, гнучкістю та безпекою.

1.4 Прикладне використання віртуалізації на основі рішення containerd

Програмною реалізацією віртуалізації, наприклад, може бути представлений системний демон containerd 2.0. Він ізолює, у більшості випадків, Linux-процеси відповідно до специфікації OCI, на рівні операційної системи, тобто на рівні ядра.

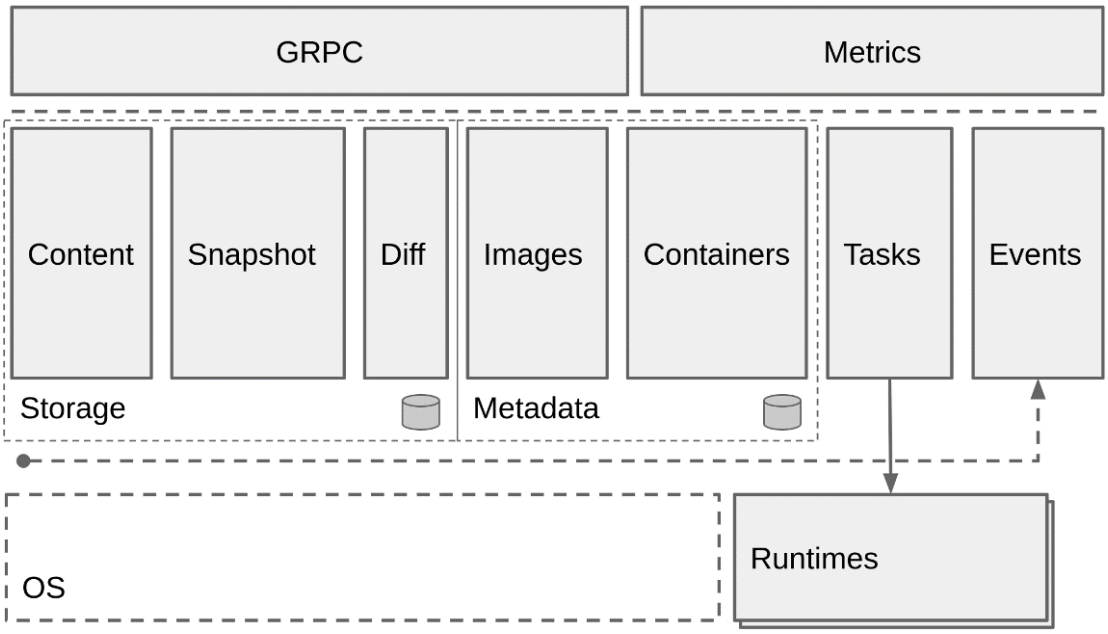

Рисунок 1.5 - Архітектура containerd [21]

Оскільки він є свого роду runtime-середовищем, його використовують контейнерні середовища, такі як Kubernetes чи Docker. Взаємодія між ними відбувається через CRI. CRI є інтерфейсом для визначення взаємодії контейнерів, наприклад Kubernetes, із runtime-середовищами containerd через спеціалізований набір API.

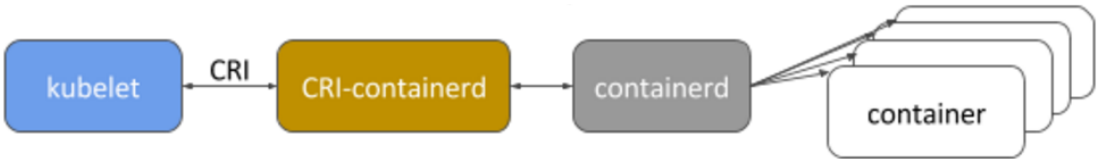

Рисунок 1.6 - Схема взаємодіями з агентом kubelet у Kubernetes [22]

Взаємодія компонентів sandbox у containerd інтегрована через runtime v2 API з використанням gRPC API. Насамперед він задає власну абстракцію namespaces, схожу на Linux namespaces. Це реалізує можливість розподілу наборів контейнерів або образів у рамках одного демона containerd. Варто зазначити, що це більше ізоляція на рівні управління, ніж на рівні ОС, проте вона дуже важлива для різних сценаріїв. Наприклад, контейнер, запущений у просторі A, не бачить простору B. Тут ctr є командним рядком для взаємодії з контейнерами та одночасно менеджером життєвого циклу контейнерів.

Далі йде моделювання контейнера набором параметрів Image, Container, Task. Image - це бінарний образ на фізичному накопичувачі згідно зі специфікацією OCI Image Spec. Він містить необхідні для запуску середовища програмні компоненти, бібліотеки та відповідні залежності. Для кожного образу при створенні генерується відповідний SHA-256 хеш-ідентифікатор. Образ може зберігатися у публічних реєстрах, таких як Docker Hub, для швидкого запуску, або створюватися (збиратися) власноруч, якщо потрібен модифікований варіант середовища та ізоляції, у приватному плані, щоб усе залишалося локально. Модифікувати або збирати контейнер зазвичай прийнято через Dockerfile.

```
# Використовуємо офіційний легкий образ
FROM alpine:3.20

# Команда за замовчуванням — просто показує версію Alpine
CMD ["cat", "/etc/alpine-release"]
```

Рисунок 1.7 - Приклад докерфайлу для автоматизованої збірки Alpine Linux рішення

Але користувач також може додати своє рішення, як наприклад автоматичне розгортання своєї програми або автоматизувати скріпти.

```
# Базовий образ
FROM alpine:3.20

# Встановлюємо curl
RUN apk add --no-cache curl

# Створюємо простий shell-скрипт прямо в контейнері
RUN echo '#!/bin/sh\n\
echo "=== Привіт із модифікованого Alpine! ==="\n\
echo "Спробуємо отримати заголовки з прикладного сайту..."\n\
curl -I https://example.com\n\
' > /hello.sh && chmod +x /hello.sh

# Виконуємо скрипт при запуску контейнера
CMD ["/hello.sh"]
```

Рисунок 1.8 - Приклад модифікації рішення з рис. 1.7

Таким чином після збирання рішення з рис. 1.8, очікуваним буде наступний результат виконання:

```
=== Привіт із модифікованого Alpine! ===
Спробуємо отримати заголовки з прикладного сайту...
HTTP/2 200
content-type: text/html; charset=UTF-8
```

Рисунок 1.9 - Очікуваний результат виконання

Container - це екземпляр образу, який буде виконуватися в ізольованому середовищі. Він є фактичним середовищем програми, запущеної за допомогою Image. Варто наголосити, що container лише зчитує програму з бінарного образу та запускає її, тоді як Image просто зберігається на накопичувачі. Наприклад, для

запуску серверу Redis буде використано вже зібраний образ, який збережений у системі.

Task - це запущений процес у просторі імен, яким керує containerd через механізм shim. Він знаходиться на вищому рівні абстракції та є основним завданням контейнера. Відповідно, якщо це сервер Redis, процес повинен пройти етапи запуску процесів, їх моніторингу та надання необхідних ресурсів. Далі відбувається керування станом процесів шляхом відстеження виконання завдань, а також надання можливості ставити виконання на паузу, перезапускати середовище або повністю зупиняти його виконання.

Варто також зазначити, що механізм shim у цьому контексті є свого роду ізолятором для контейнерів через fork або clone з відповідними значеннями. Наприклад, взаємодію між контейнером та ядром системи можна пояснити як процес координування викликів та моніторингу інших процесів [23].

Таблиця 1.4 - Примітиви та їх способи використання

Примітив syscall kernel або facility	Процес використання
1	2
fork(), clone(), execve()	Створення процесів shim’ом
waitpid(), wait4()	Очікування завершення дочірніх процесів і передача exit code
signalfd(), kill(), tgkill()	Надсилання сигналу контейнеру SIGTERM або SIGKILL
epoll(), select(), poll()	Моніторинг сокетів типу stdin, stdout, stderr, ttrpc, health.
unshare(), setns()	Встановлення простору імен чи проведення операцій які потребують приєднання до простору імен.
mount(), pivot_root()	Координація життєвого циклу приєднання або від’єднання.

Кінець таблиці 1.4

1	2
cgroupfs, sysfs	Зчитування або запису у cgroup контроллери для моніторингу або створення відбитку процессу.
Unix domain sockets	IPC між shim та containerd за допомогою gRPC або ttrpc.
prctl(), setrlimit()	Встановленя Runtime'мом лімітів та дозвіл shim робити звіти або зчитувати інформацію.
ptrace()	Інструмент для відлагодження

Тому доволі важливо враховувати точні, або майбутні, встановлення які у деяких випадках можуть бути вже не доступними після запуску контейнера.

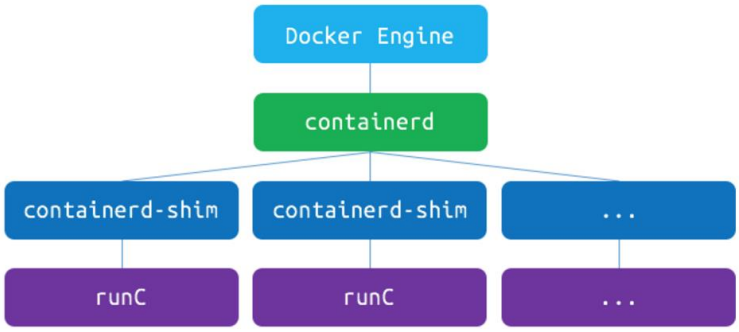

Рисунок 1.10 - Ієрархічний приклад запуску середовищ [24]

Доречно також зазначити підтримку контейнеризації графічних прискорювачів. Це досягається шляхом використання device plugin model, де кожен контейнер отримує власне рішення vGPU або MIG. У такому випадку ізоляція забезпечується трьома способами. Перший - через cgroups devices, наприклад у

/dev/nvidia*. Другий - через NVIDIA Container Runtime, який, у свою чергу, створює спеціальний простір імен для графічного прискорювача. І третій - через Kubernetes GPU device plugin, у якому для кожного середовища виділяється окремий відрізок графічного прискорювача.

1.5 Використання віртуалізації у контексті Malware Analysis

Завдяки таким рішенням, загалом на ринку ПЗ є можливість створювати середовища для закритого, ізольованого аналізу роботи шкідливого програмного забезпечення. Прикладом такого рішення є ANY.RUN. Воно є інтерактивним середовищем для швидкого запуску та аналізу підозрілого ПЗ з метою визначення методів його роботи на різних ОС. Такі рішення зазвичай використовують команди аналітики безпеки для оцінки ризиків або для внесення постфактум змін у політику безпеки шляхом створення правил, наприклад, YARA, IPS або IDS.

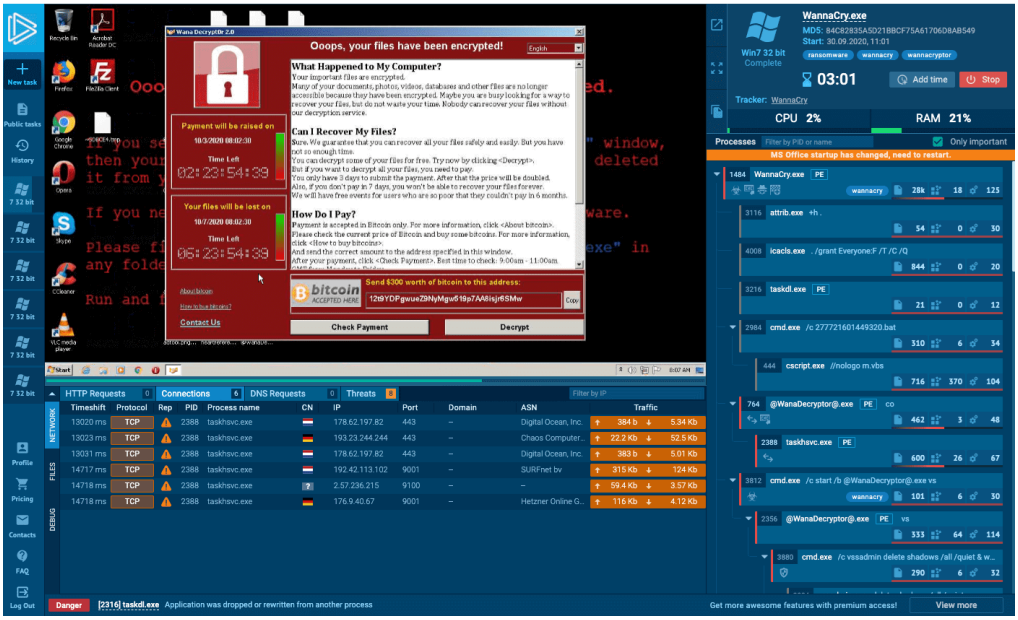

Рисунок 1.11 - Інтерфейс середовища у ANY.RUN [25]

На рис. 1.11 відображено, яким чином відбувається зараження системи Windows 7 шкідливим ПЗ WannaCry. Такий інтерфейс дозволяє проводити симуляцію в реальному часі, завдяки чому користувач може бачити алгоритм зараження та перелік адрес, до яких відбуваються звернення під час роботи шкідливого ПЗ. Якщо користувач бажає створити власне середовище, він може використати гіпервізорні рішення на кшталт VMware Workstation Pro або Docker для контейнеризації, але важливо заздалегідь врахувати нюанси подальшого перенесення середовища. У випадку VMware середовище буде зібране в один файловий образ, сумісний виключно з продуктами VMware. Для Docker контейнер має бути зібраний у OCI-сумісний архів.

У програмному плані всередині таких середовищ можуть використовуватися інструменти для дизасемблінгу або деобфускації файлів, аналізатори патернів YARA, моніторингові системи для аналізу в реальному часі - такі як Process Monitor, Wireshark для мережевого моніторингу, API Monitor для перехоплення викликів WinAPI, Volatility для аналізу дампів пам’яті та пошуку артефактів.

Ключовим елементом таких систем є модулі або плагіни для VMI [26], які працюють із віртуальними системами. Ці зовнішні монітори збирають інформацію про системні виклики, файлові зміни, мережеві з’єднання, доступ до графічних прискорювачів, пристроїв вводу-виводу, драйверів та інші події.

З технічної точки зору (на прикладі KVM), гіпервізор має доступ до пам’яті гостьової системи через таблиці сторінок EPT або NPT, тому VMI-монітору необхідно через API виконувати такі операції, як зчитування віртуальних адрес гостя, визначення структури ядра (наприклад, у Windows), спостереження за змінами структур процесів або драйверів, а також аналіз графічних викликів DirectX чи OpenGL, якщо це стосується аналізу GPU-активності.

Взаємодія між хостовим і гостьовим середовищами може бути реалізована двома способами.

Перший спосіб - використання внутрішнього агента, який працює всередині гостьової ОС і передає інформацію через захищений канал (TCP, Named Pipe, VirtIO socket) на хостову систему, зазвичай у форматі JSON. Якщо користувач не бажає, щоб між хостом і гостем був прямий зв'язок, він може додати окрему віртуальну машину, яка виконуватиме роль аналітичного хоста. З'єднання між експериментальною та аналітичною машиною в такій схемі доцільно налаштовувати у вигляді мостового або локального мережевого (LAN) підключення. Такий підхід є навіть кращим, оскільки дозволяє додати третю машину - симулятор мережі (наприклад, NAT), яка генеруватиме відповіді на запити експериментальної машини. Це актуально для аналізу поведінки шкідників на кшталт ScreenLocker'ів, що намагаються звертатися до мереж Ethereum або BTC. Зазвичай канал між експериментальною та аналітичною системами є одностороннім (наприклад, VirtIO-serial у режимі «вхід» або використання апаратного data diode).

Другий спосіб - використання зовнішнього монітора. Це актуально через те, що внутрішній агент може бути виявлений і знешкоджений шкідливим ПЗ.

Зовнішній out-of-band моніторинг працює на рівні гіпервізора і зчитує стан системи ззовні, не залишаючи видимих слідів всередині гостьового середовища.

1.6 Теоретичні відомості про квантові системи

Контекстна інформація про квантові системи починає формуватися з 1980-х років, пояснюючи ідею того, що квантові системи можуть бути створені або, точніше, змодельовані іншими квантовими системами. Ця ідея поклала початок стрімкому розвитку квантових технологій, починаючи з квантових симуляторів [27].

Важливо також підкреслити відмінності між цими поняттями. Квантові системи, у більшості випадків, є фізичними об'єктами, що підпорядковуються

законам квантової механіки. Це можуть бути фотони в оптичній резонаторній системі, надпровідникові кубіти або електрон у потенційній ямі. Вони спрямовані на вивчення власних властивостей та реалізацію обчислень, проте тип керування визначається фізичною природою об’єкта. Натомість квантовий симулятор розуміється як керована квантова система, яка моделює іншу квантову систему. Конкретними прикладами є надпровідниковий кубіт як квантова система та іонна решітка, що імітує магнітні взаємодії у моделі Гайзенберга. Сучасне пояснення полягає у використанні базового апаратного рівня з відповідною побудовою квантових елементів і прикладних інструментів для вивчення інших явищ у квантовому контексті [27].

