ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
МІНІСТЕРСТВА ОСВІТИ І НАУКИ УКРАЇНИ

Кваліфікаційна наукова
праця на правах рукопису

РЕГІДА ПАВЛО ГЕННАДІЙОВИЧ

УДК 004.3:004.75:004.49

ДИСЕРТАЦІЯ

МЕТОДИ ТА ЗАСОБИ ОРГАНІЗАЦІЇ РОЗПОДІЛЕНИХ СИСТЕМ ВИЯВЛЕННЯ ІНФІКОВАНИХ ВИКОНУВАНИХ ПРОГРАМ, СТІЙКИХ ДО ЕМУЛЯЦІЇ В СЕРЕДОВИЩІ ВИКОНАННЯ

123 Комп'ютерна інженерія
(шифр і назва спеціальності)
12 Інформаційні технології
(галузь знань)

Подається на здобуття наукового ступеня доктора філософії
Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело


П.Г. Регіда
(підпис)

Науковий керівник: Савенко Олег Станіславович, доктор технічних наук, професор

Хмельницький – 2025

АНОТАЦІЯ

Регіда П. Г. Методи та засоби організації розподілених систем виявлення інфікованих виконуваних програм, стійких до емуляції в середовищі виконання. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії з галузі знань 12 Інформаційні технології за спеціальністю 123 Комп'ютерна інженерія. – Хмельницький національний університет, Хмельницький 2025.

Вирішення задачі з покращення ефективності функціонування грід-обчислювальних систем для виявлення інфікованих програм, що використовують методи уникнення від виявлення, за рахунок організації процесу розподіленого аналізу поведінки виконання із використанням програмних переривань на автономних обчислювальних елементах, є актуальною науковою задачею.

У дисертації здійснено аналіз архітектури сучасних розподілених систем обчислень, методів їх організації функціонування, методів виявлення інфікованих програм, зокрема таких, що використовують методи протидії емуляції та обфускації для уникнення виявлення. В роботі представлено архітектуру грід-обчислювальної системи для забезпечення розподіленого виявлення зловмисної поведінки в інфікованих програмах, метод виявлення зловмисної поведінки інфікованої програми на основі аналізу поведінки виконання в ізольованому модифікованому середовищі за допомогою програмних переривань, метод організації грід-обчислювальної системи, що використовує автономні обчислювальні елементи в динамічному середовищі функціонування, застосування оцінки довіри обчислювальних елементів на основі модифікованої моделі довіри для зниження кількості повторних перевірок виконаних задач на основі ролей та елементів нечіткої логіки, а також розроблено відповідну розподілену систему, сформовані експерименти і та проведені дослідження із розробленою системою.

Об'єктом дослідження є процес організації розподіленої системи для виявлення зловмисної поведінки в інфікованих програмах, які використовують методи уникнення виявлення.

Предметом дослідження є методи організації розподілених систем із обчислювальними елементами, що мають рівень автономії для виявлення зловмисного прояву в інфікованих програмах.

Метою дисертаційного дослідження є покращення ефективності функціонування грід-обчислювальних систем із автономними обчислювальними елементами для виявлення зловмисної передачі управління головним потоком в інфікованих програмах, що базується на концепції динамічного стану емулювання середовища відтворення програмних засобів.

Наукова новизна отриманих результатів полягає в наступному:

1) вперше розроблено модель централізованих грід-обчислювальних систем, в якій враховано вимоги до залучення автономних та гетерогенних обчислювальних елементів для забезпечення виконання задач із перевіркою на коректність в динамічному середовищі виконання, і яка дає змогу залучити під'єднані обчислювальні елементи для аналізу поведінки виконання інфікованих програм, забезпечуючи розподілений процес виявлення зловмисного прояву в інфікованих програмах;

2) розроблено новий метод синтезу засобів формування шаблонів поведінки інфікованих програм, який на відміну від відомих відрізняється залученням пісочниці для їх виконання у наборі створюваних модифікованих ізольованих середовищах за допомогою виконання програмних переривань та базового емулятора із визначеним набором реалізованих низькорівневих інструкцій, що дає змогу отримувати з них шаблони поведінки на множинах станів емульованих центральних процесорів з метою виявлення зловмисної поведінки з урахуванням особливостей методів уникнення від виявлення, які реалізовані зловмисниками;

3) удосконалено метод організації функціонування грід-обчислювальних систем, який на відміну від відомих залучає жадібний алгоритм для оптимізації навантаження між автономними гетерогенними обчислювальними елементами та використовує додаткову чергу активних задач, що дає змогу забезпечити збалансоване виконання поставлених задач в розподілених системах із динамічно

змінюваною топологією для розподіленого виявлення зловмисної поведінки в інфікованих програмах;

4) удосконалено метод оцінювання довіри автономних обчислювальних елементів, який на відміну від відомих використовує механізми призначення ролей із використанням елементів нечіткої логіки, що дає змогу оптимізувати використання обчислювальних ресурсів шляхом скорочення кількості повторних обчислень у системах, що функціонують в динамічному середовищі.

Практичне значення отриманих результатів. Розроблено централізовану грід-обчислювальну систему виявлення інфікованих програм, зокрема таких, що застосовують методи виявлення емуляції та обфускації коду. Система використовує автономні обчислювальні елементи для розподіленого виконання завдань, пов'язаних з аналізом інфікованих програм в ізольованому середовищі. Особливістю запропонованої централізованої розподіленої системи є використання засобів для аналізу інфікованих програм, що поєднують базові реалізації пісочниці та емулятора, які розгортаються на обчислювальних елементах системи. Для ефективного використання залучених обчислювальних ресурсів для аналізу інфікованих програм в модифікованих ізольованих середовищах, система використовує модифікований алгоритм розподілу завдань який враховує гетерогенність обчислювальних елементів. Крім того, з метою оптимізації залучених обчислювальних ресурсів використовується оцінювання довіри автономних обчислювальних елементів, що базується на рольовій моделі і при цьому зберігає необхідний рівень захисту обчислень. Синтезовані системи такого можуть бути використані іншими науковцями та розробниками для задач, що пов'язані із аналізом особливостей виконання зразків програм.

За результатами проведених експериментальних досліджень встановлено, що розроблена централізована грід-обчислювальна система забезпечує коректне функціонування в умовах динамічного середовища виконання, ефективне залучення акумульованих обчислювальних ресурсів для виконання задач виявлення інфікованих програм.

Теоретичні та практичні результати дослідження впроваджені в ТОВ «Nolt technologies» (м. Хмельницький), ТОВ «ІТТ» (м. Хмельницький), а також, в освітньому процесі Хмельницького національного університет при викладанні дисциплін на кафедрі комп'ютерної інженерії та інформаційних систем для спеціальності 123 Комп'ютерна інженерія, зокрема в курсах «Теорія і проєктування комп'ютерних та кіберфізичних систем і мереж», «Безпека та захист комп'ютерних систем», «Комп'ютерні мережі, системне адміністрування та кібербезпека»

У вступі представлено обґрунтування актуальності наукової задачі, що полягає у покращенні ефективності виявлення інфікованих програм, які застосовують методи уникнення виявлення. Важливим напрямком досліджень визначено грід-обчислювальні системи, а також синтез засобів аналізу виконання програм, що базуються на використанні технологій пісочниці та емулятора. Продемонстровано зв'язок тематики дослідження з напрямками наукових досліджень науковців з цієї тематики, представлено основні наукові результати роботи, їх практичне використання, перелік підприємств та установ, в яких впроваджено результат роботи.

У першому розділі проведено аналіз предметної області дослідження, існуючих комерційних програмних застосунків і систем для виявлення зловмисного програмного забезпечення, а також методів, що застосовуються для його виявлення. Проаналізовано існуючі розподілені обчислювальні системи, їх організацію та особливості функціонування.

У другому розділі розглянуто властивості та особливості виконання інфікованих програм. Представлено їх модель, яка враховує методи виявлення емуляції в середовищі виконання, методи обфускації а також стратегії виконання в цільовому середовищі. Описано архітектуру засобу для формування поведінки виконання програм, що базується на технологіях пісочниці та емулятора. Також наведено архітектуру центрального серверу. Він використовує представлений засіб для організації розподіленого аналізу інфікованих програм. Архітектури як засобу, так і центрального серверу подано з погляду їхніх компонентів і функцій із використанням алгебраїчної структури. Графічно архітектуру системи, що поєднує

функціональні особливості засобу аналізу і центрального сервера. Це демонструє загальне функціонування розподіленої системи виявлення інфікованих програм.

У третьому розділі представлено новий метод синтезу засобів формування шаблонів поведінки інфікованих програм, який залучає пісочниці для їх виконання у наборі створюваних модифікованих ізольованих середовищах за допомогою виконання програмних переривань, що дає змогу отримувати з них шаблони поведінки на множинах станів емульованих центральних процесорів із використанням базового емулятора із визначеним набором реалізованих низькорівневих інструкцій, для виявлення зловмисної поведінки в інфікованих програмах з урахуванням вбудованих методів уникнення від виявлення у них. Додатково, представлено удосконалений метод організації обчислен в грід-обчислювальних системах який використовує жадібний алгоритм та додаткову чергу виконання для оптимізації розподілення навантаження між автономними гетерогенними обчислювальними елементами, що дає змогу забезпечити збалансоване виконання поставлених задач в розподілених системах із динамічно змінюваною топологією для розподіленого виявлення інфікованих програм. Також, представлено удосконалений метод оцінювання довіри автономних обчислювальних елементів, який базуючись на механізмах призначення ролей та елементів нечіткої логіки дає змогу оптимізувати використання обчислювальних ресурсів через скорочення кількості повторних обчислень у системах.

У четвертому розділі описано розроблені програмні частини центрального сервера та обчислювального елементу, особливості їх реалізації та використані бібліотеки для їх реалізації, що забезпечують працездатність реалізованої розподіленої системи. Частково подано розроблений мережевий протокол, який визначає основні кроки передачі повідомлень, що забезпечує обмін даними між обома програмними частинами. Представлено проведені експерименти, оцінювання ефективності розподіленої системи, проведено аналіз з отриманих результатів.

У висновках представлено отримані наукові та практичні результати проведеного дослідження.

У додатках представлено наукові публікації, в яких відображено наукові результати роботи, акти впровадження результатів роботи, лістинг розробленого програмного забезпечення.

Ключові слова: розподілені системи обчислень, грід-обчислювальні системи, автономні обчислювальні елементи, зловмисне програмне забезпечення, поведінка виконання, інфікована програма.

ANNOTATION

Rehida P. H. Methods and tools for organizing distributed systems for detecting infected executable programs resistant to execution environment emulation. – Qualifying scientific work on manuscript rights.

Dissertation for obtaining the scientific degree of Doctor of Philosophy in the field of knowledge 12 Information technologies in the speciality 123 Computer engineering. – Khmelnytskyi National University, Khmelnytskyi. 2025.

Solving the problem of improving the efficiency of grid computing systems for detecting infected programs that employ evasion techniques, by organizing the process of distributed execution behaviour analysis using software interrupts on autonomous computing elements, is a relevant scientific challenge.

The dissertation presents an analysis of the architecture of modern distributed computing systems, methods for organizing their operation, and techniques for detecting infected programs, including those that employ anti-emulation and obfuscation techniques to evade detection. The work introduces the architecture of a distributed computing system based on grid systems to enable distributed detection of malicious behaviour in infected programs. It also proposes a method for detecting malicious behaviour of infected programs based on the analysis of execution behaviour within an isolated modified environment using software interrupts, a method for organizing a grid-distributed system utilizing autonomous computing elements in a dynamic operational environment, and the application of a modified trust model to reduce the number of redundant task verifications based on roles and elements of fuzzy logic. Furthermore, a corresponding distributed system has been developed.

The object of the study is the process of organizing a distributed system for detecting malicious behaviour in infected programs that employ evasion techniques.

The subject of research is methods for organizing distributed systems with computing elements that possess a level of autonomy for detecting malicious behaviour in infected programs.

The aim of the dissertation research is to improve the efficiency of grid-computing systems with autonomous computing elements for detecting malicious control flow transfer in infected programs, based on the concept of dynamic emulation states of the program execution environment.

The scientific novelty of the obtained results is as follows:

1) For the first time, a model of a centralized grid computing system has been developed, which considers the requirements for involving autonomous and heterogeneous computing elements to ensure task execution with correctness verification in a dynamic execution environment. This model enables the use of connected computing elements for analysing the execution behaviour of infected programs, supporting a distributed process of detecting malicious behaviour in infected programs;

2) A new method for synthesizing tools to generate behavioural patterns of infected programs has been developed. Unlike existing approaches, this method incorporates the use of a sandbox to execute the programs within a set of custom-modified isolated environments. Execution occurs via software interrupts and a basic emulator implementing a defined set of low-level instructions. This enables the extraction of behavioural patterns from the state sets of emulated central processing units, facilitating the detection of malicious behaviour while accounting for the specific evasion techniques employed by malware developers;

3) The method for organizing the operation of grid computing systems has been improved. In contrast to existing methods, it utilizes a greedy algorithm to optimize workload distribution among autonomous heterogeneous computing elements and employs an additional active task queue. This ensures balanced execution of assigned tasks in distributed systems with dynamically changing topology, enhancing the distributed detection of malicious behaviour in infected programs;

4) The method for trust evaluation of autonomous computing elements has been improved. Unlike known methods, it introduces a role assignment mechanism based on elements of fuzzy logic. This allows for the optimization of computational resource usage by reducing redundant computations in systems operating within dynamic environments.

Practical significance of the obtained results. A centralized grid computing system for detecting infected programs, including those employing anti-emulation and obfuscation techniques, has been developed. The system utilizes autonomous computing elements for the distributed execution of tasks related to the analysis of infected programs in isolated environments. A key feature of the proposed centralized distributed system is the use of tools for analysing infected programs that combine basic implementations of sandboxing and emulation, deployed on the system's computing elements. To efficiently utilize the involved computing resources for analysing infected programs in modified isolated environments, the system employs a modified task distribution algorithm that considers the heterogeneity of the computing elements. Additionally, to optimize the use of involved computing resources, a modified trust module based on a role-based model is used, while maintaining the necessary level of computation security. In addition, in order to optimize the involved computing resources, the assessment of the confidence of autonomous computing elements, based on the role model and thus retains the required level of computing protection. Synthesized systems of this can be used by other scientists and developers for tasks related to the analysis of the features of execution of programs.

Based on the results of experimental studies, it has been established that the developed centralized grid computing system ensures correct operation under dynamic execution environments, efficient utilization of accumulated computing resources for performing tasks related to the detection of infected programs and guaranteed correct execution of the assigned tasks.

The theoretical and practical results of the research were implemented in ITT LLC (Khmelnytskyi), Nolt technologies LLC (Khmelnytskyi), as well as, in the educational process of the Khmelnytskyi National University when teaching disciplines at the Department of Computer Engineering and Information Systems for the specialty 123 Computer Engineering, in particular in the courses “Theory and Design of Computer and

Cyber-Physical Systems and Networks", "Security and Protection of Computer Systems", "Computer Networks, System Administration and Cyber Security"

The introduction presents the justification for the relevance of the scientific problem, which lies in improving the efficiency of detecting infected programs that employ evasion techniques. Grid computing systems and the synthesis of tools for analyzing program execution based on sandbox and emulator technologies are identified as important research directions. The connection between the research topic and the directions of scientific investigations by other researchers in this field is demonstrated, along with the main scientific results of the work, their practical applications, and a list of enterprises and institutions where the results have been implemented.

The first section analyses the research domain, existing commercial software applications and systems for detecting malicious software, as well as the methods employed for its detection. Existing distributed computing systems, their organization, and operational features are also reviewed.

The second section discusses the properties and execution features of infected programs. A model is presented that accounts for methods of detecting emulation within the execution environment, obfuscation techniques, and execution strategies within a target environment. The architecture of a tool for forming execution behaviour based on sandbox and emulator technologies is described. Additionally, the architecture of the central server, which utilizes the proposed tool to organize distributed analysis of infected programs, is presented. The architectures of both the tool and the central server are described in terms of their components and functions using algebraic structures. A graphical representation illustrates the system architecture, combining the functional features of the analysis tool and the central server, demonstrating the overall operation of the distributed infected program detection system.