Квантові симулятори зазвичай поділяють на два основні типи, а саме аналогові та цифрові [27]. Аналогові симулятори максимально точно відтворюють фізичну модель квантової системи для детального дослідження. Вони реалізуються за допомогою іонних пасток, оптичних решіток або надпровідникових контурів і застосовуються для вивчення фазових переходів, квантових магнітних ефектів та інших частинкових явищ. Аналогові симулятори особливо ефективні для моделювання природних квантових процесів, забезпечуючи коректне відтворення фізики у конденсованих середовищах. Наприклад, їх використовують для моделювання Ізинговських моделей або моделі Бозе-Газена.

Цифрові симулятори орієнтовані на використання квантових алгоритмів і створюються на базі багатозадачних квантових комп’ютерів. Вони застосовуються, наприклад, для реалізації алгоритму Троєва при моделюванні багаточастинкових систем або алгоритму Вігнера для динамічного моделювання квантових процесів. Крім того, цифрові симулятори дозволяють виконувати більш загальні обчислення на квантовій швидкості, включно із задачами квантової фізики, хімії, криптографії та комбінаторики.

Реалізація квантових симуляторів у сучасних умовах зазвичай вимагає розробки спеціалізованих апаратних платформ та підходів до створення матеріалів.

Першим прикладом є іонні пастки, де заряджені атоми або іони утримуються в електромагнітних полях та взаємодіють через лазерні імпульси. Кубіти на основі іонних пасток характеризуються високою точністю та можливістю маніпуляцій, таких як вимірювання та операції над квантовими станами [28].

Іншим підходом є надпровідні кубіти, що використовують надпровідні матеріали та елементи, через які проходить струм за умови дотримання критичної температури. На відміну від іонних пасток, вони схильні до декогеренції через взаємодію з навколишнім середовищем.

Використання фотонів у квантових системах дозволяє кодувати квантові стани за допомогою світла. Через те, що фотони рухаються зі швидкістю світла та не мають власної маси у звичному розумінні, вони особливо ефективні для моделювання квантових процесорів, здатних передавати інформацію на великі відстані. Водночас широке застосування фотонних систем обмежене складністю оптичних конструкцій.

Згадуючи настільки малі системи, варто також звернути увагу на напрямок розвитку квантових систем із використанням атомів або молекул. У цьому випадку вони функціонують на основі штучно створених квантових станів, наприклад, холодних атомів, охолоджених за допомогою лазера до наднизьких температур. Це дозволяє моделювати явища, специфічні для квантових систем, такі як квантові фазові переходи або взаємодії в молекулярних станах.

Найактуальнішим напрямком у 2025 році є методи, засновані на топологічних ефектах [29]. У таких системах використовують топологічні надпровідники, завдяки чому топологічні кубіти ґрунтуються на екзотичних частинках, таких як майоранові ферміони. Особливість їхньої роботи полягає у топологічній заплутаності, що дозволяє зберігати інформацію у стані, більш захищеному від декогеренції. Компанія Microsoft, яка представила чіп Majorana 1, заявляє, що він може стати більш доступним для повсякденного використання, хоча на сьогодні квантові

системи залишаються переважно корпоративними через високу вартість і складність обслуговування.

Як зазначено, Majorana 1 використовує квазічастинки майорани, що є античастинками. Новим матеріалом для створення таких систем був запропонований топологічний надпровідник - сплав арсеніду індію, який є напівпровідником, та алюмінію, що є надпровідником.

При охолодженні до температур, близьких до абсолютного нуля (приблизно −400 °F або −200 °C), та застосуванні спеціально налаштованого електромагнітного поля, матеріал формує топологічні надпровідні нанодроти з нульовими модами майорани на кінцях.

Важливою деталлю є потенційна можливість розміщення на одному кристалі кількості кубітів, близької до мільйона одиниць, зі збереженням стабільності завдяки топологічним особливостям. Велика кількість кубітів забезпечує високу швидкодію, незважаючи на компактні розміри кристала, що наближені до розміру долоні. До цього додається можливість цифрового керування, що підвищує точність та спрощує контроль над кубітами. Такий перехід можна порівняти з трансформаційними змінами в класичних комп’ютерних системах.

Компанія Google також розробляє квантову екосистему під назвою Quantum AI [30]. Вона базується на власних чипах Google із технічними назвами Sycamore, Willow та Bristlecone, поєднуючи їх із традиційними апаратними компонентами, такими як кріостати, мікрохвильові контролери та надпровідні кубіти, контроль над якими здійснюється за допомогою калібрування та спеціальних планів калібрування. Чипи орієнтовані на різні топологічні та прикладні задачі, зокрема Sycamore використовує прямокутну топологію, тоді як Willow розроблений для забезпечення підвищеної шумової стабільності.

У своїй архітектурі ці системи використовують масив транмон-кубітів, розташованих у регулярній решітці. Це дозволяє реалізувати особливий варіант взаємодії між кубітами для виконання двокубітних гейтів.

Система працює за принципом програмної віртуалізації, але у дещо іншому сенсі. Віртуалізацією тут вважається не ізоляція ресурсів, а програмна емуляція реальної квантової системи для класичного середовища, що дозволяє користувачу отримати досвід роботи, наближений до реальної квантової платформи. Віртуалізація здійснюється двома способами, а саме як симуляція в ідеальних умовах (без шуму) та з урахуванням шумових ефектів.

Кубіти у системі зазвичай організовуються у графи взаємодій, де вузли відповідають кубітам, а ребра - фізичним зв’язкам для виконання двокубітних гейтів, таких як CZ або iSWAP. Кожна топологія визначає кількість кроків, необхідних для зв’язування двох довільних кубітів, чутливість сусідніх кубітів до cross-talk, складність маршрутизації квантових станів та кількість додаткових SWAP-гейтів, які доведеться виконати.

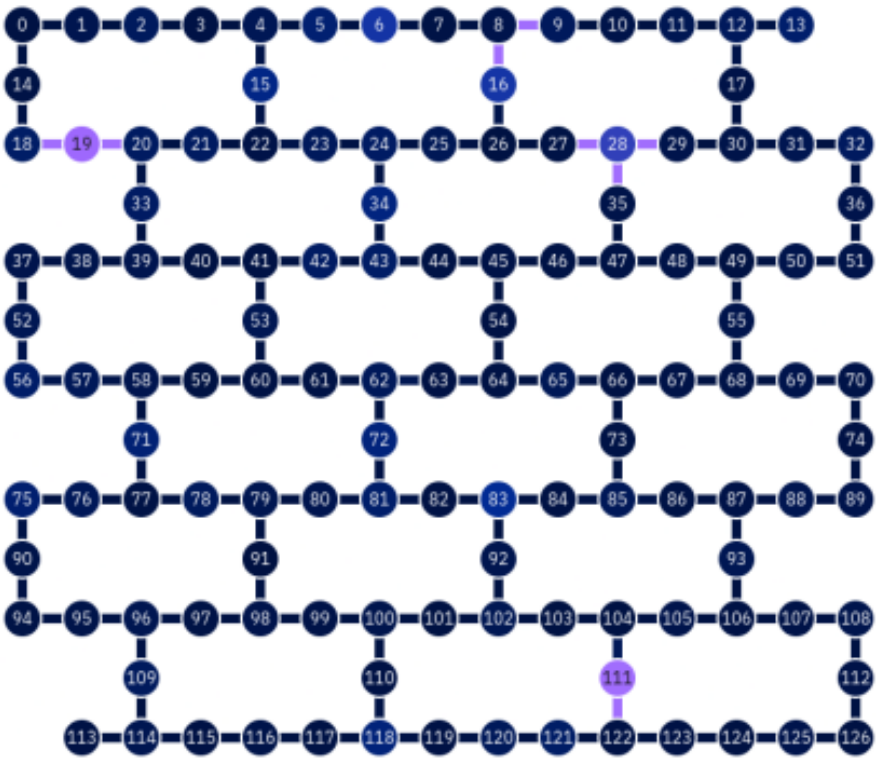

Рисунок 1.12 - Побудова чипу Eagle з 127-ма кубітами [31]

Для квантової віртуалізації ці аспекти є критично важливими, оскільки гіпервізор або планувальник повинен знати, які групи кубітів можуть бути ізольовані одна від одної, а які - ні, для забезпечення стабільної роботи квантової віртуальної машини.

1.7 Основні проблеми віртуалізації у квантових системах

Для початку необхідно визначити основні принципи, на яких ґрунтуються квантові симулятори. Одним із ключових є принцип суперпозиції, що лежить в основі квантових обчислень та дозволяє з високою швидкістю виконувати, наприклад, аналіз або пошук у великих базах даних, а також отримувати результати, недоступні для класичних систем [32].

Іншим важливим аспектом є взаємозв’язок між кубітами, що визначається явищем квантової заплутаності, а також інтерференція взаємодії кубітів, яка забезпечує управління різними квантовими ймовірностями. Поняття «декогеренція» у контексті квантових технологій стало усталеним, оскільки явище втрати когерентності з’являється у більшості квантових задач. Також слід враховувати ефект квантового тунелювання, що дозволяє квантовим частинкам долати потенційні бар’єри, які не можна подолати в класичних умовах. Такими бар’єрами можуть бути молекулярні або енергетичні структури, а також властивості нових матеріальних систем.

Різниця у рівнях ізоляції між квантовими системами та квантовими симуляторами полягає у фізичній основі. У перших очікується фізична ізоляція, тоді як у другому випадку - програмна. Для забезпечення інформаційної захищеності застосовуються топологічні коди та кодування цільової динаміки у контрольованих підсистемах.

Архітектурно квантові системи мають особливості, які неможливо реалізувати у класичних системах:

- теорема неможливості клонування (no-cloning theorem) [33];

- чутливість квантових станів (можливість декогеренції, тобто втрата когерентності або узгодженості фаз між складовими суперпозиції);

- змішування квантових станів, що неминуче призводить до інтерференції.

Більш детально кожен із принципів можна охарактеризувати так. Теорема неможливості копіювання є одним із фундаментальних положень квантової механіки і була сформульована у 1982 році Вуттерсом та Зуреком [34]. У класичних бітових системах дозволено дублювати ресурси та створювати побітові копії будь-яких даних. У квантовій системі копіювання невідомого квантового стану є неможливим. Така заборона є частиною структури законів квантової механіки, зокрема принципів лінійної еволюції станів та суперпозиції. У класичних системах вся інформація представлена у вигляді потоку бітів (нулів або одиниць), якими можна маніпулювати без втрат чи змін. Якщо створити копію будь-якої послідовності, наприклад 011000, то кожна точна копія буде ідентична оригіналу та не відрізнятиметься від системи до системи. Це можливо, оскільки класичні біти можна повністю виміряти без впливу на них, і будь-яке копіювання не модифікує об’єкт, залишаючи його без змін.

Квантові системи, навпаки, функціонують за іншими принципами. Їхні стани описуються вектором у гільбертовому просторі та можуть перебувати у суперпозиції базисних станів. У квантових системах носієм інформації є кубіт, який на відміну від класичного біта, що може перебувати у стані 0 або 1, може знаходитися не лише у стані $|0\rangle$ чи $|1\rangle$, а й у стані лінійної комбінації [35]:

$$a|0\rangle+\beta|1\rangle, \tag{1.1}$$

де a та β - комплексні амплітуди, що визначають ймовірності результатів вимірювань.

Тому, якщо користувач не знає точних значень α і β, він не може визначити їх без впливу на стан системи. Вимірювання у квантових системах спричиняє «колапс» у один із базисних станів, змінюючи початкову суперпозицію. Сама теорема також пояснює неможливість клонування тим, що навіть якби існував унітарний оператор U, який здійснює клонування, він мав би задовольняти умові $U(|\psi\rangle|0\rangle) = |\psi\rangle|\psi\rangle$ для всіх можливих станів $|\psi\rangle$. Через це виконання вимоги лінійності у квантовій механіці стає неможливим. Наприклад, для стану суперпозиції:

$$\frac{|0\rangle + |1\rangle}{\sqrt{2}}, \tag{1.2}$$

неможливо отримати клон, який би зберіг цю суперпозицію, оскільки застосування унітарного оператора дало б результат, що відрізняється від звичайного класичного клонування. Саме цей принцип істотно змінює базові розрахунки квантових обчислень та криптографії. Простим прикладом є протокол BB84, безпека якого ґрунтується на тому, що навіть якщо зловмисник перехопить і спробує скопіювати квантові біти у каналі передачі даних, його дії не залишаться непоміченими. Як було зазначено раніше, перехоплення та вимірювання неможливі без зміни стану кубіта, а сам цей процес автоматично виявляє втручання.

Ще одним поясненням цієї теореми є те, що бітовий стан у класичних системах являє собою набір, наприклад, значень напруги або станів магнітного поля компоненту, тоді як у квантовій механіці він описується набором ймовірних майбутніх результатів вимірювань. Визначити ці параметри без клонування неможливо. Цей принцип тісно пов’язаний із аспектом «неможливості точного вимірювання без збурення». Якщо необхідно дізнатися, у якій суперпозиції перебуває кубіт, доводиться проводити серію вимірювань у різних базисах, використовуючи велику кількість ідентичних копій. Це створює ситуації, коли втручання в один кубітний регістр може змінити всю схему, наприклад у випадках

квантової телепортації, змушуючи систему корекції помилок працювати за принципом «доміно».

Теорема no-cloning також забезпечує необхідні властивості відмовостійкості та унікальності станів. У класичних системах для збереження важливих даних використовують пряме копіювання через резервні копії, що дозволяє відновлювати їх у разі збою. У квантових системах такий підхід неможливий, і натомість застосовуються методи квантової корекції помилок із використанням заплутаності та надлишкових кубітів для збереження логічних станів без прямого копіювання. Цей приклад демонструє, наскільки принцип теореми no-cloning інтегрований в архітектуру квантових систем.

Чутливість квантових станів кількісно характеризується двома параметрами, а саме часом релаксації T_1 та часом декогеренції T_2. Параметр T_1 визначає середній час, протягом якого кубіт переходить зі стану $|1\rangle$ у стан $|0\rangle$. Параметр T_2 характеризує середній час збереження когерентності (transverse relaxation time), тобто час, упродовж якого зберігається фазова інформація у стані суперпозиції. У більшості систем виконується співвідношення $T_2 \leq 2T_1$. Навіть якщо кубіт не втрачає енергію, випадкові фазові зрушення можуть руйнувати інтерференційні властивості квантового стану [36].

Сучасні надпровідникові кубіти демонструють значення T_1 і T_2 у діапазоні від кількох десятків до кількох сотень мікросекунд. Це означає, що час, протягом якого система зберігає початковий стан, є значно обмеженим, на відміну від класичних систем, де дані можуть зберігатися тривалий період. Дослідження у сфері квантової пам’яті вже активно проводяться, однак на практиці такі рішення поки що залишаються нестабільними або мають суттєво обмежений час утримання квантового стану.

Якщо розглядати ситуації виникнення декогеренції, то вона зумовлена різними причинами. Теплові флуктуації температури, коли будь-яка температура, відмінна від абсолютного нуля, змушує атоми та електрони рухатися, створюють шум, який

впливає на кубіт. Тому стабільною робочою температурою у квантових кріостатах зазвичай вважають 10-20 міліКельвінів. Електромагнітне випромінювання, навіть слабке, здатне збурювати стан кубіта.

Ще однією особливістю квантових систем є обмеженість кубітів як ресурсу. На відміну від класичних систем, де можна додавати необмежену кількість бітів, у квантових системах збільшення числа кубітів ускладнює підтримку когерентного стану. Значне зростання кількості кубітів підвищує кількість каналів втрати когерентності. Тому виникає складна інженерна задача, оскільки виконання складних квантових алгоритмів вимагає збільшення числа кубітів, що, у свою чергу, ускладнює підтримку їхньої когерентності.

Для боротьби з декогеренцією розроблено різні методи, які допомагають мінімізувати її вплив. Серед них - квантова корекція помилок [37] та динамічний декуплінг [38], під час якого застосовуються спеціальні імпульси для компенсації накопичених фазових зрушень, що подовжує час когерентності.

Іншою важливою проблемою є топологія зв’язків у квантовому процесорі. Для виконання двокубітних операцій, наприклад CNOT, кубіти повинні бути фізично зв’язані або мати можливість взаємодії через проміжні кубіти. Тому навіть при логічному розділенні вони можуть залишати взаємні зв’язки, оскільки повноцінна ізоляція у квантових системах потребує повного припинення взаємодії, що в деяких випадках неможливо без фізичної перебудови архітектури.

Варто також враховувати, що навіть найсучасніші матеріальні реалізації не здатні повністю усунути декогеренцію, оскільки вони лише подовжують час, протягом якого можуть виконуватися квантові обчислення. Тому необхідно враховувати часові обмеження, щоб забезпечити стабільне завершення операцій до втрати когерентності. Чутливість квантових станів означає, що помилки не виникають періодично, як у класичних системах. У квантових системах помилки відбуваються постійно та здатні накопичуватися без активної корекції, повністю знищуючи очікуваний «правильний» результат розрахунків.

Коли фізичні кубіти залишаються елементами однієї фізичної квантової машини, вони взаємодіють через спільні поля, шум та взаємні впливи, описані хвильовими функціями. У результаті їхні стани можуть накладатися один на одного, створюючи інтерференційні ситуації. Інтерференцію розуміють як явище складання ймовірностей амплітуд різних станів, що може бути конструктивним або деструктивним, і результат вимірювань залежить від цього складання.

Якщо два квантові стани, які повинні бути незалежними, стають частково когерентними (мають ненульову перекриваність хвильових функцій), це автоматично змінює розподіл ймовірностей вимірювань у межах кожної віртуальної системи. Замість ізольованої роботи кожного середовища користувач отримує на виході змішаний еволюційний процес, у якому інформація з одного набору кубітів впливає на інший навіть без наявності прямого каналу зв'язку між ними.

Математично явище змішування пояснюють за допомогою двох типів квантових станів, а саме чистих станів, які описуються хвильовою функцією, та змішаних станів, що подаються матрицею густини ρ і відображають статистичну суміш можливих станів системи. Нехай існують дві незалежні квантові системи A та B, які не мають спільної взаємодії. У такому випадку їх спільний стан описується тензорним добутком $\rho_{AB} = \rho_A \otimes \rho_B$. Однак після появи навіть слабкої взаємодії між системами їхній спільний стан уже не можна подати у вигляді простого добутку. Він набуває корельованої або квантово-заплутаної структури $\rho_{AB} \neq \rho_A \otimes \rho_B$. У цьому випадку кореляції між підсистемами є математичним відображенням процесу змішування та формування спільної квантової поведінки.

На основі цього, при спробі реалізувати віртуалізацію на квантовій системі розглянемо приклад, коли певна кількість кубітів виділена для користувача 1, а інша - для користувача 2. Формально може бути виділений перший підпростір Гільбертового простору для групи 1 та ще один підпростір для групи 2. Виникає проблема, що фізично розміщені кубіти у спільному середовищі взаємодіють через ефекти cross-talk, теплові коливання, спільні магнітні поля або лазерне керування.

Отже, навіть якщо користувач встановлює їх математично як відокремлені, існує ймовірність того, що фізично-хвильові функції кубітів цих систем зазнають впливу загальних флуктуацій, що викликає випадкові зсуви фаз або амплітуд. У деяких контрольованих алгоритмах допускається використання накладення амплітуд для підсилення ймовірності певного результату, наприклад, при спеціально створюваній інтерференції в алгоритмах Гровера або Шора. Проте для реалізації віртуалізації це може стати критичною проблемою, коли випадкове змішування призводить до того, що результат вимірювання в одному віртуальному середовищі перестає бути незалежним і ізольованим від іншого середовища, що порушує коректне виконання паралельних обчислень.

На даний момент фізичні реалізації квантових комп’ютерів містять безліч джерел такого змішування. У надпровідних кубітах, наприклад, кубіти розташовані на спільному чіпі та керуються мікрохвильовими імпульсами[39]. Навіть наявність власного резонансного діапазону може створювати паразитні cross-збудження сусідніх кубітів. Іонні пастки реалізують кубіти як внутрішні стани іонів, що перебувають у спільному електромагнітному полі. Навіть за умови локалізованого впливу лазерного променя існує ненульова ймовірність того, що частина енергії потрапить на інші іони.

У фотонних квантових системах кубіти реалізуються як поляризаційні або часові моди фотонів. Тому, у цьому випадку постає проблема, оскільки у спільних оптичних елементах фотони різних віртуальних потоків можуть випадково інтерферувати через накладення їхніх шляхів або частотних спектрів.

1.8 Гіпотетичні рішення проблем віртуалізації у квантових системах

Можливі гіпотетичні шляхи реалізації також існують, наприклад, використання часової мультиплексії кубітів [40]. Ідея полягає у тому, що, як і в звичайних системах, система перемикається між процесами, виділяючи короткі часові слоти для виконання своїх завдань. Оскільки ці перемикання відбуваються у дуже малий проміжок часу (від мілісекунд до мікросекунд), користувачу здається, що програмне оброблення відбувається одночасно. Таким чином створюється ілюзія паралельного виконання, аналогічно до випадку використання одноядерного процесора. Навіть у сучасних системах цю ідею поєднують з реальним паралелізмом, оскільки вона досі залишається складовою операційної логіки.

Враховуючи принципи теореми про неможливість копіювання та особливості змішування станів, теоретично часова концепція дозволяє реалізувати подібний підхід. Суть полягає в тому, що замість спроби виконувати дві або більше квантових програм на одному наборі кубітів, їх можна виконувати послідовно, дуже швидко перемикаючи між собою, що створює ілюзію паралельності. У такому випадку, у системі з 100 кубітів користувач може отримати дві програми одночасно, якщо кожна з них потребує лише 50 кубітів для виконання обчислень. Замість фізичного розділення системи програма 1 використовує кубіти для обчислень, після чого стан кубітів повністю скидається, ініціалізується заново та запускається програма 2. Якщо цей процес відбувається достатньо швидко, користувач отримує ілюзію паралельного виконання програм, хоча насправді вони виконуються строго послідовно.

Ця концепція схожа на time-slicing у класичних операційних системах, але має суттєві відмінності через специфіку квантових обчислень. У класичних системах процес може бути зупинений у будь-який момент та збережений у пам’яті для подальшого відновлення роботи з точки зупинки. Натомість у квантовій системі

зняти «зліпок» стану неможливо. Тому при часовому перемиканні кубітів не зберігається жодний попередній стан, а обчислення починається знову.

Такий підхід з погляду часової організації та віртуалізації враховує наявність затримки під час кожного перемикання станів. У класичних системах скидання регістру триває наносекунди, тоді як у квантових системах повернення кубіта до базового стану не є тривіальною операцією і потребує попередньої підготовки. Спочатку необхідно дочекатися, поки кубіт релаксує із збудженого стану до основного, що займає мікро або мілісекунди. Після релаксації слід переконатися, що кубіт не залишився в частково змішаному стані. Це вимагає додаткових калібрувань, які також потребують часу. У глобальному вимірі така процедура створює черги на виконання операцій.

Такі черги зазвичай поділяють за складністю або за необхідністю отримання послідовних результатів. Тому часова мультиплексія може розглядатися як технічний компроміс у випадку, коли квантові ресурси обмежені, а багатокористувацький доступ є необхідним. Наприклад, можлива організація паралельних обчислень на двох фізичних квантових системах із перенесенням залишкових обчислень на третю систему через часову мультиплексію. Подібний принцип частково реалізовано у сучасних хмарних сервісах IBM Quantum та Amazon Braket, хоча це не є повною віртуалізацією у класичному розумінні.

Просторова мультиплексія є ще однією гіпотетичною стратегією у контексті квантових обчислень [41]. У цьому підході фізичні кубіти розподіляють на підмножини, кожна з яких функціонує як незалежне віртуальне середовище. Концептуально ідея є простою, оскільки якщо система має 100 кубітів, її можна поділити на чотири незалежні сегменти та розмістити в окремих фізичних просторах, що створює імітацію паралельності. Це можуть бути окремі канали, модулі чи тракти. Подібність до класичного «розділення» особливо помітна в оптичних системах, де фотони передаються різними каналами. Однак такий підхід

ускладнює синхронізацію між каналами через підвищені вимоги до точності розміщення та ізоляції.

Слід також зазначити існування схожого підходу - розділення квантово-класичного контролю [42]. У ньому намагаються адаптувати принципи класичного керування великою кількістю задач до квантового рівня, враховуючи специфічні обмеження квантових технологій. Такий підхід передбачає класичну систему обробки, яка включає контролер послідовної передачі сигналу, та квантовий процесор як елемент обчислення. Якщо просторова мультиплексія здебільшого стосується фізичного розміщення компонентів квантових систем для досягнення аналогії з паралельністю, то квантово-класичне розділення фокусується на розмежуванні обробки між класичними та квантовими модулями. Основними викликами цього підходу є затримки між квантовими і класичними компонентами та потреба у високошвидкісному інтерфейсі.

Концепція квантової гіпервіртуалізації на основі поверхневого коду є найбільш наближеною до ідеї реальної віртуалізації квантових ресурсів. У цьому підході застосовують квантову корекцію помилок для формування логічних кубітів із великої кількості фізичних кубітів. Це дає змогу розглядати такі логічні кубіти як ізольовані, стабільні обчислювальні елементи, які, у свою чергу, можуть бути розподілені як необхідне віртуальне середовище для виконання квантових програм.

1.9 Підхід HyperQ до квантової віртуалізації

Реалізація повноцінного квантового гіпервізора за останній рік набула значного прискорення. Зокрема, система HyperQ стала одним із найпомітніших проривів у квантових технологіях, її було представлено у 2025 році дослідниками Columbia Engineering [43]. Ця система створює віртуальні квантові машини (qVM),

що частково розв’язують проблему однозадачності квантових процесорів, забезпечуючи більш ефективний мультизадачний режим.

У загальних рисах HyperQ складається з трьох ключових компонентів, гіпервізора, планувальника та фізичного квантового пристрою (квантового чипа).

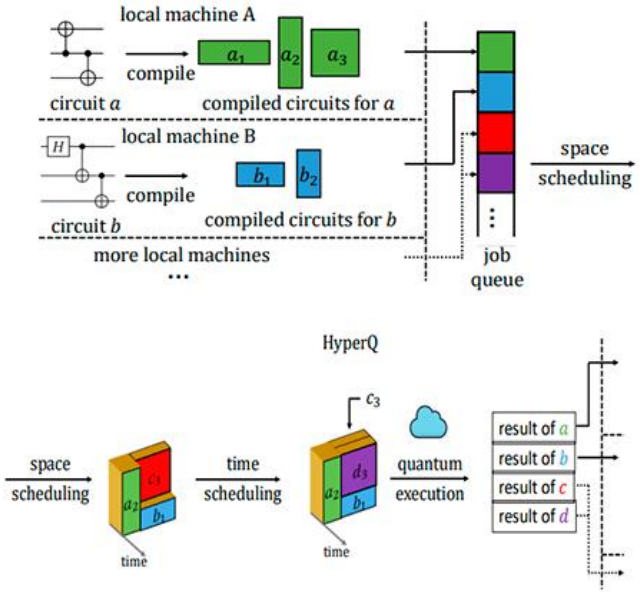

Рисунок 1.13 - Робоча схема системи HyperQ [44]

Фізичним квантовим пристроєм у цій схемі може бути будь-який сучасний квантовий чип. У HyperQ використано саме чип IBM, який містить 127 кубітів у системі Brisbane, побудованій на основі архітектури Eagle, та доступний через хмарний сервіс IBM Quantum. Гіпервізор у цьому випадку виступає програмним шаром, що здійснює розподіл фізичних кубітів між різними програмами.

Робота гіпервізора полягає в тому, що фізичні кубіти поділяють на підгрупи, розбиваючи загальний фізичний простір на кілька qVM. Кожна qVM отримує виділену підмножину кубітів, яку ізолюють шляхом формування буферної зони навколо відповідної підсистеми. Такий підхід забезпечує зменшення перехресних впливів та шумів між логічно розділеними областями. У низці випадків отриманий

логічний простір можна розглядати як аналог оперативної пам’яті або процесорних ресурсів.

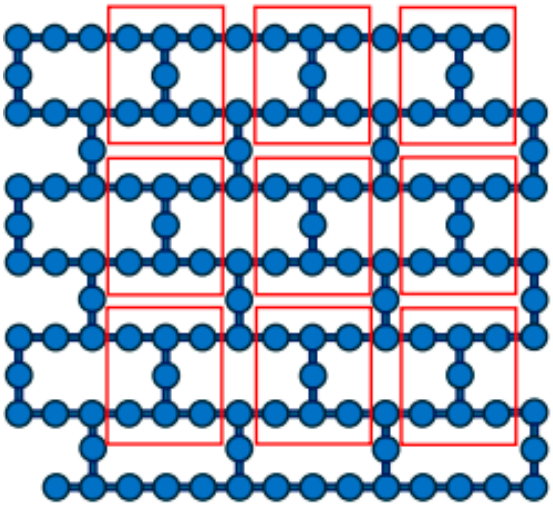

Рисунок 1.14 - Сітка базової мапи юнітів [44]

Сітка, наведена на рис. 1.14, демонструє початкову графову топологію фізичних кубітів. Порожні вузли між виділеними областями утворюють буферну зону, яка знижує ризик крос-декогеренції та неконтрольованих кореляцій між підсистемами.

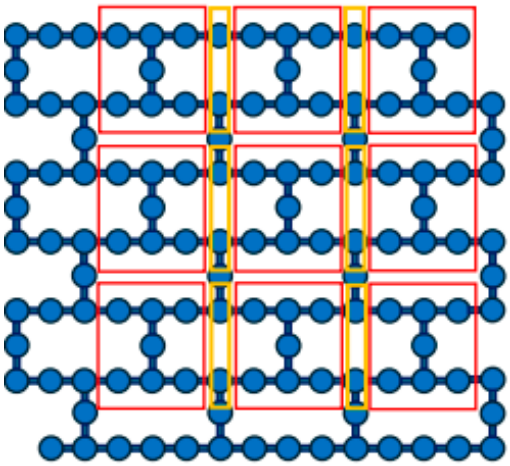

Рисунок 1.15 - Вертикальне з’єднання qVM [44]

Наступним етапом є механізм вертикальної інтеграції між qVM, зображений на рис. 1.15. Його зазвичай використовують у випадках необхідності передачі квантової інформації між різними областями. Вертикальне з'єднання забезпечує логічний канал між qVM, який фізично реалізовано як тимчасову квантову шину або шлях телепортації стану кубіта.

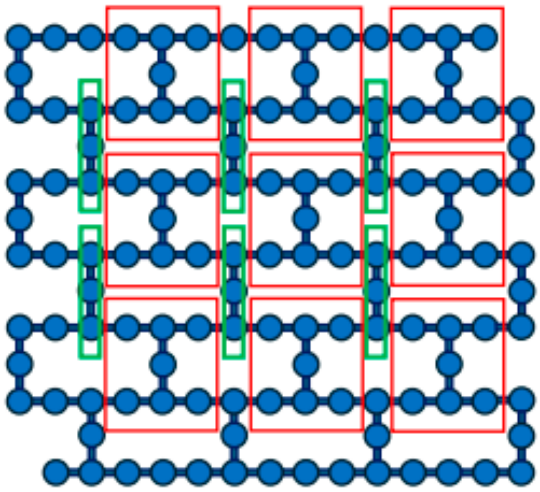

Рисунок 1.16 - Горизонтальне з'єднання qVM [44]

Третім кроком є горизонтальне з'єднання, яке забезпечує інтеграцію qVM у випадках, коли вони працюють у спільному часовому просторі над одним комбінованим завданням.

Наступним важливим елементом є планувальники. Планувальниками виступають інтелектуальні алгоритми, які аналізують потреби кожної задачі, зокрема кількість кубітів, чутливість до шуму та тип зв'язку між кубітами, а потім розподіляють кілька qVM на один квантовий чіп. Першим планувальником є «Планувальник місця», який функціонує подібно до «майстра тетрісу», розташовуючи кожну задачу у різні частини квантового процесору. У деяких випадках такий підхід підвищує точність обчислень, перенаправляючи чутливі задачі на найбільш оптимальні ділянки кристалу.

```
selection ← {}
occupied[r][c] ← false, ∀ 0 ≤ r < maxrow, 0 ≤ c < maxcol
for e in L do
    for v in all versions of e do
        for (r, c) in all positions do
            if regions in [r, r+v.width) × [c, c+v.height) are
              all unoccupied then
                add (v, r, c) into selection
                mark all regions used by v occupied
                if all regions occupied then
                    return selection
                break
return selection
```

Рисунок 1.17 - Алгоритм планувальника місця [43]

Алгоритмічно планувальник місця функціонує таким чином, що на вході алгоритм очікує список запусків L, а на виході він формує компіляцію кілець із зазначеними положеннями.

```
selection ← S
length[r][c] ← 0, reuse[r][c] ← 0,
  ∀ 0 ≤ r < maxrow, 0 ≤ c < maxcol
for v in S do
    for (r, c) in positions used by v do
        length[r][c] ← v.length
for e in L do
    for (r, c) in all positions do
        for v in all versions of e do
            if putting v at [r, r+v.width) × [c, c+v.height)
              does not increase max length and does not
              exceed reuse limit then
                add (v, r, c) into selection
                update length and reuse of all regions used
                  by v
                break
return selection
```

Рисунок 1.18 - Алгоритм планування часу [43]

Іншим планувальником є «Планувальник часу». Його завдання полягає у координації моментів виконання задач у qVM, які були розміщені планувальником місця. Він забезпечує ефективне використання часових відрізків квантового процесора, щоб уникнути конфліктів між задачами, враховуючи їх фізичні та логічні обмеження. Організація цього процесу здійснюється шляхом визначення послідовності та зсуву у часі під час виконання, а також впровадження окремих запусків для мінімізації завад, таких як міжкубітний перехресний шум або блокування каналів зчитування. На вході алгоритм отримує список запусків L та лист S компільованих кілець із їх положеннями, обраними планувальником місця. На виході формується лист обраних компільованих кілець із відповідними положеннями.