The third section presents a new method for synthesizing tools for constructing behavioral templates of infected programs. This method utilizes sandboxes to execute the programs within a set of modified isolated environments using software interrupts. It enables the forming of behavioral templates from the state sets of emulated central processing units, employing a basic emulator with a predefined set of implemented low-level instructions.

The approach facilitates the detection of malicious behavior in infected programs, taking into account embedded evasion techniques. Also, the section introduces an improved method for organizing computations in grid computing systems. This method incorporates a greedy algorithm and an additional execution queue to optimize workload distribution across autonomous heterogeneous computational elements. It ensures balanced task execution in distributed systems with dynamically changing topologies, thereby enhancing the distributed detection of infected programs. Furthermore, an improved method for trust evaluation of autonomous computational elements is proposed. Based on role assignment mechanisms and elements of fuzzy logic, this method optimizes resource usage by reducing redundant computations within the system.

The fourth section describes the developed software applications for the central server and the computing elements, as well as discusses the specifics of their implementation. The network communication protocol between the server and computing elements is partially presented. Special attention is given to the description of conducted experiments and the analysis of the obtained results.

The conclusions present the scientific and practical results obtained from the conducted research.

The appendices include scientific publications reflecting the research results, implementation certificates of the developed work, and the source code listings of the developed software.

Keywords: distributed computing systems, grid computing system, autonomous computing elements, malicious software, execution behaviour, infected program.

Список публікацій здобувача за темою дисертації

Наукові праці, в яких опубліковані основні наукові результати дисертації

1. Регіда П.Г., Бармак О.В., Каштальян А.С., Манзюк Е.А. Концепція застосування розподілених систем для аналізу поліморфних вірусів. *Вісник Хмельницького національного університету. Технічні науки.* 2024. Т. 331. №1. С. 38-43. DOI: https://doi.org/10.31891/2307-5732-2024-331-4

2. Регіда П.Г., Савенко О.С. Метод виявлення зловмисної активності в інфікованих програмах. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2024. №1. С. 178-186. DOI: https://doi.org/10.32782/IT/2024-4-21

3. Регіда П.Г. Метод організації розподіленої системи виявлення інфікованих програм в ізольованих середовищах. *Вісник Хмельницького національного університету. Технічні науки*. 2025. Т. 347. № 1. С. 554-560. DOI: https://doi.org/10.31891/2307-5732-2025-347-76

4. Регіда П.Г. Поведінкова модель довіри в грід обчислювальній системі на основі нечіткої логіки. *Вимірювальна та обчислювальна техніка в технологічних процесах*. 2025. №1. С. 287-293. DOI: https://doi.org/10.31891/2219-9365-2025-81-35

Праці, які засвідчують апробацію матеріалів дисертації

5. Регіда П. Г. Засіб розподіленого виявлення зловмисного програмного забезпечення із використанням технології емулювання. *XX ювілейна міжнародна науково-практична конференція «Математичне та програмне забезпечення інтелектуальних систем» (МПЗІС-2022)*, Дніпро, Україна, листопад 23-25, 2022. С. 170-171.

6. Rehida P., Sochor T., Martynyuk V., Tarasova O., Orlenko V. A distributed malware detection model based on sandbox technology. *2023 4th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS)*, Khmelnytskyi, Ukraine, March 22–24, 2023. P. 475-485. *(Scopus)* URL: https://ceur-ws.org/Vol-3373/paper32.pdf

7. Rehida P., Savenko O., Kashtalian A., Sachenko A. Malware Detection Tool Based on Emulator State Analysis. *2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Dortmund, Germany, 7–9 September 2023. P. 135-140. *(Scopus)* DOI: https://doi.org/10.1109/IDAACS58523.2023.10348678

8. Rehida P., Markowsky G., Sachenko A., Savenko O. State-based Sandbox Tool for Distributed Malware Detection with Avoid Techniques. *2023 13th International Conference*

on Dependable Systems, Services and Technologies (DESSERT), Athens, Greece, 13–15 October 2023. P. 1-6. *(Scopus)* DOI: https://doi.org/10.1109/DESSERT61349.2023.10416467

9. Rehida P., Savenko O., Sachenko A., Drozd A., Vizhevski P. A trust model that ensures the correctness of computing in grid computing system. 2024 5th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS), Khmelnytskyi, Ukraine, March 28, 2024. P. 388-401. *(Scopus)* URL: https://ceur-ws.org/Vol-3675/paper28.pdf

10. Регіда П., Капустян М. Лигун О. Динамічне емулювання як засіб виявлення поліморфних вірусів із маскувальними техніками. *The 13th International Scientific Conference «ITSec»*, м. Львів, Україна Травень 9-11, 2024. С. 179-180.

Публікації, які додатково відображають наукові результати дисертації

11. Регіда П. Г., Савенко О. С., Нічепорук А. О., Дрозд А. І. Авторське свідоцтво №134190. Україна. Комп’ютерна програма «Програмне забезпечення організації розподіленої системи виконання програмного коду в ізольованих середовищах» («РСВКІС»). Дата реєстрації: 10 березня 2025 р.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ІТ – інформаційні технології

ПВ – поліморфні віруси

ІП – інфікована програма

ПП – персональний пристрій

ЗПЗ – зловмисне програмне забезпечення

ПЗ – програмне забезпечення

ЦП – пентральний процесор

КС – комп’ютерна система

РСО – розподілена система обчислення

ОЕ – обчислювальний елемент

ЦС – центральний сервер

АПЗ – антивірусне програмне забезпечення

ІСЕ – ізольоване середовище із емуляцією

ВСТУП

Актуальність роботи. Використання інформаційних технологій (ІТ) для різних сфер у сучасному житті та практиці є розповсюдженим явищем, що дозволяє автоматизувати рутинні процеси. Таке активне використання супроводжується використанням програмного забезпечення (ПЗ) та різного рівня конфіденційності файлів користувача. Зважаючи на це, зловмисники вдаються до різних підходів для отримання даних, застосовуючи зловмисне програмне забезпечення (ЗПЗ) різних видів та класифікацій. Сучасні методи, хоч і забезпечують високий рівень їх виявлення, але потребують постійного розвитку. Це пов’язано із тим, що розробники ЗПЗ прикладають усіх зусиль, щоб знайти вразливості в нових застосунках та в програмних оновленнях уже існуючих. Особливу увагу потрібно, також, приділити окремому випадку ЗПЗ – поліморфним вірусам, які наділені широким спектром методів уникнення виявлення для уникнення виявлення антивірусними програмними засобами (АПЗ). Враховуючи те, що такі методи реалізуються великою кількістю підходів, а їх різне комбінування потенційно може створювати велику кількість нових загроз для користувачів, то ПЗ, яке вони використовують та їх дані, є складним для дослідження та, відповідно, виявлення.

Ефективним та безпечним рішенням, що застосовується в більшості сучасних засобах виявлення та ідентифікації ЗПЗ є використання технології пісочниць. Основна перевага використання пісочниць пов’язана із використанням технологій емуляції та ізольованого середовища виконання, що надає безпечний та контрольований характер процесу аналізу. Таке рішення є провідним, та потребує детального аналізу і дослідження, з метою створення нових методів виявлення або удосконалення уже існуючих на їх основі.

Розробники ЗПЗ активно досліджують методи, якими відбувається їх виявлення і вдаються до різних стратегій їх захисту, зокрема застосовують анти-емуляційні методи. Ці методи націлені на аналіз середовища виконання, з метою визначення присутності емуляції, що дозволяє приховувати аномальну поведінку на хості. Таким чином, автори ЗПЗ намагаються приховати його аномальну(зловмисну?) поведінку або відтермінувати його виявлення у часі. В разі виявлення, його сигнатура буде

проаналізована та буде додана у відповідну базу усіх сучасних антивірусних програмних засобів (АПЗ) та більше не буде становити загрози. Тому, важливим завданням є організація дослідження динамічної емуляції для перевірки ПЗ на наявність аномальної поведінки. Використання такого підходу дозволить відтворювати ПЗ в різних умовах, які будуть забезпечуватись різними факторами емуляції. Аналізуючи результат відтворення дозволить зробити висновок про аномальність поведінки.