Завдяки такому підходу створюється ефект паралельності, що проявляється у скороченні середнього часу обчислення приблизно в 40 разів та збільшенні кількості задач, які виконуються за одиницю часу, до 10 разів. У результаті алгоритмічна задача, яка раніше займала дні, тепер виконується протягом кількох годин. Крім того, зазначений підхід забезпечує сумісність з існуючими інструментами програмування квантових комп’ютерів і не потребує використання спеціальних компіляторів або попередньо узгоджених задач, функціонуючи динамічно з доступними засобами.

Хоча проблема перехресної декогеренції залишається невирішеною, пріоритетом було визначено переправлення задач із шумних ділянок чипу для мінімізації впливу шуму на обчислення. Результати для 127-кубітної системи вважаються обнадійливими, проте на рівні тисяч кубітів залишаються відкритими питання щодо ефективності поділу ресурсів та впливу на якість остаточних обчислень.

1.10 Віртуалізація Gate

Схожим методом є концепт квантової віртуальної машини через віртуалізацію Gate [45]. Такий підхід орієнтований на вертикальне масштабування одного великого кола на обмежену кількість кубітів шляхом розрізання та фрагментації кола з розподіленим виконанням. У більшій мірі HyperQ реалізує горизонтальну віртуалізацію багатьох незалежних задач на одному фізичному квантовому процесорі.

Підхід віртуалізації Gate дозволяє застосовувати його всередині квантової віртуальної машини. Наприклад, коли користувачу необхідно виконати об'ємний алгоритм, маючи під контролем лише невелику підмножину кубітів. Гіпотетично цей підхід може бути інтегрований у HyperQ для розширення можливостей віртуального квантового процесору.

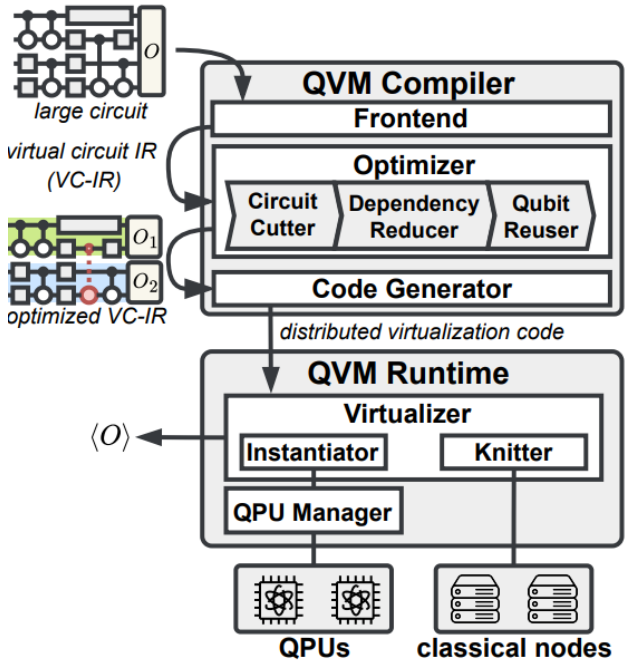

Рисунок 1.19 - Огляд архітектури віртуалізації Gate [45]

Ключовим абстрактним компонентом архітектури віртуалізації Gate є VC-IR - представлення віртуального кола. Воно містить як повноцінні, так і віртуалізовані Gate, а також інформацію про розбиття кола.

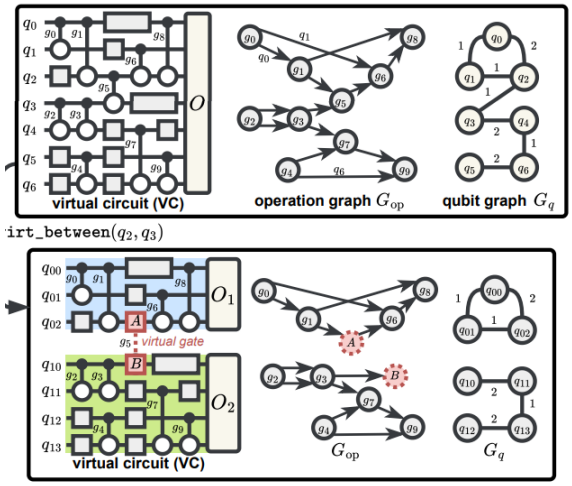

Рисунок 1.20 - Віртуальне кільце IR (VC-IR) [45]

Функціонально це виглядає наступним чином:

$$C : |0\rangle^{\otimes m} \rightarrow U_c \rightarrow U_c|0\rangle^{\otimes m} \quad , \tag{1.3}$$

де U_c - це унітарна матриця $2^m \times 2^m$.

Основною метою віртуалізації Gate є відтворення поведінки C навіть тоді, коли фізичний квантовий процесор має лише n<m кубітів. Розбиваючи коло на фрагменти з додатковими класичними реконструкціями, отримуємо наступне:

$$C \approx \otimes_{i=1}^{k} C_i, \tag{1.3}$$

де кожен C_i діє на підмножину кубітів.

Після формулювання VC-IR наступним кроком є виконання операцій модульної оптимізації шляхом розбиття кола C на фрагменти C_i з обмеженням на ширину (Circuit Cutter), коли кількість кубітів ≤ s, де s ≈ n фізичних кубітів. Далі здійснюється зменшення внутрішніх залежностей та кількості SWAP-операцій, що відповідають за непродовжувані зв'язки між Gate і підвищують частоту помилок (Dependency Reducer). Після цього проводиться повторне використання кубітів у межах фрагментів (Qubit Reuser), що дозволяє зменшити одночасно активну кількість кубітів за рахунок збільшення глибини реконструкції.

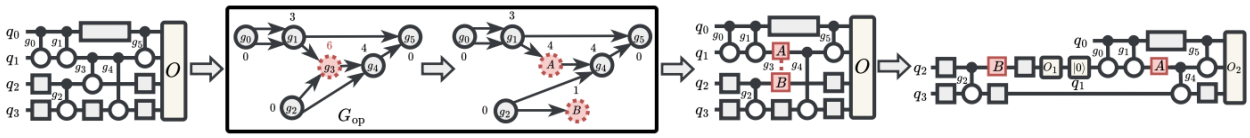

Рисунок 1.21 - Процес переходу Dependency Reducer у Qubit Reuser [45]

Фінальним кроком є трансляція фрагментів для використання на конкретних квантових процесорах або квантовій архітектурі (Code Generator).

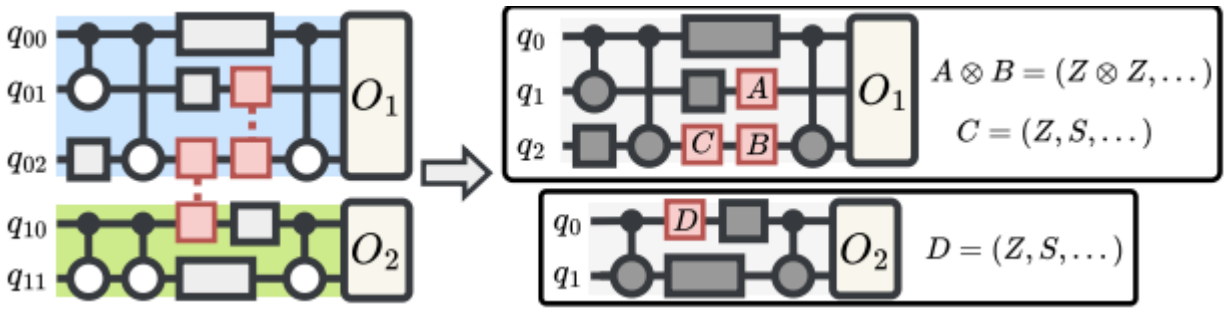

Рисунок 1.22 - Процес роботи Code Generator шляхом переходу з віртуального кільця у оптимізовані фрагменти [45]

Цей процес також може розглядатися як компіляція. Після його завершення здійснюється розподілене, або паралельне, виконання. На початковому етапі фрагменти C_i надсилаються на набір квантових процесорів, після чого результати вимірювань комбінуються за допомогою класичної обробки. Механізм «Virtualizer» встановлює всі можливі сценарії виконання, враховуючи віртуалізовані Gate, та забезпечує менеджеру квантового процесору керування ресурсами, чергами завдань і паралельним виконанням.

На основі попередньої інформації про архітектуру важливим елементом цього підходу є операція заміни двокубітного Gate на набір одно- або багатокубітних операцій із додатковою класичною обробкою після виконання.

1.11 Рішення від QMware

Компанія QMware, заснована у 2020 році, орієнтувалася на створення гібридних рішень як альтернативи класичним квантовим комп’ютерам з метою інтеграції класичних систем і квантових процесорів. Одним із таких рішень є поєднання класичних процесорів та графічних прискорювачів із квантовими процесорами (QPU). У рамках цього підходу було запропоновано продукт під назвою qognite, який є промисловою платформою для віртуалізації класично-квантових завдань, фактично виконуючи функції квантового гіпервізора з адаптацією до специфічних потреб квантових обчислень.

Рішення дозволяє підприємствам використовувати ці платформи у контейнеризованому або віртуалізованому середовищі для забезпечення ефективного контролю та адаптації до потенційних апаратних архітектур майбутнього. Простими словами, qognite надає користувачам досвід роботи з

віртуальними машинами, абстрагуючи специфіку QPU і надаючи керування через головний контролер, який одночасно управляє класичними і квантовими системами.

Архітектура платформи включає такі компоненти: гіпервізор, який керує підсистемами, контролер квантових юнітів, що використовує класичні контролери для керування кубітами, управління пам’яттю та забезпечення безпечної комунікації між різними частинами системи, контейнеризацію QPU для максимально паралельного запуску квантових обчислень, аналогічного до роботи класичних віртуальних машин, надання віртуального квантового прискорювача, абстрагованого від конкретного рішення QPU, а також підтримку інтеграції технологій на кшталт NVIDIA CUDA або MPI для забезпечення взаємодії класичних обчислювальних компонентів зі складними завданнями. Для захисту даних застосовується гібридний ключ, що комбінує класичну схему ECDHE із PQC KEM для забезпечення стійкості до потенційних майбутніх квантових атак.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ГІПЕРВІЗОРУ

Проєктування архітектури квантового гіпервізора є ключовим етапом створення моделі, що імітує принципи квантової віртуалізації, ізоляції процесів, логічної міграції станів та унітарного керування просторами імен.

Завданням моделі, розробленої у рамках цієї кваліфікаційної роботи, є демонстрація того, як гіпервізор може забезпечувати одночасне виконання ізольованих квантових задач, а також як ці задачі можуть взаємодіяти із класичним хостом для виконання спеціалізованих обчислень для підвищення безпеки інформаційних систем. Незважаючи на те, що апаратна реалізація повноцінної квантової віртуалізації сьогодні недоступна, запропонована модель дозволяє дослідити механізми, які можуть бути використані у подібних системах у майбутньому, коли квантові технології досягнуть необхідного рівня розвитку.

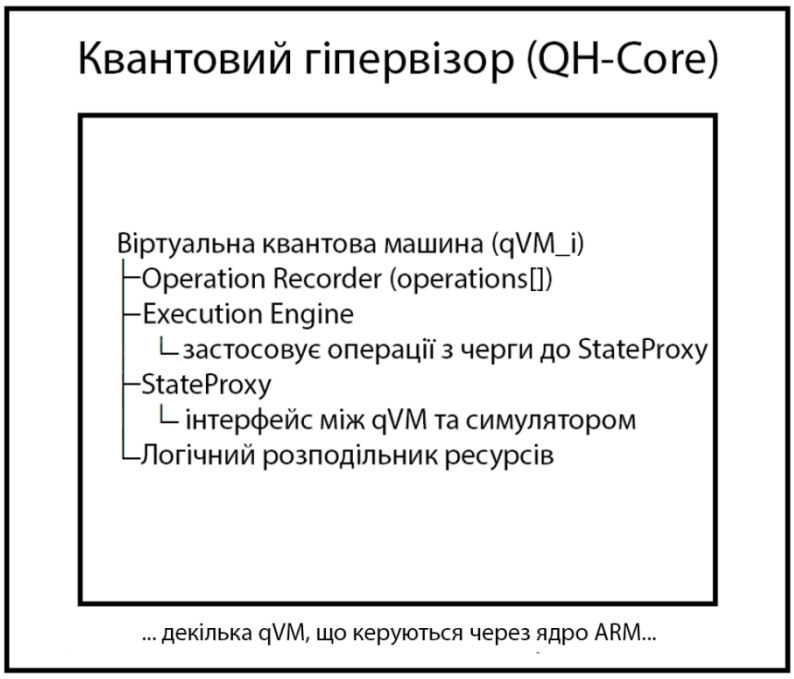

Рисунок 2.1 - Архітектура моделі шарів квантового гіпервізору

У вихідному вигляді квантовий гіпервізор структурується у вигляді набору логічних шарів, кожен з яких відповідає за певну частину процесу керування квантовими операціями. Загальна архітектура моделі представлена на рис. 2.1. Вона включає ядро гіпервізора (QH-Core), яке містить віртуальні квантові машини qVM_i, реєстратор операцій (Operation Recorder), механізм застосування гейтів (Execution Engine), інтерфейс взаємодії з симулятором, модуль StateProxy для зберігання унітарного сліду, а також логічний розподільник ресурсів, що визначає відображення фізичних кубітів на логічні регістри віртуальних середовищ. Кожен компонент виконує окреме функціональне завдання, але всі вони працюють у спільному оркестрованому середовищі, де ізоляція, доступ до кубітів та маршрутизація квантової заплутаності відбуваються згідно з політиками гіпервізора. Архітектура забезпечує можливість створення, зупинки, заморожування, міграції та повторного запуску віртуальних квантових середовищ.

2.1 Компонентна взаємодія

Подальший приклад внутрішньої взаємодії між компонентами гіпервізора дозволяє зрозуміти, як саме користувач або зовнішня система отримує доступ до механізмів керування квантовими середовищами. Взаємозв’язки між акторами системи та основними підсистемами моделі представлено на рис. 2.2.

Рисунок 2.2 - Usecase UML діаграма взаємодія акторів і компонентів системи

Діаграма демонструє, що класифікований набір акторів взаємодіє з системою на різних рівнях. Наприклад, «Адміністратор ARM-хосту» ініціює створення qVM, призначає правила ізоляції, аналізує журнали операцій та викликає механізми повторного запуску. У свою чергу, «Квантова віртуальна машина» виступає внутрішнім актором, що взаємодіє з Execution Engine, StateProxy і маршрутизатором квантової заплутаності. Саме через цю процедуру відбувається формування логічного шару, де кожна qVM отримує власний простір імен, у якому діють окремі правила маршрутизації міжкубітних операцій.

Необхідно врахувати, що діаграма UseCase має концептуальний характер, адже сучасні квантові пристрої не здатні забезпечити одночасну повноцінну ізольовану роботу декількох квантових просторів. Тому у програмній моделі ці

функції симулюються шляхом контролю доступу до gatelist, черг операцій та політик ізоляції через EntanglementRouter. У реальних системах ці механізми відповідали б апаратним контролерам, які керують фізичними кубітами та маршрутизацією квантових каналів.

Розглянемо детальніше логічну поведінку віртуального середовища протягом його життєвого циклу.

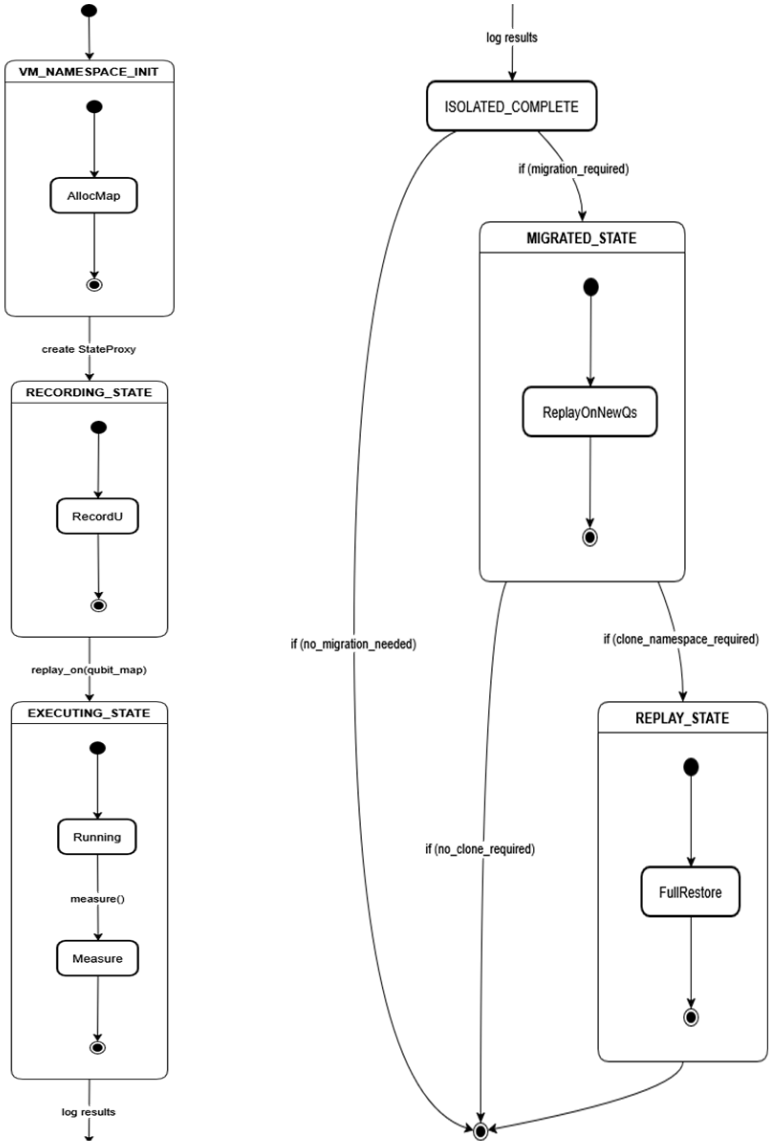

Рисунок 2.3 - UML діаграма стану життєвого циклу одного віртуального середовища

На діаграмі, представленій на рис. 2.3, показано життєвий цикл віртуального середовища. Життєвий цикл qVM починається зі стану VM_NAMESPACE_INIT. На цьому етапі виділяється початковий логічний простір імен, у якому ще немає жодної операції, але вже формується таблиця відповідності фізичних та логічних кубітів. Після ініціалізації середовище переходить у RECORDING_STATE, де гіпервізор приймає інструкції користувача або ARM-хосту, проте не виконує їх негайно. Операції лише записуються до журналу operation_log, що є критично важливим для відтворення, міграції та перевірки політик. Саме на цьому етапі модуль StateProxy фіксує унітарні перетворення без фактичного застосування гейтів, що дозволяє відтворювати стан у майбутньому без порушення заборони на клонування.

Коли всі інструкції підготовлено, qVM переходить у EXECUTION_STATE. У цьому стані Execution Engine починає послідовно застосовувати унітарні гейти до логічних кубітів. Маршрутизатор квантової заплутаності перевіряє, чи допускається відповідна топологія операцій між кубітами, а також чи не порушується межа namespace. Оскільки модель реалізує концепцію квантової ізоляції, будь-які спроби застосувати CX або SWAP між різними просторами імен блокуються. Після виконання всіх операцій відбувається вимірювання стану, результат якого зберігається у StateProxy для подальшого аналізу або передачі на ARM-хост.

Після виконання операцій гіпервізор переходить у стан ISOLATED_COMPLETE, де проводиться оцінка поточного середовища. На цьому етапі визначається, чи необхідно виконати міграцію логічного стану або повторне відтворення журналу. Якщо середовище залишається в межах своїх ресурсних обмежень, життєвий цикл qVM завершується. Однак якщо виникають умови, наприклад швидка декогеренція або потреба у переміщенні логічних кубітів, гіпервізор переходить у MIGRATED_STATE. У цьому стані виконується реконфігурація логічного простору: таблиці квантової заплутаності перебудовуються, простір імен переноситься, а нова конфігурація фіксується у StateProxy. Міграція є концептуальним механізмом, що імітує можливість

перенесення квантового стану без вимірювання. У реальних системах це може бути частково реалізовано через квантову телепортацію, коли технологія буде достатньо розвинена.

Якщо необхідно повторно запустити мігроване середовище, гіпервізор переходить у REPLAY_STATE. На цьому етапі StateProxy взаємодіє з Operation Recorder, відтворюючи журнал операцій у новому просторі імен. Це дозволяє перевірити поведінку qVM на попередніх інструкціях, що є корисним для аналізу програм або шкідливих процесів. Після завершення відтворення гіпервізор повертається у фінальний стан і передає результати ARM-хосту для аналізу.

2.2 ARM компонент

З огляду на те, що модель використовується для демонстрації інтеграції класичних та квантових компонентів у кібербезпеці, до архітектури було додано компонент ARM-хосту. Його функціональність схематично зображена на рис. 2.4. ARM-хост виконує роль класичного керуючого рівня, що ініціює середовища, передає інструкції на гіпервізор, аналізує журнали операцій, а також отримує результати квантових обчислень для поведінкового аналізу можливого шкідливого ПЗ. ARM-хост також може виконувати планування задач, перенаправляти дані між qVM і класичними алгоритмами та вести журнал активності для оцінки аномалій. Взаємодія між класичним хостом і квантовими модулями у моделі реалізується через логічний обмін повідомленнями, що імітує апаратну взаємодію між квантовим контролером та центральним процесором.

Рисунок 2.4 - Оновлена архітектура моделі з додаванням шару ARM хосту

В архітектурі також продемонстровано логічний обмін повідомленнями як демонстрацію «апаратної» взаємодії між компонентами.

Підсумовуючи основні завдання, можна скласти наступну таблицю:

Таблиця 2.1 - Перелік основних реалізацій моделі

Параметр	Процес роботи	Завдання
1	2	3
Ізоляція	Заборона CX/SWAP між namespace`ами	Концептуальне рішення маршрутизатору квантової заплутаності для ізоляції процесів без десинхронізації.

Кінець таблиці 2.1

1	2	3
Віртуалізація без копіювання	Відтворення стану без копіювання через StateProxy	Рішення унітарної віртуалізації без порушення принципу no-cloning заборони
Унітарні операції між квантовими кільцями	Міграція стану без колапсу	Гіпотетичний механізм безвимірювальної передачі стану між фізичними регіонами
Аналіз шкідливого ПЗ	Паралельні квантові sandbox середовища	Приклад моделі використання квантової віртуалізації для поведінкового аналізу шкідливого ПЗ

Розроблена архітектура демонструє поєднання таких концепцій як квантова ізоляція, унітарна віртуалізація, міграція логічних станів та інтеграція з класичним рівнем керування. Таблиця основних реалізацій включає механізми заборони міжпросторових гейтів, реконструкцію станів без порушення закону про неможливість клонування, логічну міграцію між регіонами та можливість використання квантових середовищ у задачах аналізу шкідливого ПЗ. Запропонована модель репрезентує теоретичну основу для можливих майбутніх систем, у яких квантові обчислення та класичні механізми кібербезпеки працюватимуть у єдиному віртуалізованому просторі.

3 МОДЕЛЮВАННЯ ВІРТУАЛІЗАЦІЇ У КВАНТОВИХ СИСТЕМАХ

Квантовий гіпервізор здійснюватиме розподіл кубітів за прикладом HyperQ, вести журнал унітарних операцій (StateProxy) та гарантувати ізоляцію за допомогою маршрутизатора Entanglement.

Програмна реалізація моделі буде виконана на мові програмування Python з використанням бібліотеки Qiskit, яка забезпечить необхідні інструменти для побудови, симуляції та запуску квантових схем. Для моделювання квантових обчислень буде застосована бібліотека AER, що дозволяє враховувати вплив шумів, характерних для реальних квантових систем.

У даній моделі квантове кільце логічно створюється як впорядкована послідовність кубітів у межах кільця, що дозволяє відтворювати поведінку квантової системи та реалізувати принципи віртуалізації кубітів.

3.1 Класи у моделі

3.1.1 Клас гіпервізору

Квантовий гіпервізор виконує роль центрального диспетчера віртуальних квантових машин, створюючи ізольовані простори імен через метод create_namespace та забезпечуючи, щоб кожен простір мав власний набір кубітів, не пов’язаний із жодним іншим. Крім того, він веде глобальну таблицю виділених кубітів (self.allocations) і надає кожній qVM непересічний діапазон фізичних індексів, що частково відтворює підхід, застосований у HyperQ.

```
class QuantumHypervisor:
    def __init__(self, n_physical_qubits: int, router: Optional[SecureEntanglementRouter] = None):
        self.n = n_physical_qubits
        self.free_qubits = set(range(n_physical_qubits))
        self.namespaces: Dict[str, List[int]] = {}
        self.state_proxies: Dict[str, StateProxy] = {}
        self.telemetry: Dict[str, Dict] = {}
        self.security_log: List[Dict] = []
        self.router = router if router is not None else SecureEntanglementRouter({})

    def create_namespace(self, name: str, size: int) -> bool: ...

    def destroy_namespace(self, name: str): ...

    def get_ns_qubits(self, name: str) -> List[int]: ...

    def record_op(self, namespace: str, op_name: str, qubits: List[int], params: Tuple = ()): ...

    def build_circuit_for_namespace(self, namespace: str, apply_noise: bool = True) -> QuantumCircuit: ...

    def run_joint(self, namespaces: List[str], extra_ops=None, shots=DEFAULT_SHOTS) -> Dict: ...

    def unitary_migration(self, namespace: str, target_phys: List[int]) -> bool: ...

    def snapshot_telemetry(self) -> Dict: ...
```

Рисунок 3.1 - Основні методи класу гіпервізору

Гіпервізор має власні внутрішні структури даних. Для ініціалізації він отримує такі параметри:

- кількість фізичних кубітів n;

- маршрутизатор зв’язків квантової заплутаності.

Усередині класу створюються: множина вільних кубітів free_qubits, простори імен namespaces, об’єкти збереження стану (state_proxies) для кожного простору імен, телеметрія telemetry, журнал безпеки security_log, а також інтегрований маршрутизатор квантової заплутаності, який відповідає за підтримання топології та контролює можливість зв’язування або роз’єднання просторів імен.

Основні методи класу включають:

- create_namespace - виділяє необхідну кількість кубітів для нового простору імен та повертає False у разі нестачі ресурсів. Метод також оновлює структури маршрутизатора.

- destroy_namespace - повністю знищує логічний простір, звільняє пов’язані ресурси, видаляє відповідний StateProxy і телеметрію, а також оновлює дані маршрутизатора.

- get_ns_qubits - повертає список фізичних індексів кубітів, виділених конкретному простору імен.

- record_op - записує логічну операцію у відповідний StateProxy. На цьому етапі логічні кубіти залишаються незалежними від фізичного розподілу до моменту виконання або повторного запуску.

Квантове кільце формується за допомогою build_circuit_for_namespace, який відтворює журнал операцій на виділених кубітах. У цьому методі також реалізовано модель контролю декогеренції. Стохастична симуляція шуму реалізується шляхом вставлення операцій reset з імовірністю 8% на рівні окремого простору імен. Такий підхід забезпечує більш реалістичну поведінку, моделюючи локальні помилки (аналог фазового шуму каналу Паулі), але не призводить до припинення виконання.

Метод run_joint створює глобальну схему для кількох просторів імен, збираючи їхні кубіти в один простір виконання. Для кожного простору імен відтворюється відповідний StateProxy. За необхідності можуть бути додані контрольовані крос-операції (extra_ops). На цьому рівні моделюється менший шум - 2%, що відображає взаємодію окремих «квантових контейнерів» у межах одного апаратного кільця.

Метод unitary_migration виконує безвимірювальну міграцію стану з одного набору фізичних кубітів на інший, оновлюючи топологічну карту маршрутизатора. У модельному виконанні існує 5% імовірність імітації втрати цілісності логічного

стану під час міграції, що є критичною подією: процес переривається, повертається False, а інцидент реєструється у журналі безпеки.

Метод snapshot_telemetry повертає глибоку копію телеметрії та використовується як механізм моніторингу стану системи.

Застосований підхід моделює процес запобігання неконтрольованому змішуванню станів між віртуальними середовищами та відтворює концепцію ізоляції на рівні квантового кільця.

3.1.2 Клас маршрутизатору квантової заплутаності

Маршрутизатор квантової заплутаності SecureEntanglementRouter блокує двокубітні операції між кубітами, що належать різним просторам імен, забезпечуючи відсутність крос-заплутаності. Це гарантується умовою:

$$\left[H_{namespace_i}, H_{namespace_j}\right] = 0, \quad i \neq j, \tag{3.1}$$

що інтерпретується як неможливість впливу еволюції одного логічного простору на інший.

Архітектурно SecureEntanglementRouter виконує роль ізоляційного шару гіпервізора, контролюючи та обмежуючи двокубітні операції між віртуальними квантовими машинами. Він моделює неможливість прямої взаємодії між незалежними квантовими просторами імен та за своїми функціями нагадує MAC-політику контролю доступу.

```
class SecureEntanglementRouter:
    def __init__(self, namespace_map: Dict[str, List[int]]):
        self.ns_map = namespace_map
        self.rev: Dict[int, str] = {}
        for ns, qlist in namespace_map.items():
            for q in qlist:
                self.rev[q] = ns
        self.policy = set()
        self.audit_log: List[Dict] = []
        self.blocks = 0

    def refresh_map(self, namespace_map: Dict[str, List[int]]):…

    def allow_link(self, ns1: str, ns2: str):…

    def revoke_link(self, ns1: str, ns2: str):…

    def can_interact(self, q1: int, q2: int) -> bool:…

    def log_block(self, q1: int, q2: int, origin_ns: str, op_name: str):…


EntanglementRouter = SecureEntanglementRouter
```

Рисунок 3.2 - Основні методи класу маршрутизатору квантової заплутаності

Під час ініціалізації клас очікує карту просторів імен namespace_map, де кожному простору відповідає список фізичних кубітів, виділених цьому простору. Основними атрибутами класу є:

- ns_map – актуальна карта відповідності «простір імен → фізичні кубіти»;

- rev – реверсивне представлення, яке дозволяє за індексом кубіта визначити, до якого простору він належить;

- policy – множина пар просторів імен, для яких дозволена взаємодія;

- audit_log – журнал аудиту, у який фіксуються дозволені, заблоковані та невдалі операції;

- лічильник заблокованих міжпросторових операцій, що дозволяє здійснювати моніторинг порушень політики.

Метод refresh_map оновлює карту просторів імен та її реверсивне представлення у випадку зміни розподілу кубітів. Використовується після операцій створення або видалення простору імен у гіпервізорі.

Метод allow_link встановлює пару просторів імен, між якими дозволено безпечну взаємодію. Кожна така дія фіксується в audit_log. Реверсивним методом є revoke_link, який видаляє дозвіл на взаємодію між відповідною парою просторів.

Метод can_interact перевіряє можливість створення зв’язку між двома кубітами, використовуючи реверсивну карту rev, та визначає, чи належать ці кубіти дозволеним просторам імен.

Для фіксації заборонених або небезпечних операцій використовується механізм log_block, що записує такі події у журнал аудиту та збільшує лічильник блокувань.

3.1.3 Класс StateProxy

Основним завданням цього класу є збереження та відтворення логічних операцій у квантовому просторі імен, а його метою - забезпечення абстракції логічних кубітів від фізичних ресурсів та можливості безпечного повторного виконання операцій на виділених кубітах. Це дозволяє гіпервізору керувати декількома просторами імен, контролювати політику взаємодій між кубітами та моделювати потенційні обмеження для підвищення реалістичності системи.

Зважаючи на обмеження, накладені теоремою no-cloning, у якості механізму віртуалізації було вирішено зберігати послідовність унітарних операцій у StateProxy. Ця структура не є копією стану, а радше моделлю логічної пам’яті.

Функціонально StateProxy може бути представлений рівнянням:

$$\psi\rangle = U_n U_{\{n-1\}} \cdots U_1 \,|0\rangle\,, \tag{3.2}$$

де U_i - це послідовність записаних унітарних операторів, що визначають логічну еволюцію системи, а $|0\rangle$ - початковий стан.

```
class StateProxy:
    def __init__(self):
        self.operations = []

    def record(self, op_name: str, qubits: List[int], params: Tuple = ()):
        self.operations.append((op_name, list(qubits), tuple(params)))

    def replay_on(self, qc: QuantumCircuit, qubit_map: List[int], router=None, ns_name: Optional[str] = None):
        for op_name, qlist, params in self.operations:
            phys_qubits = [qubit_map[q] for q in qlist]
            if op_name in ("cx", "swap") and router is not None and ns_name is not None:
                q1, q2 = phys_qubits[0], phys_qubits[1]
                if not router.can_interact(q1, q2):
                    router.log_block(q1, q2, ns_name, op_name)
                    qc.barrier()
                    continue
```

Рисунок 3.3 - Основні методи класу StateProxy

На початку клас створює список operations, у якому зберігаються записані квантові операції у вигляді кортежів. Основним методом є record, що фіксує логічні операції, зокрема назву операції op_name, список логічних кубітів qubits, до яких вона застосовується, а також додаткові параметри, наприклад кути для ротаційних гейтів. У цій моделі застосовуються гейти Адамара («h»), контрольований NOT («cx»), а також ротаційні гейти «rx», «ry» та «rz» із параметром кута.