Особливістю використання емуляції є надлишкове використання обчислювальних ресурсів системи порівнюючи із звичним відтворенням програмних застосунків на хості. Застосування динамічного емулювання потребує застосункових обчислювальних ресурсів, тому буде необхідно залучати наукові дослідження суміжних галузей для ефективного її вирішення. Одним із передових та перспективних напрямків є використання розподілених комп'ютерних систем для вирішення поставленої задачі, які можуть залучити обчислювальні ресурси різнорідних хостів.

Дослідженням теорії та практики функціонування розподілених систем займаються Андерсон Д. [5], Вуд Б. [113], Гудсон Н. [46], Мухін В. [74], Харченко В.[56], Фенг Дж. [36]. Розвитком теорій, що стосуються виконання ЗПЗ, інфікування ПЗ та методів захису від виявлення, які вони застосовують, займаються Бек С. [8], Ґаллоро Н. [38], Гнатюк С.[131], Коваленко А. [123], Леон Р. [63], Летичевський О. [64], Марковський Дж. [68], Саченко А. [69], Хохлачова Ю. [57], Юдін О.[131].

Виходячи із цього, важливим напрямком дослідження є організація та функціонування розподілених комп'ютерних систем для обчислення як засіб забезпечення обчислювальними ресурсами задач динамічного емулювання. Така система повинна забезпечувати ефективне використання обчислювальних вузлів та захищений процес виявлення ЗПЗ для великої кількості різнорідних обчислювальних пристроїв. Результатом розробки такої системи може буде гранична обчислювальна система із керуючим центральним сервером, набором проміжних серверів, що йому підпорядковуються, та великої кількості обчислювальних елементів (ОЕ), які розподілені між ними. Така система повинна ефективно використовувати наявні

обчислювальні ресурси застосовуючи планування виконання завдань в реальному часі та враховувати програмно-апаратні властивості усіх ОЕ. У системі, також, повинні бути передбачені механізми перевірки правильності виконаного аналізу ПЗ, що будуть враховувати характер функціонуючої системи. В запропонованому класі розподілених систем усі ОЕ мають певний рівень автономії, а отже можуть приймати рішення щодо частки обчислювальних ресурсів, яку вони можуть надавати в поточний момент часу, та кількості часу, протягом якого вони приймають участь в роботі системи. Зважаючи на це, доцільно використати концепцію цифрового ідентифікатора, що буде базуватись на поведінкових особливостях кожного ОЕ та буде використовуватись в задачах перевірки коректності обчислень та їх плануванні в режимі реального часу.

Тому, важливою та актуальною науковою задачею є покращення ефективності функціонування грід-обчислювальних систем для виявлення інфікованих програм з втіленими в них методами протидії емуляції. Забезпечення прояву зловмисного коду в таких інфікованих програмах ускладнене через постійне вдосконалення зловмисниками методів уникнення виявлення в поширюваних ними інфікованих програмах. Для виявлення таких інфікованих програм необхідні більші обчислювальні ресурси порівняно з виявленням інфікованих програм без реалізованих в них методів уникнення виявлення та забезпечення гнучко змінюваного середовища виконання, в якому могли б бути активовані зловмисні коди. Щоб забезпечити прояв зловмисних кодів у таких інфікованих програмах доцільно модифіковані створювати ізольовані середовища для їх виконання та забезпечити динамічне балансування ресурсами грід-обчислювальних систем, щоб цілісна система емуляції інфікованих програм формувала для них пастки. В результаті для виявлення зловмисної передачі управління головним потоком в інфікованих програмах може бути доцільним застосувати концепцію динамічного стану емулювання середовища відтворення програмних засобів.

Зазначена науково-прикладна задача відповідає предметній області Стандарту вищої освіти України зі спеціальності 123 Комп’ютерна інженерія для третього (освітньо-наукового) рівня вищої освіти, зокрема, таким об’єктам вивчення та

діяльності: методи та способи подання, отримання, зберігання, передавання, опрацювання та захисту інформації в комп'ютерних системах, математичні моделі обчислювальних процесів та технології виконання обчислень, архітектура та організація їх функціонування, інтерфейси та протоколи взаємодії їх компонентів.

Зв'язок роботи з науковими програмами, планами, темами. Дисертаційне дослідження виконувалось у рамках науково-дослідної тематики Хмельницького національного університету: держбюджетної науково-дослідної теми №1Б-2021 «Самоорганізована розподілена система виявлення зловмисного програмного забезпечення в комп'ютерних мережах» (номер держреєстрації 0121U109936); держбюджетної науково-дослідної теми №2Б-2024 «Система виявлення ЗПЗ та комп'ютерних атак в корпоративних мережах з використанням хибних об'єктів атак та пасток» (номер держреєстрації 0124U000980), в яких автор дисертації був виконавцем.

Мета і завдання дослідження.

Об'єктом дослідження є процес організації розподіленої системи для виявлення зловмисної поведінки в інфікованих програмах, які використовують методи уникнення виявлення.

Предметом дослідження є методи організації розподілених систем із обчислювальними елементами, що мають рівень автономії для виявлення зловмисного прояву в інфікованих програмах.

Метою дисертаційного дослідження є покращення ефективності функціонування грід-обчислювальних систем із автономними обчислювальними елементами для виявлення зловмисної передачі управління головним потоком в інфікованих програмах, що базується на концепції динамічного стану емулювання середовища відтворення програмних засобів.

Завдання дослідження у цій роботі сформовані наступним чином:

1) провести аналіз методів організації та функціонування грід-обчислювальних систем, дослідити методи оцінювання ефективності залучення обчислювальних елементів із рівнем автономії та їх безпечного і коректного функціонування для організації виявлення інфікованих програм;

2) розробити формальний опис та моделі інфікованих програм, що використовують методи уникнення виявлення, а саме ідентифікацію/визначення емульованого середовища виконання та обфускацію коду виконання, що дозволить відтворити їх поведінку для опису їх впливу на цільову систему користувача;

3) розробити та представити формальний опис функціонування і модель грід-обчислювальних систем із використанням автономних обчислювальних елементів, що повинна враховуватись при організації захищеного та правильного (коректного) виявлення зловмисної поведінки в інфікованих програмах;

4) розробити метод синтезу засобів формування шаблонів поведінки інфікованих програм, які використовують методи уникнення виявлення, що залучають базові пісочницю та емулятор використовуючи множини станів емульованих центральних процесорів для виявлення зловмисної поведінки;

5) удосконалити метод організації грід-обчислювальних систем для оптимального розподілу завдань між залученими автономними гетерогенними обчислювальними елементами для виявлення зловмисної поведінки в інфікованих програмах;

6) удосконалити метод оцінювання довіри автономних обчислювальних елементів для оптимізації використання обчислювальних ресурів шляхом скорочення кількості повторних обчислень в грід-обчислювальних системах;

7) розробити грід-обчислювальну систему виявлення інфікованих програм, які використовують методи уникнення виявлення, для експериментального дослідження з метою покращення її характеристик та її подальшого впровадження для практичного застосування.

Методи дослідження. Для вирішення поставлених задач використовуються основні положення абстрактної алгебри, теорії розподілених систем, методи маскування інфікованих програм, теорії елементів штучного інтелекту та генетичних алгоритмів:

1. Теорії абстрактної алгебри для визначення моделі грід-обчислювальних систем та деталізованого представлення її основних елементів та їх компонентів.

2. Теорії множин і теорії графів для визначення інфікованих програм, що характеризуються певним набором методів для уникнення виявлення.

3. Теорії розподілених систем для організації функціонування системи та її компонентів виявлення зловмисної передачі управління поліморфними вірусами.

4. Методи уникнення виявлення інфікованих програм для формування моделі системи їх виявлення на основі аналізу відтворення у пісочниці.

Обґрунтованість і достовірність наукових положень, висновків і рекомендацій. Наукові положення, висновки і рекомендації дисертації обґрунтовані коректним та доцільним використанням математичного апарату у вигляді математичних моделей усіх компонентів запропонованої розподіленої обчислювальної системи та різним варіантам моделей інфікованих програм, що формуються різним набором методів уникнення виявлення, реалізацією розробленої системи що складається із програмних частин ЦС та ОЕ, які забезпечують процес аналізу низькорівневих інструкцій коду з метою визначення наявності зловмисної передачі управління виконання застосунку, практичним та наочним впровадженням результатів дисертаційного дослідження на підприємствах, які використовують розподілені обчислювальні системи і які демонструють відповідність отриманих результатів теоретичних досліджень із практичними результатами застосування.

Наукова новизна отриманих результатів полягає в наступному:

1) вперше розроблено модель грід-обчислювальних систем, в якій враховано вимоги до залучення автономних та гетерогенних обчислювальних елементів для забезпечення виконання задач із перевіркою на коректність в динамічному середовищі виконання, і яка дає змогу залучити під’єднані обчислювальні елементи для аналізу поведінки виконання інфікованих програм, забезпечуючи розподілений процес виявлення зловмисного прояву в інфікованих програмах;

2) розроблено новий метод синтезу засобів формування шаблонів поведінки інфікованих програм, який на відміну від відомих відрізняється залученням пісочниці для їх виконання у наборі створюваних модифікованих ізольованих середовищах за допомогою виконання програмних переривань та базового емулятора із визначеним набором реалізованих низькорівневих інструкцій, що дає змогу отримувати з них

шаблони поведінки на множинах станів емульованих центральних процесорів з метою виявлення зловмисної поведінки з урахуванням особливостей методів уникнення від виявлення, які реалізовані зловмисниками;

3) удосконалено метод організації функціонування грід-обчислювальних систем, який на відміну від відомих залучає жадібний алгоритм для оптимізації навантаження між автономними гетерогенними обчислювальними елементами та використовує додаткову чергу активних задач, що дає змогу забезпечити збалансоване виконання поставлених задач в розподілених системах із динамічно змінюваною топологією для розподіленого виявлення зловмисної поведінки в інфікованих програмах;

4) удосконалено метод оцінювання довіри автономних обчислювальних елементів, який на відміну від відомих використовує механізми призначення ролей із використанням елементів нечіткої логіки, що дає змогу оптимізувати використання обчислювальних ресурсів шляхом скорочення кількості повторних обчислень у системах, що функціонують в динамічному середовищі.

Практичне значення отриманих результатів. Розроблено централізовану грід-обчислювальну систему виявлення інфікованих програм, зокрема таких, що застосовують методи виявлення емуляції та обфускації коду. Система використовує автономні обчислювальні елементи для розподіленого виконання завдань, пов’язаних з аналізом інфікованих програм в ізольованому середовищі. Особливістю запропонованої централізованої розподіленої системи є використання засобів для аналізу інфікованих програм, що поєднують базові реалізації пісочниці та емулятора, які розгортаються на обчислювальних елементах системи. Для ефективного використання залучених обчислювальних ресурсів для аналізу інфікованих програм в модифікованих ізольованих середовищах, система використовує модифікований алгоритм розподілу завдань який враховує гетерогенність обчислювальних елементів. Крім того, з метою оптимізації залучених обчислювальних ресурсів використовується оцінювання довіри автономних обчислювальних елементів, що базується на рольовій моделі і при цьому зберігає необхідний рівень захисту обчислень. Синтезовані системи такого можуть бути використані іншими науковцями

та розробниками для задач, що пов’язані із аналізом особливостей виконання зразків програм.

За результатами проведених експериментальних досліджень встановлено, що розроблена централізована грід-обчислювальна система забезпечує коректне функціонування в умовах динамічного середовища виконання, ефективне залучення акумульованих обчислювальних ресурсів для виконання задач виявлення інфікованих програм.

Теоретичні та практичні результати дослідження впроваджені в ТОВ «Nolt technologies» (м. Хмельницький), ТОВ «ІТТ» (м. Хмельницький), а також, в освітньому процесі Хмельницького національного університет при викладанні дисциплін на кафедрі комп’ютерної інженерії та інформаційних систем для спеціальності 123 Комп’ютерна інженерія, зокрема в курсах «Теорія і проєктування комп’ютерних та кіберфізичних систем і мереж», «Безпека та захист комп’ютерних систем», «Комп’ютерні мережі, системне адміністрування та кібербезпека».

Особистий внесок здобувача. Всі основні результати дисертаційного дослідження, які представлені до захисту, отримані автором особисто.

У роботах, опублікованих одноосібно автором, отримано наступні результати: [124] – розроблено модель системи виявлення інфікованої програми; [128] – розроблено метод організації розподіленої системи для виявлення інфікованих програм; [129] – розроблено модуль довіри для грід-обчислювальних систем.

У роботах, які опубліковані у співавторстві, автору належать основні ідеї, теоретична та практична розробка положень, відображених у характеристиці наукової новизни отриманих результатів, а саме: [91] – розроблено модель для аналізу інфікованих програм на основі пісочниці; [89] – розроблено базовий засіб для формування поведінки виконання програми в ізольованому середовищі; [88] – розроблено програмний застосунок для обчислювального елементу системи, який містить засіб для аналізу виконуваних програм; [126] – проведений аналіз сучасних способів організації розподілених систем; [90] – розроблено модель із ролями обчислювальних елементів розподіленої системи, та її реалізація; [127] – запропоновано концепцію використання модифікованих ізольованих середовищ для

виявлення зловмисного прояву; [130] – розроблено метод виявлення інфікованої програми із використанням грід-обчислювальної системи.

У роботах, які опубліковані у співавторстві, співавторам належать такі результати: [91] – запропоновано алгоритм залучення нових обчислювальних елементів в систему (Сохор Т.), запропоновано алгоритм перевірки правильності виконання на основі голосування (Мартинюк В.), запропоновано використання довіреного обчислювального елементу для верифікації голосування (Тарасова О.), представлено алгоритм залучення проміжних серверів (Орленко В.); [89] – запропоновано концепцію для розроблення моделі систем для аналізу інфікованих програм (Савенко О.), проведено аналіз методів протидії емуляції та обфускації (Каштальян А.), здійснено дослідження часової залежності формування станів різними стратегіями (Саченко А.); [88] – здійснено дослідження зміни поведінки для порівняння оригінальної програми і обфускованої (Марковський Дж.), здійснено постановку експерименту для дослідження ефективності застосування розподіленої системи для аналізу програм (Саченко А.), визначено стратегії поведінки виконання інфікованих програм в цільовому середовищі (Савенко О.); [126] – визначено ключові характеристики для дослідження функціонування систем, а саме: стійкість та рівновага (Бармак О.); запропонована концепція розподіленої системи виявлення поліморфних вірусів (Каштальян А.); запропоновано планування експерименту та аналіз його результатів (Манзюк Е.); [90] – запропоновано визначення ролі обчислювального елемента на основі його поведінки під час функціонування системи (Савенко О.), запропоновано використання рейтингу обчислювального елементу та бінарного класифікатора для його коригування (Саченко А.), проведено аналіз готових засобів для розгортання бінарного класифікатора (Дрозд А.), сформовано список інформації, який можна збирати на обчислювальному елементі, шляхом розширення його функціональності (Віжевський П.); [127] – представлено базову модель інфікованих програм (Капустян М.), представлено спосіб формування модифікованого ізольованого середовища (Лигун О.); [130] – здійснено оптимізацію пам’яті програми при формуванні поведінки інфікованої програми (Савенко О.).

Апробація результатів дисертації. Апробацію основних положень, ідей, висновків дисертаційної роботи проведено на науковому семінарі кафедри комп'ютерної інженерії та інформаційних систем у Хмельницькому національному університеті. Наукові результати роботи доповідались на таких конференціях: XX ювілейна міжнародна науково-практична конференція «Математичне та програмне забезпечення інтелектуальних систем» (МПЗІС-2022, Дніпро, Україна, 23-25 листопада 2022); 4th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS, Khmelnytskyi, Ukraine, March 22–24, 2023); 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS, Dortmund, Germany, 7–9 September 2023); 13th International Conference on Dependable Systems, Services and Technologies (DESSERT, Athens, Greece, 13–15 October 2023); 5th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS, Khmelnytskyi, Ukraine, March 28, 2024); The 13th International Scientific Conference («ITSec», м. Львів, Україна Травень 9-11, 2024).

Публікації. За результатами проведених досліджень основні наукові результати опубліковано у 4 наукових статтях в трьох фахових наукових журналах України [126, 128-130]. Апробація засвідчена публікаціями 6 праць в матеріалах міжнародних та всеукраїнських конференцій [88-91, 124, 127], з яких 4 праці проіндексовано у наукометричній базі Scopus [88-91]. Опубліковано та отримано одне свідоцтво про реєстрацію авторського права на твір (програму) [125].