Метод replay_on призначено для відтворення записаних операцій на контурі квантового кільця, з використанням відображення qubit_map для зіставлення логічних індексів із фізичними. Переданий маршрутизатор виконує перевірку двокубітних операцій та фіксує їх. Записана послідовність може бути відтворена в іншій топології кубітів завдяки використанню qubit_map. Вона є аналогом журналу квантових операцій (U-gate sequences), у тому числі містить інформацію про

результати вимірювань, що дозволяє відтворювати еволюцію стану (змінюючи вектор $|\psi\rangle$ під дією унітарних операторів U і шумових каналів E) без його дублювання. Таким чином, замість клонування стан зберігається у вигляді функціонального журналу, що є аналогом журналу транзакцій у класичних базах даних.

Метод replay_on створює відповідний квантовий гейт для кожної записаної операції та може бути формально описаний як:

$$\psi_{proxy}\rangle = \prod_{i=1}^{n} U_i \, |0\rangle, \tag{3.3}$$

де кожен оператор U_i послідовно застосовується до відповідних кубітів, визначених за допомогою qubit_map.

Завдяки цьому забезпечується можливість відтворення будь-якої послідовності операцій незалежно від конкретного фізичного розподілу кубітів.

3.2. Поєднання з ARM

Клас ARMHost моделює ARM-контролер, якому підпорядковуються квантові віртуальні машини, та слугує прикладом інтеграції класичних обчислювальних систем із квантовими.

```
class ARMHost:
    def __init__(self, hypervisor: QuantumHypervisor, router: SecureEntanglementRouter):
        self.hv = hypervisor
        self.router = router
        self.security_log: List[Dict] = []

    def submit_quantum_job(self, namespace: str, extra_ops=None, shots=DEFAULT_SHOTS) -> Dict: ...

    def submit_joint_job(self, namespaces: List[str], entangle_ops: List[Tuple[str, int, str, int, str]] = None,

    def record_logical_op(self, namespace: str, op_name: str, logical_qubits: List[int], params: Tuple = ()):
        self.hv.record_op(namespace, op_name, logical_qubits, params)

    def allow_policy(self, ns1: str, ns2: str): ...

    def revoke_policy(self, ns1: str, ns2: str): ...
```

Рисунок 3.4 - Основні методи класу ARM хосту

Під час ініціалізації клас очікує отримати екземпляри квантового гіпервізора та маршрутизатора квантової заплутаності. Усередині класу створюються відповідні об’єкти - гіпервізор hv, маршрутизатор router, а також журнал безпеки security_log, у якому фіксуються події порушення політик, аномальні взаємодії та будь-які інциденти на рівні логічної взаємодії середовищ.

Основними методами класу є:

- submit_quantum_job - відправляє квантове завдання на виконання у межах одного простору імен, повертаючи результати обчислень та відповідну телеметрію.

- submit_joint_job - виконує об’єднані квантові завдання між кількома просторами імен, включаючи крос-namespace операції за умови дозволів, визначених маршрутизатором.

- record_logical_op - здійснює запис логічної операції у відповідне віртуальне середовище, забезпечуючи накопичення операційної історії для подальшого формування схем або міграцій станів.

- allow_policy - призначений для створення правил взаємодії між двома просторами імен через оновлення таблиці дозволів у маршрутизаторі. Подія фіксується у security_log із зазначенням часу та відповідної пари просторів.

- revoke_policy - реверсивний метод, що скасовує дозвіл на взаємодію між просторами імен, також фіксуючи подію у журналі безпеки.

Завдяки цим методам клас виконує роль проміжного керуючого шару між логікою віртуальних квантових машин та політиками маршрутизації квантової заплутаності. Він не лише забезпечує формальну передачу завдань, але й підтримує цілісність ізольованих обчислювальних просторів, контролює їхню взаємодію та гарантує відтворюваність квантових трас. У сукупності ці механізми формують логічний контур безпеки для керування крос-середовищними квантовими операціями, що є критичним для побудови масштабованих гіпервізорних систем.

3.3 Програмна послідовність роботи коду демонстрації моделі

Логічна послідовність включає також низку умовних механізмів, які застосовуються виключно у межах моделювання. До таких механізмів належать: симульована карта відповідності логічних і фізичних кубітів, міграційний шар, система заборони операцій між просторами імен, а також можливість керування квантовими середовищами з боку ARM-хосту, що функціонує аналогічно до класичного аналізатора у sandbox-системах. Запровадження цих компонентів дає змогу оцінити послідовність взаємодій між гіпервізором, ізольованими квантовими середовищами та класичним керуючим шаром. Крім того, модель дозволяє визначити типові програмні структури, які можуть бути потрібні для побудови гіпотетичних рішень у сфері безпечної та контрольованої роботи з квантовими ресурсами.

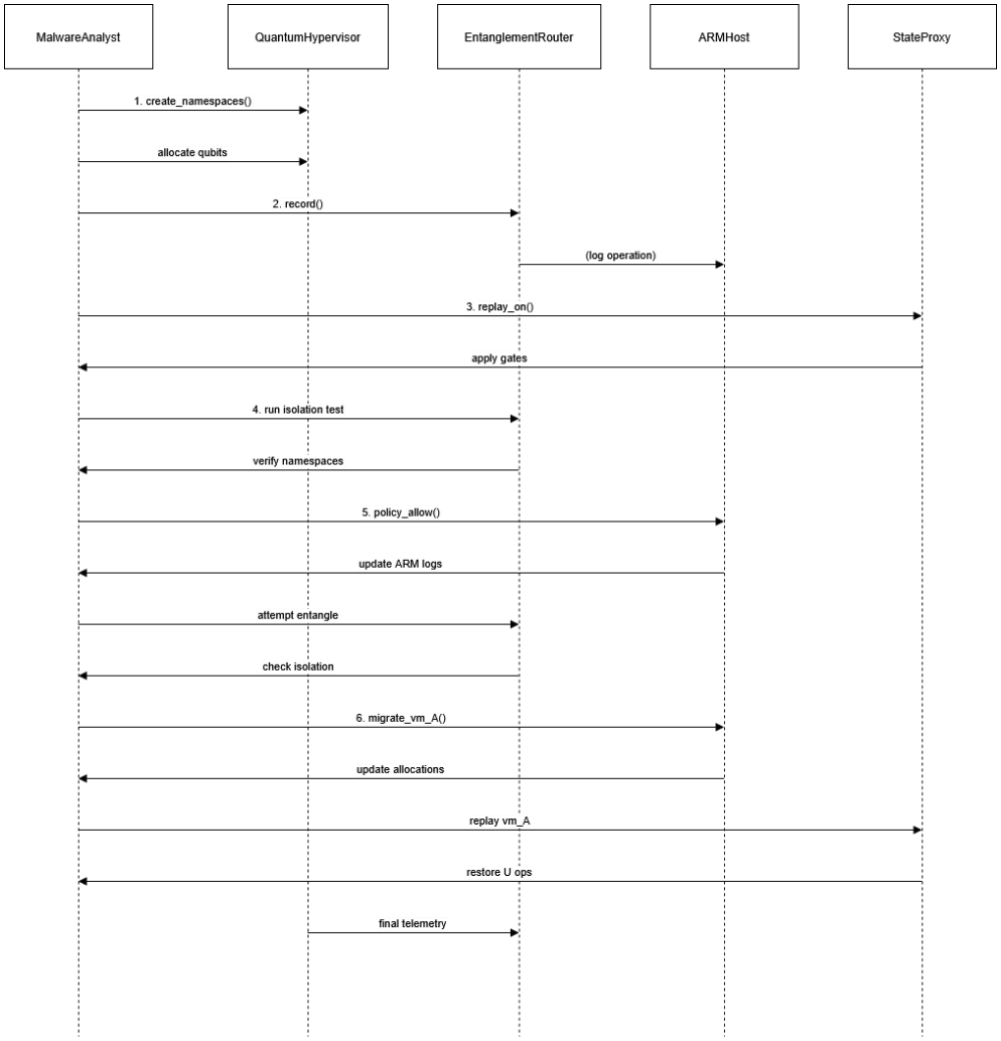

Рисунок 3.5 - UML діаграма послідовності

Діаграма послідовності, подана на рис. 3.5, є найбільш деталізованим представленням повного циклу керування квантовим середовищем.

Логічна послідовність процесу має такий вигляд. На початковому етапі гіпервізор створює зберігач ідентифікаційних номерів для середовищ, структуру журналів, політик та таблицю майбутньої алокації. Далі ініціалізується базова архітектура керування за допомогою виклику hv = QuantumHypervisor(n_physical_qubits=12), якому у цьому випадку передається 12 фізичних кубітів. Паралельно ініціалізується ARM-хост arm = ARMHost(hv, router),

що отримує екземпляри квантового гіпервізора та маршрутизатора квантової заплутаності.

Наступним кроком гіпервізор створює простори імен шляхом виклику hv.create_namespace(). Це призводить до виділення квантових ресурсів, які відіграють роль ізольованих віртуальних квантових машин, призначених для моделювання аналізу шкідливого програмного забезпечення. На цьому етапі резервується необхідна кількість кубітів, створюється StateProxy та елемент телеметрії, після чого формується оновлена карта простору імен. Після ініціалізації ARM-хост здійснює запис логічних операцій до простору імен через ланцюг викликів ARMHost → QuantumHypervisor → StateProxy. Унітарні операції зберігаються в StateProxy у вигляді кортежів виду (op_name, logical_qubits, params).

Така організація дозволяє моделювати ізоляцію між віртуальними квантовими середовищами та забезпечувати формування результатів вимірювань у форматі розрахункових даних і телеметрії.

На наступному етапі відбувається моделювання крос-взаємодій квантової заплутаності. ARM-хост формує запит arm.submit_quantum_job() на зв’язування кубітів, однак виконання забороняється політикою безпеки через відсутність відповідного дозволу у маршрутизаторі. Після цього ARM-хост викликає arm.allow_policy(), створюючи дозвіл на взаємодію між обраними просторами імен. Після оновлення політик виконується повторна спроба операції квантової заплутаності з використанням оператора hcx.

Подальшим кроком є моделювання унітарної міграції віртуальної машини на новий набір фізичних кубітів та оновлення карти топологічних зв’язків. Після завершення міграції повторно запускається arm.submit_quantum_job() з метою перевірки коректності роботи політик доступу в оновленому стані системи.

Для демонстрації механізму відтворення стану створюється нове віртуальне середовище, яке отримує копію стану попередньої віртуальної машини. Після цього

запускається виконання обчислень у відокремленому контексті, що дає змогу уникнути будь-якого потенційного порушення ізоляції первинного середовища.

Завершальні етапи демонстрації включають процедуру очищення, що виконується через виклик hv.destroy_namespace(). Цей крок моделює видалення віртуальної машини та вивільнення відповідних квантових ресурсів, зокрема фізичних кубітів, які були задіяні в попередніх обчисленнях.

Також важливо зазначити, що в межах симуляційної моделі використовується єдине фізичне кільце кубітів, тоді як логічно система функціонує на кількох рівнях. На фізичному рівні всі унітарні операції проходять через базове кільце кубітів, яке розглядається як фактичний носій станів. На логічному рівні структура qubit_map містить власну систему індексів, що трактуються як «власні» логічні кубіти, хоча фізично вони не існують - кожен логічний індекс лише посилається на відповідне фізичне відображення. Окремий міграційний рівень забезпечує перенесення логічної послідовності між різними qubit_map-конфігураціями. Він відповідає за коректне перевідображення логічних індексів на інше фізичне кільце без копіювання реального квантового стану, що моделює унітарну міграцію в межах гіпервізора.

Після запуску симуляції отримано результати, які відображають послідовність виконання операцій, застосування політик доступу, логіку міграційного шару та відтворення стану в окремому віртуальному середовищі. Ці результати дозволяють комплексно оцінити роботу моделі та підтверджують коректність реалізованих механізмів у контексті гіпотетичних систем безпечної віртуалізації квантових ресурсів.

```
(venv) PS D:\ZNTU\5й\Диплом_2\app> python ./main.py
=== Demo: quantum hypervisor simulation ===
Namespaces created: True True True
Allocated map: {'malware_vm_A': [0, 1, 2], 'malware_vm_B': [3, 4, 5], 'observer_vm': [6, 7]}
Router map: {0: 'malware_vm_A', 1: 'malware_vm_A', 2: 'malware_vm_A', 3: 'malware_vm_B', 4: 'malware_vm_B', 5: 'malware_vm_B', 6: 'observer_vm', 7: 'observer_vm'}

-- Isolated run VM A --
VM A counts (top 5): [('111', 521), ('000', 503)] telemetry: {'alloc_time': 1762694821.6769667, 'noise_events': 0, 'migrations': 0, 'security_blocks': 0}

-- Isolated run VM B --
VM B counts (top 5): [('001000', 267), ('000000', 591), ('011000', 51), ('010000', 115)] telemetry: {'alloc_time': 1762694821.6769667, 'noise_events': 0, 'migratio
ns': 0, 'security_blocks': 0}

-- Attempt joint entangle between VM A and VM B (should be blocked) --
Joint result (blocked) keys: dict_keys(['counts', 'telemetry', 'router_audit'])
Router audit entries (recent): [{'time': 1762694821.9539661, 'event': 'blocked_op', 'op': 'hcx', 'qubits': (0, 3), 'namespaces': ('malware_vm_A', 'malware_vm_B'),
'origin_ns': 'malware_vm_A'}]

[POLICY] allowed A <-> B

-- Attempt joint entangle between VM A and VM B (should be allowed now) --
Joint counts (top 10): [('001110', 78), ('000000', 142), ('001000', 83), ('001111', 142), ('000110', 146), ('000001', 57), ('001001', 159), ('010000', 33), ('01100
1', 29), ('011111', 28)]
Router audit entries (recent): [{'time': 1762694821.9539661, 'event': 'blocked_op', 'op': 'hcx', 'qubits': (0, 3), 'namespaces': ('malware_vm_A', 'malware_vm_B'),
'origin_ns': 'malware_vm_A'}, {'time': 1762694822.129965, 'event': 'allow', 'pair': ('malware_vm_A', 'malware_vm_B')}]

Security log (ARM): [{'time': 1762694821.9539661, 'event': 'blocked_entangle', 'qubits': (0, 3), 'op': 'hcx'}, {'time': 1762694822.129965, 'event': 'policy_allow',
 'pair': ('malware_vm_A', 'malware_vm_B')}]
HV telemetry snapshot: {'malware_vm_A': {'alloc_time': 1762694821.6769667, 'noise_events': 0, 'migrations': 0, 'security_blocks': 1}, 'malware_vm_B': {'alloc_time'
: 1762694821.6769667, 'noise_events': 1, 'migrations': 0, 'security_blocks': 1}, 'observer_vm': {'alloc_time': 1762694821.6769667, 'noise_events': 0, 'migrations':
 0, 'security_blocks': 0}}

-- Attempt unitary migration of VM A to new physical qubits [9,10,11] --
Migration success: True
New allocations: {'malware_vm_A': [9, 10, 11], 'malware_vm_B': [3, 4, 5], 'observer_vm': [6, 7]}
Router mapping after migration: {9: 'malware_vm_A', 10: 'malware_vm_A', 11: 'malware_vm_A', 3: 'malware_vm_B', 4: 'malware_vm_B', 5: 'malware_vm_B', 6: 'observer_v
m', 7: 'observer_vm'}

-- Joint run A+B after migration --
Joint result keys: dict_keys(['counts', 'telemetry', 'router_audit'])
Router audit tail: [{'time': 1762694821.9539661, 'event': 'blocked_op', 'op': 'hcx', 'qubits': (0, 3), 'namespaces': ('malware_vm_A', 'malware_vm_B'), 'origin_ns':
 'malware_vm_A'}, {'time': 1762694822.129965, 'event': 'allow', 'pair': ('malware_vm_A', 'malware_vm_B')}]

-- Replay VM_A proxy into a fresh namespace (replay_vm) --
Replay counts (top 5): [('000000000', 531), ('100000110', 493)]

Demo complete.
```

Рисунок 3.7 - Результат виводу програми

Аналізуючи результати моделювання, подані на рис. 3.7, можна побачити, що система послідовно створює три логічні середовища - malware_vm_A, malware_vm_B та observer_vm, - після чого коректно ініціалізує їх, що підтверджується повідомленням “Namespaces created: True True True”. Наступним етапом формується карта кубітів, де чітко відображено межі кожного простору імен відповідно до закріплених за ним ресурсів.

Таблиця маршрутизатора демонструє відповідність між фізичними кубітами та їх логічною прив’язкою до визначених середовищ. Це дозволяє підтвердити факт ізоляції між усіма створеними віртуальними середовищами, оскільки кожному з них відповідає власний сегмент фізичного кільця.

На першому етапі виконання («Isolated run VM») видно, що malware_vm_A безперешкодно здійснює власні унітарні операції, фіксуючи результати у вигляді бітових патернів та відповідної телеметрії. У межах моделі квантової віртуалізації цей етап демонструє відсутність небажаних взаємодій або перехресного впливу на інші середовища. Аналогічний результат отримує і malware_vm_B, що підтверджує незалежність квантових робочих просторів на ранніх фазах дослідження.

Далі модель переходить до спроби встановлення зв’язку квантової заплутаності між malware_vm_A та malware_vm_B. Як і очікувалось, система повертає результат “Blocked”, оскільки на момент виконання політика безпеки забороняє будь-які крос-namespace операції. У журналі маршрутизатора зафіксовано відповідну подію у вигляді запису {event: 'blocked_op'}, що відображає спробу виконання контрольованої операції типу HCX та вказує на кубіти й простори імен, залучені до забороненої взаємодії. Таким чином підтверджується коректність механізмів контролю ізоляції.

Після цього система фіксує повідомлення “[POLICY] allowed A <-> B”, яке відповідає команді дозволу взаємодії між двома середовищами. Повторна спроба виконання крос-операції завершується успішно, що відображено виводом “Joint counts”. Модель демонструє змішані квантові стани з властивостями квантової заплутаності між кубітами різних просторів, а ARMHost реєструє в журналі як попереднє блокування, так і подальше надання дозволу.

На наступному етапі проводиться моделювання унітарної міграції середовища malware_vm_A на новий сегмент фізичних кубітів. Система коректно оновлює карту розподілу ресурсів і виводить повідомлення “Migration success: True”. Маршрутизатор фіксує перев’язування логічних кубітів із новими фізичними індексами та від’єднання попередніх, що демонструє стабільність механізму перенесення середовища без втрати інформації та без порушення чинних політик доступу.

Оскільки модель імітує роботу квантової системи з гарантованою ізоляцією, результати вважаються коректними лише за умови відсутності неконтрольованої інтерференції. Повернені counts відображають різницю у розподілах вимірювань, що підтверджує незалежність середовищ. Результат подається у вигляді рядка бітів, який репрезентує фінальний квантовий стан, та числа повторень цього стану у вибірці. Наприклад, у випадку виводу “'001110', 78” це означає, що у 78 випадках з 1000 вимірювань система колапсувала у стан |0⟩|0⟩|1⟩|1⟩|1⟩|1⟩|0⟩, що у межах симуляційної моделі відображає ймовірнісну природу вимірювання.

Після міграції виконується спільний запуск malware_vm_A та malware_vm_B, що завершується успішно. Журнали безпеки не містять нових блокувань, що підтверджує коректність копіювання логічного стану та відсутність конфліктів. У журналі маршрутизатора («Router audit tail») також відображено попередню історію подій, що демонструє стабільність поведінки системи після перенесення середовища.

На завершальному етапі демонструється механізм відтворення стану шляхом створення нового середовища replay_vm, у яке переноситься логічний стан malware_vm_A. Це дозволяє обійти обмеження теореми про заборону клонування за рахунок відтворення послідовності логічних операцій. Результати Replay counts підтверджують відповідність логічних повторень початковій конфігурації.

Загалом отримана модель може бути інтегрована до структур типу SoC, де вона здатна слугувати інструментом для аналізу шкідливого програмного забезпечення у квантових або гібридних системах як додатковий механізм для підвищення безпеки інформаційних систем. Повний перелік реалізованого коду наведено у ДОДАТКУ А.

ВИСНОВКИ

Під час створення дипломного проєкту було проведено комплексний аналіз механізмів віртуалізації на всіх рівнях класичних систем - від апаратної підтримки ізоляції та розподілу ресурсів до організації контейнеризації, гіпервізорів та характеру їхньої взаємодії з ядром операційної системи. Особлива увага була приділена тому, як класичні системи забезпечують безпечне виконання процесів, контроль за ресурсами, перемикання контекстів та дотримання політик між незалежними середовищами. Паралельно з цим було проведено огляд сучасних квантових технологій, включно з фізичними обмеженнями квантових процесорів, топологією зв’язків, механізмами квантової заплутаності, принципом no-cloning, а також чинними проблемами масштабування та надійності квантових обчислень. На основі цих даних було сформовано вимоги та ключові принципи до побудови моделі квантової віртуалізації. У ході роботи було розроблено багатошарову модель квантового гіпервізора, що включає шар керування ресурсами, шар ізоляції, підсистему відтворення стану (StateProxy), маршрутизатор квантової заплутаності, механізм логічної міграції та взаємодію з умовним ARM-хостом для моделювання поведінкового аналізу шкідливого ПЗ. Створена програмна демонстрація показує, як квантові середовища можуть створюватися, виконувати ізольовані унітарні операції, проходити через процеси запису стану, міграції між логічними просторами та відтворення без порушення принципу заборони копіювання. Запропонована модель також демонструє можливість сценарної інтеграції класичної системи управління (ARM-host), що відповідає за планування, політики доступу та аналіз результатів, подібно до поведінки sandbox-платформ у класичній безпеці. У результаті виконаної роботи було сформовано модельну архітектуру квантового гіпервізора, яка відтворює ключові компоненти, необхідні для гіпотетичного керування квантовими ресурсами в умовах багатокористувацького або багатозадачного середовища.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Kangralkar N. Full, Para, and Hardware-Assisted Virtualization Compared [Електронний ресурс] / Narendra Kangralkar, Kevin Gilmore // Baeldung. - Режим доступу: https://www.baeldung.com/cs/virtualization-techniques-compared (дата звернення: 04.11.2025). - Назва з екрана.

2. Contributors to Wikimedia projects. Hypervisor - Wikipedia [Електронний ресурс] / Contributors to Wikimedia projects // Wikipedia, the free encyclopedia. - Режим доступу: https://en.wikipedia.org/wiki/Hypervisor (дата звернення: 04.11.2025). - Назва з екрана.

3. Bigelow S. J. What's the difference between Type 1 vs. Type 2 hypervisor? | TechTarget [Електронний ресурс] / Stephen J. Bigelow // Search IT Operations. - Режим доступу: https://www.techtarget.com/searchitoperations/tip/Whats-the-difference-between-Type-1-vs-Type-2-hypervisor (дата звернення: 15.11.2025). - Назва з екрана.

4. Contributors to Wikimedia projects. x86 virtualization - Wikipedia [Електронний ресурс] / Contributors to Wikimedia projects // Wikipedia, the free encyclopedia. - Режим доступу: https://en.wikipedia.org/wiki/X86_virtualization (дата звернення: 04.11.2025). - Назва з екрана.

5. Virtualization in Cloud Computing and Types - GeeksforGeeks [Електронний ресурс] // GeeksforGeeks. - Режим доступу: https://www.geeksforgeeks.org/virtualization-cloud-computing-types/ (дата звернення: 15.11.2025). - Назва з екрана.

6. Contributors to Wikimedia projects. Protection ring - Wikipedia [Електронний ресурс] / Contributors to Wikimedia projects // Wikipedia, the free encyclopedia. - Режим доступу: https://en.wikipedia.org/wiki/Protection_ring (дата звернення: 04.11.2025). - Назва з екрана.

7. Děcký M. Advanced Operating Systems [Електронний ресурс] / Martin Děcký // Department of Distributed and Dependable Systems | D3S. - Режим доступу: https://d3s.mff.cuni.cz/files/teaching/nswi161/2023_24/04_virtualization.pdf (дата звернення: 17.11.2025). - Назва з екрана.

8. Fang Y. Introduction to the ARMv8 Virtualization System | openEuler [Електронний ресурс] / Ying Fang // openEuler | OS for Digital Infrastructure. - Режим доступу: https://www.openeuler.org/en/blog/yorifang/2020-10-24-arm-virtualization-overview.html (дата звернення: 04.11.2025). - Назва з екрана.

9. The Generic Interrupt Controller [Електронний ресурс] // Arm Developer. - Режим доступу: https://developer.arm.com/documentation/102909/0100/The-Generic-Interrupt-Controller (дата звернення: 13.11.2025). - Назва з екрана.

10. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3 [Електронний ресурс] // intel. - Режим доступу: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf (дата звернення: 04.11.2025). - Назва з екрана.

11. Secure Virtual Machine Architecture Reference Manual [Електронний ресурс] // 0x04. - Режим доступу: https://www.0x04.net/doc/amd/33047.pdf (дата звернення: 06.11.2025). - Назва з екрана.

12. Rayanfam Blog. - Режим доступу: https://rayanfam.com/assets/files/VMCS.pdf (дата звернення: 16.11.2025). - Назва з екрана.

13. medievalghoul. AMD-V Hypervisor Development - A Brief Explanation [Електронний ресурс] / medievalghoul // Private Group Of Back Engineers. - Режим доступу: https://blog.back.engineering/04/08/2022/ (дата звернення: 16.11.2025). - Назва з екрана.

14. The Definitive KVM (Kernel-based Virtual Machine) API Documentation - The Linux Kernel documentation [Електронний ресурс] // The Linux Kernel documentation -

The Linux Kernel documentation. - Режим доступу: https://docs.kernel.org/virt/kvm/api.html (дата звернення: 04.11.2025). - Назва з екрана.

15. Murray J. Using GPUs with Virtual Machines on vSphere - Part 3: Installing the NVIDIA Virtual GPU Technology [Електронний ресурс] / Justin Murray // VMware Cloud Foundation (VCF) Blog. - Режим доступу: https://blogs.vmware.com/cloud-foundation/2018/09/17/using-gpus-with-virtual-machines-on-vsphere-part-3-installing-the-nvidia-grid-technology/ (дата звернення: 24.11.2025). - Назва з екрана.

16. Bigelow S. J. GPU virtualization evolves with new chip types on the horizon | TechTarget [Електронний ресурс] / Stephen J. Bigelow // Search IT Operations. - Режим доступу: https://www.techtarget.com/searchitoperations/tip/GPU-virtualization-evolves-with-new-chip-types-on-the-horizon (дата звернення: 04.11.2025). - Назва з екрана.

17. Momtazpour M. Fig 5 - available from: The Journal of Supercomputing [Електронний ресурс] / Mahmoud Momtazpour, Ahmad Siavashi // Researchgate. - Режим доступу: https://www.researchgate.net/figure/GPU-execution-stack-and-virtualization-methods_fig4_328317992 (дата звернення: 04.11.2025). - Назва з екрана.

18. Trifonov D. Host Setup for Qemu KVM GPU Passthrough with VFIO on Linux [Електронний ресурс] / Dmitro Trifonov // Rent Datacenter GPUs for AI & ML • Cloudrift. - Режим доступу: https://www.cloudrift.ai/blog/host-setup-for-qemu-kvm-gpu-passthrough-with-vfio-on-linux (дата звернення: 04.11.2025). - Назва з екрана.

19. Namespaces(7) - Linux manual page [Електронний ресурс] // Michael Kerrisk - man7.org. - Режим доступу: https://www.man7.org/linux/man-pages/man7/namespaces.7.html (дата звернення: 04.11.2025). - Назва з екрана.

20. Capabilities(7) - Linux manual page [Електронний ресурс] // Michael Kerrisk - man7.org. - Режим доступу: https://man7.org/linux/man-pages/man7/capabilities.7.html (дата звернення: 04.11.2025). - Назва з екрана.

21. Containerd/docs/historical/design/architecture.png at main · containerd/containerd [Електронний ресурс] // GitHub. - Режим доступу:

https://github.com/containerd/containerd/blob/main/docs/historical/design/architecture.png (дата звернення: 04.11.2025). - Назва з екрана.

22. Containerd/docs/historical/cri/performance.png at main · containerd/containerd [Електронний ресурс] // GitHub. - Режим доступу: https://github.com/containerd/containerd/blob/main/docs/historical/cri/performance.png (дата звернення: 24.11.2025). - Назва з екрана.

23. The Linux Kernel/Syscalls - Wikibooks, open books for an open world [Електронний ресурс] // Wikibooks. - Режим доступу: https://en.wikibooks.org/wiki/The_Linux_Kernel/Syscalls (дата звернення: 24.11.2025). - Назва з екрана.

24. Containerd/docs/historical/cri/containerd.png at main · containerd/containerd [Електронний ресурс] // GitHub. - Режим доступу: https://github.com/containerd/containerd/blob/main/docs/historical/cri/containerd.png (дата звернення: 04.11.2025). - Назва з екрана.

25. Cyber Writes Team. Interactive malware sandbox - free file analysis, live malware hunting & threat intelligence [Електронний ресурс] / Cyber Writes Team // Cyber Security News. - Режим доступу: https://cybersecuritynews.com/interactive-malware-sandbox-for-business/ (дата звернення: 04.11.2025). - Назва з екрана.

26. Contributors to Wikimedia projects. Virtual machine introspection - Wikipedia [Електронний ресурс] / Contributors to Wikimedia projects // Wikipedia, the free encyclopedia. - Режим доступу: https://en.wikipedia.org/wiki/Virtual_machine_introspection (дата звернення: 24.11.2025). - Назва з екрана.

27. Sanchez-Palencia L. Quantum simulation: from basic principles to applications [Електронний ресурс] / Laurent Sanchez-Palencia // arXiv.org e-Print archive. - Режим доступу: https://arxiv.org/pdf/1812.01110 (дата звернення: 04.11.2025). - Назва з екрана.

28. M. Alsing P. Feynman’s “Simulating Physics with Computers” [Електронний ресурс] / Paul M. Alsing, Carlo Cafaro, Stefano Mancini // arXiv.org e-Print archive. - Режим доступу: https://arxiv.org/html/2405.03366v1 (дата звернення: 04.11.2025). - Назва з екрана.

29. Karar A. Microsoft's majorana 1: a new era in quantum computing and a new state of matter! [Електронний ресурс] / Abe Karar. - Режим доступу: https://www.linkedin.com/pulse/microsofts-majorana-1-new-era-quantum-computing-state-abe-karar-kszjf (дата звернення: 04.11.2025). - Назва з екрана.

30. Quantum computer | google quantum AI [Електронний ресурс] // Google Quantum AI. - Режим доступу: https://quantumai.google/quantumcomputer (дата звернення: 04.11.2025). - Назва з екрана.

31. Dahale G. R. Greetings from the IBM spring challenge: map of 127 qubits [Електронний ресурс], 2023 / Gopal Ramesh Dahale // Medium. - Режим доступу: https://dahalegopal27.medium.com/greetings-from-the-ibm-spring-challenge-map-of-127-qubits-ce08817176a7 (дата звернення: 04.11.2025). - Назва з екрана.

32. Schneider J. What is quantum computing? | IBM [Електронний ресурс] / Josh Schneider, Ian Smalley // IBM. - Режим доступу: https://www.ibm.com/think/topics/quantum-computing (дата звернення: 04.11.2025). - Назва з екрана.

33. Contributors to Wikimedia projects. No-cloning theorem - Wikipedia [Електронний ресурс] / Contributors to Wikimedia projects // Wikipedia, the free encyclopedia. – Режим доступу: https://en.wikipedia.org/wiki/No-cloning_theorem (дата звернення: 12.12.2025). – Назва з екрана.

34. Weigert S. No-Cloning theorem [Електронний ресурс] / Stefan Weigert. - Режим доступу: https://www.researchgate.net/profile/Stefan-Weigert/publication/240994528_No-Cloning_Theorem/links/575abf6808aed884620d8f5b/No-Cloning-Theorem.pdf (дата звернення: 17.11.2025). - Назва з екрана.

35. Учасники проєктів Вікімедіа. Кубіт - Вікіпедія [Електронний ресурс] / Учасники проєктів Вікімедіа // Вікіпедія. - Режим доступу: https://uk.wikipedia.org/wiki/Кубіт (дата звернення: 04.11.2025). - Назва з екрана.

36. Krantz P. A quantum engineer's guide to superconducting qubits [Електронний ресурс] / P. Krantz, W. D. Oliver // AIP Publishing. - Режим доступу: https://pubs.aip.org/aip/apr/article/6/2/021318/570326/A-quantum-engineer-s-guide-to-superconducting (дата звернення: 04.11.2025). - Назва з екрана.

37. Generalized toric codes on twisted tori for quantum error correction [Електронний ресурс] / Zijian Liang [та ін.] // APS. - Режим доступу: https://journals.aps.org/prxquantum/abstract/10.1103/rmy6-9n89 (дата звернення: 04.11.2025). - Назва з екрана.

38. Dynamical decoupling for superconducting qubits: a performance survey [Електронний ресурс] / Nic Ezzell [та ін.] // arxiv. - Режим доступу: https://arxiv.org/pdf/2207.03670 (дата звернення: 04.11.2025). - Назва з екрана.

39. Control and mitigation of microwave crosstalk e ect with superconducting qubits [Електронний ресурс] / Ruixia Wang [та ін.] // arxiv. - Режим доступу: https://arxiv.org/pdf/2207.08416 (дата звернення: 04.11.2025). - Назва з екрана.

40. Overhead in quantum circuits with time-multiplexed qubit control [Електронний ресурс] / Marvin Richter [та ін.] // arxiv. - Режим доступу: https://arxiv.org/pdf/2508.20752 (дата звернення: 04.11.2025). - Назва з екрана.

41. Boosting computational power through spatial multiplexing in quantum reservoir computing [Електронний ресурс] / Kohei Nakajima [та ін.] // arxiv. - Режим доступу: https://arxiv.org/pdf/1803.04574 (дата звернення: 04.11.2025). - Назва з екрана.