Структура та обсяг дисертації. Дисертаційна робота складається з анотації, змісту, переліку умовних скорочень, вступу, чотирьох розділів, висновку, списку використаних джерел та чотирьох додатків. Повний обсяг роботи містить 188 сторінок друкованого тексту, з них анотація – на 10 стор., зміст – на 2 стор., перелік умовних скорочень – на 1 стор., основний текст – на 128 стор., список із 131 використаних джерел – на 17 стор., додатки – на 27 стор. Дисертація містить 27 рисунків та 10 таблиць.

РОЗДІЛ 1.
АНАЛІЗ МЕТОДІВ ОРГАНІЗАЦІЇ РОЗПОДІЛЕНИХ СИСТЕМ ВИЯВЛЕННЯ ІНФІКОВАНИХ ВИКОНУВАНИХ ПРОГРАМ З ПОЛІМОРФНИМИ ВЛАСТИВОСТЯМИ

У сучасному світі використання ІТ зростає кількісно і якісно для усіх сфер життя. Із кожним роком, ІТ пропонують користувачам нові засоби для автоматизації рутинних, як приватних так і комерційних, щоденних справ. Завдяки зручності та доступності використання, вони регулярно отримують оновлення та покращення функціональних особливостей. Такі засоби найчастіше представлені у вигляді програмних настільних і мобільних застосунків, веб-сервісів та різноманітних програмно-апаратних рішень. Щоб покращити спосіб взаємодії користувачів та таких засобів, розробники вдаються до різного виду інтеграцій засобу із платформою його відтворення та залучають дані користувачів. В результаті, подібна інтеграція дозволяє засобам оперувати персональними або корпоративними даними. Знаючи це, розробники ЗПЗ досліджують актуальне ПЗ та шукають можливість його модифікації із метою його інфікування, що дозволить отримати доступ до цих конфіденційних даних. Результати досліджень наукових лабораторій, що спеціалізуються на виявленні, аналізі та оцінці нових загроз в ІТ, свідчать про стрімке зростання кількості нових зразків ЗПЗ. Проведені дослідження вказують на те, що сучасні ЗПЗ стають дедалі складнішими, так як використовують комбіновані та модифіковані технології обходу традиційних систем захисту. У відповідь на таку динаміку фахівці з кібербезпеки розробляють і впроваджують не лише нові методи, але й комплексі підходи до захисту, які застосовуються в настільних застосунках і модульних програмних системах, розгорнутих на різних пристроях корпоративних систем. Основними методами протидії є використання технологій емуляції та пісочниці, застосування статичного та динамічного аналізу, а також використання елементів штучного інтелекту, реверс-інжинірингу та криптографічного аналізу.

1.1. Аналіз відомих систем виявлення інфікованих виконуваних програм з поліморфними властивостями

Широке використання ІТ у багатьох сферах сучасного життя змушує розробників ЗПЗ адаптувати свої рішення для роботи в різних середовищах. У звітах [105,20] про аналіз сучасних загроз визначено кілька ключових напрямів атак, зокрема: шифрувальні атаки, атаки з вимаганням викупу, атаки із застосуванням ЗПЗ, атаки для отримання несанкціонованого доступу до даних або ресурсів, а також використання вразливостей пристроїв IoT [123]. Фахівці з кібербезпеки, реагуючи на такі загрози, пропонують численні засоби захисту – як ті, що встановлюються безпосередньо на пристрій [28,92], так і ті, що охоплюють усі компоненти мережі [42,44]. Постійно вдосконалюючи їхні можливості, вони формують різноманітні засоби, які можна розділити на дві категорії.

До першої категорії відносять засоби, які представлені у вигляді програмних застосунків які використовуються в рамках настільних або мобільних операційних систем. Вони орієнтовані здебільшого на користувачів і призначені для виявлення та нейтралізації ЗПЗ, виконуючи сканування та аналіз системної активності та мережевого трафіку [30].

Антивірусний застосунок Norton 360 пропонується для більшості настільних та мобільних операційних систем. Він вирізняється швидкодією та частими оновленнями бази сигнатур. Серед його особливостей можна виділити [35] вбудований VPN для користувачів, засоби сканування підозрілої активності пристроїв IoT, підключених до локальної мережі, а також хмарне зберігання резервних копій даних.

Bitdefender total security є комплексним рішенням для забезпечення безпеки, який має проактивний характер [13] і базується на використанні штучного інтелекту. Вирізняється застосуванням захисту встановленої веб-камери, вбудованим VPN, а також функцією Safepay для захисту онлайн-платежів. Окремо слід визначити наявність засобів для сканування незахищених WIFI мереж.

Avast Security також відноситься до категорії антивірусних програм, орієнтованих на захист пристроїв в реальному часі [15]. Серед його особливостей можна відзначити захист підключених периферійних пристроїв і локальної мережі шляхом сканування на наявність вразливостей. Крім того, програма пропонує інструменти для створення та зберігання паролів облікових записів.

Комплексний застосунок McAfee Total Protection, окрім захисту від ЗПЗ пропонує розширені можливості для протидії фішингу завдяки веб-фільтрації [111]. Вирізняється широкою сумісністю з найпопулярнішими операційними системами. Окрім VPN і менеджера паролів, застосунок містить засоби шифрування файлів для захисту конфіденційних даних.

ESET Smart Security – антивірусний застосунок, який забезпечує проактивний захист у реальному часі, використовуючи машинне навчання та регулярні оновлення бази сигнатур [32]. Однією з його особливостей є технологія UEFI Scanner, яка дозволяє сканувати додаткові області пам’яті, що можуть використовуватись під час кожного запуску операційних систем.

Антивірус Avira Prime забезпечує захист операційної системи в реальному часі використовуючи хмарне сканування [61]. Серед його особливостей – безпечне видалення конфіденційних файлів, що унеможливлює їхнє відновлення, а також розширений менеджер оновлень для встановлених програм, який допомагає усувати потенційні вразливості з точки зору безпеки.

Panda Dome Advanced є комплексним рішенням для проактивного захисту на основі машинного навчання. Пропонує кілька режимів роботи, що дозволяє коригувати навантаження на систему. Серед його особливостей – функція антикрадіжки, яка забезпечує віддалене визначення локації, блокування та стирання даних [78], допомагаючи зберегти конфіденційність. Також, Panda Dome Advanced забезпечує захист підключених периферійних пристроїв.

Незважаючи на присутні відмінності та особливості функціонування, усі розглянуті засоби використовують технологію пісочниці для глибокого аналізу файлів і програм перед їх запуском у робочому середовищі. Загалом, пісочниця – це технологія ізоляції підозрілих файлів або процесів у віртуальному середовищі [77],

що дозволяє обмежити їхнє виконання, заблокувати потенційно небезпечну активність [81] і навіть передати об’єкти для подальшого аналізу іншим засобам. До основних переваг використання пісочниці можна віднести такі: безпека – ізоляція загроз у віртуальному середовищі [54]; глибокий аналіз – можливість вивчення поведінки файлів у реальному часі [2]; захист від нульових загроз – виявлення нових шкідливих програм, які ще не мають сигнатур [19].

У сучасних антивірусних рішеннях пісочниця може бути реалізована локально або у вигляді хмарного сервісу [11]. Так, Avast Premium Security та Bitdefender використовують локальну пісочницю, яка розгортається безпосередньо на системі, де встановлений антивірус. З іншої сторони, Avira Prime, McAfee Total Protection та Panda Dome Advanced покладаються на хмарний аналіз, що не вимагає використання локальної пісочниці. У цьому випадку файл або програма передаються до захищеної інфраструктури компанії-розробника [26] для більш точного аналізу [87].

Друга категорія включає засоби, що використовують хмарну інфраструктурну та обчислювальні потужності розробника, забезпечуючи централізований аналіз загроз [122]. У деяких випадках такі рішення надають можливість локального розгортання на власних серверах, що є важливим для організацій із завищеними вимогами до безпеки та контролю над даними. Такі рішення здебільшого орієнтовані на корпоративний сегмент, де важливо не лише виявлення ЗПЗ, а й інтеграція з іншими системами безпеки [99], масштабованість і централізоване управління. Вони часто входять до складу комплексних платформ інформаційної безпеки [110, 131], що об’єднують антивірусний захист, поведінковий аналіз, моніторинг мережевого трафіку, виявлення аномалій та інші функції.