42. Ahamed Mohammad I. Meta-optimization of resources on quantum computers - PMC [Електронний ресурс] / Ijaz Ahamed Mohammad, Matej Pivoluska, Martin Plesch // PMC Home. - Режим доступу: https://pmc.ncbi.nlm.nih.gov/articles/PMC11070419/ (дата звернення: 04.11.2025). - Назва з екрана.

43. Quantum virtual machines [Електронний ресурс] / Runzhou Tao [та ін.] // usenix. - Режим доступу: https://www.usenix.org/system/files/osdi25-tao.pdf (дата звернення: 04.11.2025). - Назва з екрана.

44. HyperQ - virtual machines for quantum computers [Електронний ресурс] / Aaron Bordeaux [та ін.] // Columbia Engineering. - Режим доступу: https://www.engineering.columbia.edu/sites/default/files/2024-09/Aaron%20Bordeaux%20SURE%20Poster%20-%20Aaron%20Nathan%20Bordeaux.pdf (дата звернення: 04.11.2025). - Назва з екрана.

45. Tornow N. Scaling QuantumComputationsviaGateVirtualization [Електронний ресурс] / Nathaniel Tornow, Emmanouil Giortamis, Pramod Bhatotia // arxiv. - Режим доступу: https://arxiv.org/pdf/2406.18410 (дата звернення: 04.11.2025). - Назва з екрана.

ДОДАТОК А

Код програми

```
import random
import time
from copy import deepcopy
from typing import List, Dict, Tuple, Optional
from qiskit import QuantumCircuit, transpile
from qiskit_aer import Aer
from qiskit.quantum_info import Statevector

# ---------------------------
# Helpers / Config
# ---------------------------
SIM_BACKEND = Aer.get_backend("aer_simulator")
DEFAULT_SHOTS = 1024


# ---------------------------
# StateProxy: запис унітарних операцій
# ---------------------------
class StateProxy:
    def __init__(self):
        self.operations = []

    def record(self, op_name: str, qubits: List[int], params:
Tuple = ()):
        self.operations.append((op_name, list(qubits),
tuple(params)))

    def replay_on(self, qc: QuantumCircuit, qubit_map:
List[int], router=None, ns_name: Optional[str] = None):
        for op_name, qlist, params in self.operations:
            phys_qubits = [qubit_map[q] for q in qlist]
            if op_name in ("cx", "swap") and router is not None
and ns_name is not None:
                q1, q2 = phys_qubits[0], phys_qubits[1]
                if not router.can_interact(q1, q2):
                    router.log_block(q1, q2, ns_name, op_name)
```

```
                    qc.barrier()
                    continue
                if op_name == "h":
                    for q in phys_qubits:
                        qc.h(q)
                elif op_name == "cx":
                    qc.cx(phys_qubits[0], phys_qubits[1])
                elif op_name == "rx":
                    qc.rx(params[0], phys_qubits[0])
                elif op_name == "ry":
                    qc.ry(params[0], phys_qubits[0])
                elif op_name == "rz":
                    qc.rz(params[0], phys_qubits[0])
                else:
                    pass

# ---------------------------
# SecureEntanglementRouter: policy-based контроль взаємодій
# ---------------------------
class SecureEntanglementRouter:
    def __init__(self, namespace_map: Dict[str, List[int]]):
        self.ns_map = namespace_map
        self.rev: Dict[int, str] = {}
        for ns, qlist in namespace_map.items():
            for q in qlist:
                self.rev[q] = ns
        self.policy = set()
        self.audit_log: List[Dict] = []
        self.blocks = 0

    def refresh_map(self, namespace_map: Dict[str, List[int]]):
        self.ns_map = namespace_map
        self.rev = {}
        for ns, qlist in namespace_map.items():
            for q in qlist:
                self.rev[q] = ns

    def allow_link(self, ns1: str, ns2: str):
        key = tuple(sorted([ns1, ns2]))
        self.policy.add(key)
```

```
        self.audit_log.append({"time": time.time(), "event":
"allow", "pair": key})

    def revoke_link(self, ns1: str, ns2: str):
        key = tuple(sorted([ns1, ns2]))
        if key in self.policy:
            self.policy.remove(key)
            self.audit_log.append({"time": time.time(), "event":
"revoke", "pair": key})

    def can_interact(self, q1: int, q2: int) -> bool:
        ns1 = self.rev.get(q1, None)
        ns2 = self.rev.get(q2, None)
        if ns1 is not None and ns1 == ns2:
            return True
        if ns1 is None or ns2 is None:
            return False
        return tuple(sorted([ns1, ns2])) in self.policy

    def log_block(self, q1: int, q2: int, origin_ns: str,
op_name: str):
        ns1 = self.rev.get(q1, None)
        ns2 = self.rev.get(q2, None)
        rec = {
            "time": time.time(),
            "event": "blocked_op",
            "op": op_name,
            "qubits": (q1, q2),
            "namespaces": (ns1, ns2),
            "origin_ns": origin_ns
        }
        self.audit_log.append(rec)
        self.blocks += 1


EntanglementRouter = SecureEntanglementRouter


# --------------------------
# QuantumHypervisor
# --------------------------
```

```
class QuantumHypervisor:
    def __init__(self, n_physical_qubits: int, router:
Optional[SecureEntanglementRouter] = None):
        self.n = n_physical_qubits
        self.free_qubits = set(range(n_physical_qubits))
        self.namespaces: Dict[str, List[int]] = {}
        self.state_proxies: Dict[str, StateProxy] = {}
        self.telemetry: Dict[str, Dict] = {}
        self.security_log: List[Dict] = []
        self.router = router if router is not None else
SecureEntanglementRouter({})

    def create_namespace(self, name: str, size: int) -> bool:
        if name in self.namespaces:
            raise ValueError("namespace already exists")
        if len(self.free_qubits) < size:
            return False
        allocated = []
        for _ in range(size):
            allocated.append(self.free_qubits.pop())
        self.namespaces[name] = allocated
        self.state_proxies[name] = StateProxy()
        self.telemetry[name] = {"alloc_time": time.time(),
"noise_events": 0, "migrations": 0, "security_blocks": 0}
        self.router.refresh_map(self.namespaces)
        return True

    def destroy_namespace(self, name: str):
        qs = self.namespaces.pop(name, [])
        for q in qs:
            self.free_qubits.add(q)
        self.state_proxies.pop(name, None)
        self.telemetry.pop(name, None)
        self.router.refresh_map(self.namespaces)

    def get_ns_qubits(self, name: str) -> List[int]:
        return list(self.namespaces.get(name, []))

    def record_op(self, namespace: str, op_name: str, qubits:
List[int], params: Tuple = ()):
        sp = self.state_proxies[namespace]
```

```
        sp.record(op_name, qubits, params)

    def build_circuit_for_namespace(self, namespace: str,
apply_noise: bool = True) -> QuantumCircuit:
        phys = self.get_ns_qubits(namespace)
        if not phys:
            raise ValueError("namespace not found or empty")
        maxq = max(phys) + 1
        qc = QuantumCircuit(maxq)
        sp = self.state_proxies[namespace]
        sp.replay_on(qc, phys, router=self.router,
ns_name=namespace)
        if apply_noise:
            if random.random() < 0.08:  # 8% event
                victim = random.choice(phys)
                qc.reset(victim)
                self.telemetry[namespace]["noise_events"] += 1
        return qc

    def run_joint(self, namespaces: List[str], extra_ops=None,
shots=DEFAULT_SHOTS) -> Dict:
        used_qubits = set()
        for ns in namespaces:
            used_qubits.update(self.get_ns_qubits(ns))
        if not used_qubits:
            return {"error": "no_qubits"}

        maxq = max(used_qubits) + 1
        qc = QuantumCircuit(maxq)
        for ns in namespaces:
            phys = self.get_ns_qubits(ns)
            sp = self.state_proxies[ns]
            sp.replay_on(qc, phys, router=self.router,
ns_name=ns)

        if extra_ops:
            extra_ops(qc, self.namespaces, self.router, self)

        for ns in namespaces:
            if random.random() < 0.02:
                phys = self.get_ns_qubits(ns)
```

```
                victim = random.choice(phys)
                qc.reset(victim)
                self.telemetry[ns]["noise_events"] += 1

        qc.measure_all()
        tq = transpile(qc, SIM_BACKEND)
        try:
            job = SIM_BACKEND.run(tq, shots=shots)
            result = job.result()
            counts = result.get_counts()
            tel = {ns: deepcopy(self.telemetry.get(ns, {})) for
ns in namespaces}
            audit = deepcopy(self.router.audit_log)
            return {"counts": counts, "telemetry": tel,
"router_audit": audit}
        except Exception as e:
            return {"error": str(e)}

    def unitary_migration(self, namespace: str, target_phys:
List[int]) -> bool:

        src = self.get_ns_qubits(namespace)
        if not src:
            return False
        if len(src) != len(target_phys):
            return False
        maxq = max(max(src), max(target_phys)) + 1
        qc = QuantumCircuit(maxq)
        for a, b in zip(src, target_phys):
            if a == b:
                continue
            qc.swap(a, b)
        tq = transpile(qc, SIM_BACKEND)
        noise_roll = random.random()
        if noise_roll < 0.05:
            self.telemetry[namespace]["migrations"] += 1
            self.security_log.append({"time": time.time(),
"event": "migration_failed", "namespace": namespace})
            return False
        for q in src:
            self.free_qubits.add(q)
```

```
        for q in target_phys:
            if q in self.free_qubits:
                self.free_qubits.remove(q)
        self.namespaces[namespace] = list(target_phys)
        self.router.refresh_map(self.namespaces)
        return True

    def snapshot_telemetry(self) -> Dict:
        return deepcopy(self.telemetry)


# ---------------------------
# ARMHost: координує аналіз та політику
# ---------------------------
class ARMHost:
    def __init__(self, hypervisor: QuantumHypervisor, router:
SecureEntanglementRouter):
        self.hv = hypervisor
        self.router = router
        self.security_log: List[Dict] = []

    def submit_quantum_job(self, namespace: str, extra_ops=None,
shots=DEFAULT_SHOTS) -> Dict:

        phys = self.hv.get_ns_qubits(namespace)
        if not phys:
            return {"error": "no_namespace"}

        qc = self.hv.build_circuit_for_namespace(namespace,
apply_noise=True)
        if extra_ops:
            extra_ops(qc, phys)

        qc.measure_all()
        tqc = transpile(qc, SIM_BACKEND)
        try:
            job = SIM_BACKEND.run(tqc, shots=shots)
            result = job.result()
            counts = result.get_counts()
            return {"counts": counts, "telemetry":
deepcopy(self.hv.telemetry[namespace])}
```

```
        except Exception as e:
            return {"error": str(e)}

    def submit_joint_job(self, namespaces: List[str],
entangle_ops: List[Tuple[str, int, str, int, str]] = None,
shots=DEFAULT_SHOTS) -> Dict:
        def extra_ops(qc: QuantumCircuit, ns_map: Dict[str,
List[int]], router: SecureEntanglementRouter, hv:
QuantumHypervisor):
            if not entangle_ops:
                return
            for (ns1, lq1, ns2, lq2, op_name) in entangle_ops:
                phys1_list = ns_map.get(ns1)
                phys2_list = ns_map.get(ns2)
                if phys1_list is None or phys2_list is None:
                    self.security_log.append({"time":
time.time(), "event": "invalid_namespace", "details": (ns1,
ns2)})
                    continue
                q1 = phys1_list[lq1]
                q2 = phys2_list[lq2]
                if not router.can_interact(q1, q2):
                    router.log_block(q1, q2, ns1, op_name)
                    if ns1 in hv.telemetry:
                        hv.telemetry[ns1]["security_blocks"] +=
1
                    if ns2 in hv.telemetry:
                        hv.telemetry[ns2]["security_blocks"] +=
1
                    self.security_log.append({"time":
time.time(), "event": "blocked_entangle", "qubits": (q1, q2),
"op": op_name})
                    continue
                if op_name == "cx":
                    qc.cx(q1, q2)
                elif op_name == "swap":
                    qc.swap(q1, q2)
                elif op_name == "hcx":
                    qc.h(q1); qc.cx(q1, q2)
                else:
```

```
                    self.security_log.append({"time":
time.time(), "event": "unsupported_entangle", "op": op_name})
        return self.hv.run_joint(namespaces,
extra_ops=extra_ops, shots=shots)

    def record_logical_op(self, namespace: str, op_name: str,
logical_qubits: List[int], params: Tuple = ()):
        self.hv.record_op(namespace, op_name, logical_qubits,
params)

    def allow_policy(self, ns1: str, ns2: str):
        self.router.allow_link(ns1, ns2)
        self.security_log.append({"time": time.time(), "event":
"policy_allow", "pair": (ns1, ns2)})

    def revoke_policy(self, ns1: str, ns2: str):
        self.router.revoke_link(ns1, ns2)
        self.security_log.append({"time": time.time(), "event":
"policy_revoke", "pair": (ns1, ns2)})


# ---------------------------
# Demo: приклад workflow
# ---------------------------
def demo_workflow():
    print("=== QH-HMA Demo: Secure HyperQ-like quantum
hypervisor simulation ===")
    router = SecureEntanglementRouter({})
    hv = QuantumHypervisor(n_physical_qubits=12, router=router)
    arm = ARMHost(hv, router)
    ok1 = hv.create_namespace("malware_vm_A", size=3)
    ok2 = hv.create_namespace("malware_vm_B", size=3)
    ok3 = hv.create_namespace("observer_vm", size=2)
    print("Namespaces created:", ok1, ok2, ok3)
    print("Allocated map:", hv.namespaces)
    print("Router map:", router.rev)
    arm.record_logical_op("malware_vm_A", "h", [0])
    arm.record_logical_op("malware_vm_A", "cx", [0, 1])
    arm.record_logical_op("malware_vm_A", "cx", [1, 2])
    arm.record_logical_op("malware_vm_B", "rx", [0],
params=(1.2,))
```

```
    arm.record_logical_op("malware_vm_B", "ry", [1],
params=(0.8,))
    arm.record_logical_op("malware_vm_B", "rz", [2],
params=(2.1,))
    print("\n-- Isolated run VM A --")
    resA = arm.submit_quantum_job("malware_vm_A")
    print("VM A counts (top 5):", list(resA.get("counts",
{}).items())[:5], "telemetry:", resA.get("telemetry"))
    print("\n-- Isolated run VM B --")
    resB = arm.submit_quantum_job("malware_vm_B")
    print("VM B counts (top 5):", list(resB.get("counts",
{}).items())[:5], "telemetry:", resB.get("telemetry"))
    ent_ops = [("malware_vm_A", 0, "malware_vm_B", 0, "hcx")]
    print("\n-- Attempt joint entangle between VM A and VM B
(should be blocked) --")
    joint_res_blocked = arm.submit_joint_job(["malware_vm_A",
"malware_vm_B"], entangle_ops=ent_ops)
    print("Joint result (blocked) keys:",
joint_res_blocked.keys())
    print("Router audit entries (recent):", router.audit_log[-
3:])
    arm.allow_policy("malware_vm_A", "malware_vm_B")
    print("\n[POLICY] allowed A <-> B")
    print("\n-- Attempt joint entangle between VM A and VM B
(should be allowed now) --")
    joint_res_allowed = arm.submit_joint_job(["malware_vm_A",
"malware_vm_B"], entangle_ops=ent_ops)
    print("Joint counts (top 10):",
list(joint_res_allowed.get("counts", {}).items())[:10])
    print("Router audit entries (recent):", router.audit_log[-
5:])
    print("\nSecurity log (ARM):", arm.security_log[-5:])
    print("HV telemetry snapshot:", hv.snapshot_telemetry())
    print("\n-- Attempt unitary migration of VM A to new
physical qubits [9,10,11] --")
    target = [9, 10, 11]
    success = hv.unitary_migration("malware_vm_A", target)
    print("Migration success:", success)
    print("New allocations:", hv.namespaces)
    print("Router mapping after migration:", router.rev)
    print("\n-- Joint run A+B after migration --")
```

```
    joint_res_after_mig = arm.submit_joint_job(["malware_vm_A",
"malware_vm_B"], entangle_ops=ent_ops)
    print("Joint result keys:", joint_res_after_mig.keys())
    print("Router audit tail:", router.audit_log[-5:])
    print("\n-- Replay VM_A proxy into a fresh namespace
(replay_vm) --")
    hv.create_namespace("replay_vm", size=3)
    hv.state_proxies["replay_vm"] =
deepcopy(hv.state_proxies["malware_vm_A"])
    resR = arm.submit_quantum_job("replay_vm")
    print("Replay counts (top 5):", list(resR.get("counts",
{}).items())[:5])
    hv.destroy_namespace("malware_vm_A")
    hv.destroy_namespace("malware_vm_B")
    hv.destroy_namespace("replay_vm")
    hv.destroy_namespace("observer_vm")
    print("\nDemo complete.")


if __name__ == "__main__":
    demo_workflow()
```