Одним із найвідоміших рішень подібних систем є Symantec Content Analysis and Sandboxing (SCAS), яке належить до передових засобів аналізу ЗПЗ. Воно використовує багаторівневий та динамічний аналіз виконуваних програм та файлів [108], досліджуючи їхню поведінку в ізольованому середовищі. Це рішення також вирізняється застосуванням машинного навчання та евристичних методів для виявлення нових загроз. Користувачі можуть використовувати його як хмарне

рішення для перевірки підозрілих об’єктів або розгортати як локальне рішення для інтеграції з іншими засобами захисту через API.

VirusTotal відноситься до онлайн-сервісів для швидкого аналізу файлів та посилань. Система пропонує безкоштовний доступ для окремих користувачів, а також зручну інтеграцію в поточну інфраструктуру захисту через API. Основний метод виявлення базується на поєднанні статичного аналізу та елементів динамічного аналізу, що забезпечує швидкий аналіз, хоча може мати меншу точність виявлення [120]. Для аналізу сервіс використовує десятки антивірусних рушіїв, результати роботи яких співставляються для визначення, чи є файл або програма зловмисними.

Платформа Cuckoo Sandbox відрізняється відкритістю, а її основним завданням є виконання динамічного аналізу ЗПЗ в ізольованому середовищі. Вона використовується для проактивного виявлення загроз у підозрілих файлах, посиланнях і навіть у мережевому трафіку. Ізольоване середовище для аналізу в цій системі базується на технологіях віртуалізації та контейнеризації [33]. Крім того, платформа підтримує модульність, яка дозволяє змінювати середовище виконання ЗПЗ, методи його аналізу під час виконання, а також способи збору інформації про систему. Також доступні модулі для інтеграції з іншими системами захисту. У цій системі впроваджуване оцінювання, що дозволяє класифікувати підозрілі файли на основі шкали балів, де менша кількість балів означає менший ризик використання.

Хмарний інструмент Hybrid Analysis від CrowdStrike призначений для глибокого аналізу ЗПЗ, що поєднує використання статичного та динамічного аналізу [47]. Статичний аналіз базується на дослідженні структури файлу та його метаданих. Динамічний аналіз виконується в ізольованому середовищі для спостереження за поведінкою програм під час виконання та їхньою взаємодією із системою, включаючи аналіз файлів, які створюються видаляються або модифікуються. Hybrid Analysis проводить аналіз загроз, орієнтованих на різні операційні системи, зокрема Windows, Linux і Android. Інструмент також пропонує широкий функціонал для інтеграції із системами захисту через API.

Інтерактивна пісочниця ANY.RUN забезпечує динамічний аналіз в реальному часі для загроз, орієнтованих на операційні системи Windows та Linux [4]. Глибокий

аналіз охоплює реєстр, файлову систему, поведінку процесів і мережеву активність. Ця платформа вирізняється високою інтерактивністю завдяки можливості взаємодіяти із зразком під час аналізу, вводячи дані або змінюючи файли. *ANY.RUN* пропонує просту інтеграцію із системами інформаційної безпеки, такими як SIEM. Також, підтримується створення шаблоні налаштувань для визначених запусків віртуальних середовищ, що дозволяє автоматизувати процес виявлення загроз у різних умовах виконання.

Платформа Joe Sandbox використовує пісочницю як основний механізм аналізу файлів і програм на наявність загроз. Вона забезпечує аналіз ЗПЗ, орієнтованого на всі сучасні настільні та мобільні операційні системи [50]. Платформа виконує статичний, динамічний і гібридний аналізи для отримання повнішого розуміння поведінки програми. Joe Sandbox вирізняється формуванням детальних звітів про активність, які відображаються у лог-файлах взаємодії файлів із системою, а також візуалізацією взаємозв’язків між компонентами, а інтегровані алгоритми штучного інтелекту покращують автоматичне виявлення загроз.

Система VMRay Analyzer використовує динамічний аналіз як основний метод виявлення ЗПЗ, з основним акцентом на дослідження поведінки виконання програм чи файлів в ізольованому середовищі. Вона вирізняється технологією agentless monitoring, яка зменшує ймовірність виявлення системи вірусами, та здійснювати детальне спостереження за всіма системними викликами до операційної систем та бібліотек ядра [39]. VMRay Analyzer може бути використаний як хмарний інструмент або локально розгорнутий із подальшою інтеграцією в існуючі системи захисту через API. Система також надає можливість використання інтерактивного дебагера для детального дослідження ЗПЗ.

Умовно розділені категорії засоби забезпечення захисту, як локальні застосунки, так і комплексні системи призначені для корпоративного рівня, покладаються на використання як статичного так і динамічного аналізу [16]. Ключовим елементом в обох випадках виступає використання технологій пісочниці та емуляції [84]. Це викликано тим, що поєднання обох технологій забезпечує власне виконання динамічного аналізу, що дозволяє в свою чергу проводити детальне та

комплексне [96], і при цьому безпечне дослідження зразків ЗПЗ різного виду. Завдяки цьому підходу стає можливим отримати глибше розуміння способів роботи ЗПЗ, і як наслідок розробляти на впроваджувати нові механізми захисту [95].

Однією із найскладніших проблем у сфері кібербезпеки є поліморфні віруси (ПВ). Ці загрози здатні змінювати свій код при кожному новому зараженні, що значно ускладнює їхнє виявлення традиційними методами [43, 82]. Використання сигнатурного аналізу стає неефективним, оскільки ЗПЗ такого типу щоразу має інший вигляд на рівні коду, але при цьому зберігає свою зловмисну поведінку. Саме тому більшість сучасних АПЗ застосовують низку різних підходів одночасно, зокрема поведінковий аналіз та еврестичний аналіз, які дозволяють виявляти загрози навіть при зміні їхньої структури.

Пісочниці відіграють ключову роль у виявленні нових зразків ПВ. Завдяки динамічному аналізу вони дозволяють відстежувати поведінкові ознаки зловмисної активності в реальному часі. Це дає змогу не лише ідентифікувати загрозу [68], а й виявляти нові методи маскування, що активно використовуються для уникнення від виявлення. Окрім того, пісочниця формує ізольоване середовище виконання програм, що захищає від зловмисних проявів цільову систему, на час аналізу потенційно небезпечного файлу чи програми [93]. Технологія емуляції, в свою чергу, є також важливим інструментом, що дозволяє імітувати роботу апаратного і програмного забезпечення. Саме вона дозволяє відтворювати програмні застосунки для подальшого аналізу в ізольованому середовищі, що формує пісочниця. Наявність програмної частини в емуляторі, надає гнучкості у його використанні та аналізі процесів, що виконуються у цьому під час його роботи. Іншою важливою властивістю емулятора є можливість запуску програмного забезпечення, що було спроектовано для інших програмно-апаратних платформ. А до недоліків відносять зниження продуктивності виконання програм, що залежить від точності реалізації та наявності вірної документації емульованого об’єкту.

Таким чином, наявність пісочниці та емулятора є основним елементом виявлення нових загроз у сучасних АПЗ. Вони значно підвищують шанс виявлення ПВ, а емулятори доповнюють цей процес, дозволяючи проводити більш детальний

аналіз потенційно небезпечних програм та файлів. Широке застосування, а також наукові дослідження визначають їх як обов'язкові до використання, а отже і дослідження в рамках виявлення нових загроз кібербезпеки.

1.2. Методи виявлення інфікованих виконуваних програм

Дослідження активності появи нових зразків ЗПЗ та аналіз їхнього успішного застосування здійснюється науково-дослідними центрами, провідними інститутами, а також фахівцями з кібербезпеки у приватних компаніях. Аналіз результатів дослідження свідчить про стійку тенденцію до зростання кількості ЗПЗ упродовж останніх років [43], динаміка зростання яких зображена діаграмою на рис. 1.1.

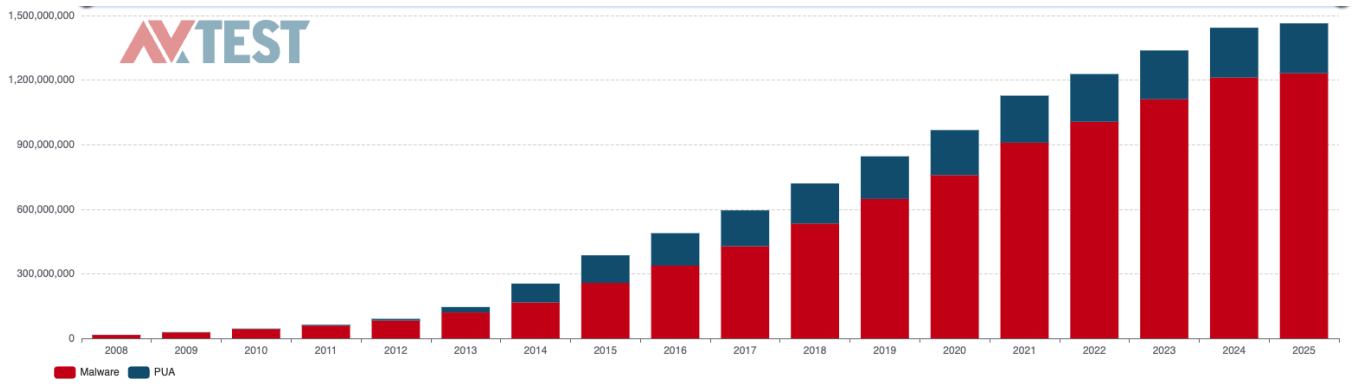


Рисунок 1.1 – Зростання загроз: статистка ЗПЗ за даними AVTest

Результати досліджень з ресурсу [7] формують висновок про те, що хоча ріст традиційних загроз сповільнюється, але з'являються нові типи атак, пов'язані з новими сферами застосування IT [45]. У дослідженні [105], також, визначено основні способи розповсюдження таких загроз. Зокрема, виділяють наступні: фішингові атаки через електронну пошту; використання вразливостей програмних застосунків, з яких найбільшим чином це стосується тих, що використовуються в корпоративному сегменті для обміну даних та передачу повідомлень; експлуатація вразливостей, які спричинені неправильними конфігураціями або відсутністю оновлень у програмних застосунках; використання недосконалостей пошукових рушіїв, через які поширюються небезпечні сайти. Все це обумовлює успішність виконаних атак. У звіті в [24] визначають основні цілі загроз на 2024 рік, а саме: крадіжка облікових даних,

інструменти для розповсюдження веб-орієнтованих атак, використання вразливостей незахищених пристроїв, модифікації драйверів пристроїв з метою використання в зловмисних цілях, атаки на мобільні пристрої. Згідно з дослідженнями [105], 70% кібератак є успішними завдяки методом соціальної інженерії [24]. Крім того, за даними [106] ЗПЗ для настільних застосунків у 90% випадків базується на прихованому використанні системного терміналу операційної системи для реалізації своїх зловмисних цілей. Також, фіксується стабільне зростання «викупу», який зловмисники вимагають у малих і середніх організацій за можливість уникнути втрати даних [66], адже самі такі організації традиційно належать до пріоритетних цілей.

Застосування ЗПЗ як інструменту для зараження ПЗ є одним із найрозповсюджених способів впливу на персональний пристрій (ПП) користувачів. У цьому контексті програма, що зазнала процесу інфікування, вважається інфікованою програмою (ІП). Інфікування зазвичай відбувається на програмах, яким користувачі довіряють і має на меті запуск фрагментів коду зі зловмисною активністю [67]. Одне із основних завдань ІП пролягає у створенні умов для виконання ділянки коду, не пов’язаною з оригінальною програмою [101], під виглядом якої ІП маскується. Ці фрагменти коду несуть зловмисний прояв, основною метою якого є отримання певної вигоди зловмисниками [114]. Часто вони відповідають за запуск заздалегідь підготовлених зловмисниками програм, що працюють у фоні операційних систем користувача. За їх допомогою зловмисники можуть викрадати дані, стежити за активністю ПП, приховано експлуатувати його ресурси, здійснювати інтерактивне шахрайство, розповсюджувати інфіковані програми або отримувати доступ до ПП [55]. Усе це досягається втручанням у нормальний потік виконання програм та передачею її управління. Вважатимемо, що коли така ситуація виникає, то програма виконує зловмисну дію, або іншими словами, має зловмисний прояв. Після завершення виконання відповідного фрагменту коду управління повертається до нормального стану. Здебільшого такі дії відбуваються миттєво з точки зору користувача і залишаються для нього непомітними, адже програма відновлює свій звичний стан.

Окрім зловмисної активності ІП активно застосовують різноманітні методи уникнення виявлення [12]. У межах цих методі, ІП може здійснювати комплексний аналіз середовища виконання для визначення наявності інструментів дебагінгу, емуляційних технологій, пісочниць [75], віртуальних машин [49,38], а також засобів моніторингу мережевого трафіку чи системних інтерфейсів. Усі ці підходи можемо представити у вигляді наступної класифікації:

Обфускація коду охоплює різноманітні поліморфні та метаморфні методи, а також низку технік, спрямованих на заплутування коду [31] шляхом додавання нових фрагментів [103], перемішування існуючих частин [23] або зміни поведінки виконання. Завдяки усьому цьому суттєво ускладняється аналіз логіки програм [40].

Уникнення аналізу полягає не лише у використанні засобів пошуку інструментів аналізу, а й у застосуванні спеціальних пасток [59], що ускладнюють або сповільнюють роботу таких інструментів.

Протидія емуляції використовує перевірку середовища виконання на ознаки емуляторів і пісочниць шляхом аналізу пам’яті на наявність штучних артефактів та коректності її використання [59], вивчення швидкості виконання системних операцій та оцінювання апаратних складових.

Виявлення аналізаторів полягає у пошуку запущених у системі процесів, що можуть бути пов’язані з антивірусними засобами, утилітами аналізу або моніторингу, а також у використанні маніпуляцій із різними API [97] для приховування своєї активності.

Уникнення мережевого аналізу передбачає використання шифрування процесів мережевої взаємодії, застосування спеціалізованих алгоритмів для динамічного формування доменних імен [62], а також використання популярних сервісів для проксування та тунелювання мережевого трафіку, що ускладнює його відстеження.

Системні маніпуляції полягають у внесенні змін до системних файлів і реєстру, впровадженні в легітимні процеси та, у деяких випадках, застосуванні руткіт-технік, які дають змогу приховати наявність зловмисної активності [34].

Розробники ЗПЗ не можуть застосовувати всі класи методів уникнення в межах однієї ІП, оскільки це може надмірно спотворити властивості та поведінку

оригінальної програми. Така ситуація здатна викликати підозру у користувача щодо безпечності роботи програми, знижуючи ймовірність успішного застосування зловмисних дій. Тому, розробники ЗПЗ зазвичай впроваджують ці методи вибірково, орієнтуючись на найбільш характерні для конкретного середовища виконання методи виявлення. Зважаючи на небезпечність та широке розповсюдження ЗПЗ, фахівці у галузі кібербезпеки розробили цілу низку підходів до виявлення таких загроз, які умовно поділяють на два основні класи: статичні та динамічні методи.

Статичні методи ґрунтуються на аналізі програмного коду без його безпосереднього виконання [10]. Вони досліджують вихідний або бінарний код для ідентифікації потенційно зловмисних характеристик, поведінки вразливостей та інших ознак, що можуть свідчити про зловмисний характер програми. До основних методів належить такі: сигнатурний аналіз; аналіз коду; правила на основі вмісту; аналіз ентропії; аналіз графа потоку; аналіз потоку даних. Розглянемо їх детальніше.

Сигнатурний аналіз – найпоширеніший підхід, який базується на порівнянні фрагментів або хешів коду з уже відомими шаблонами (сигнатурами) ЗПЗ [52]. Цей метод ефективний для виявлення уже відомих загроз, але може бути малоефективним щодо нових загроз або навіть модифікованих варіантів ЗПЗ.

Аналіз коду передбачає детальний розгляд структури та інструкцій програмних файлів. Сканування виконується для виявлення нестандартних викликів, підозрілих рядків та відхилень від типової поведінки програм [117] в певній операційній системі.

Правила на основі вмісту базуються на використанні специфічних правил для визначення сигнатур, рядків, метаданих та секцій коду [14]. Завдяки можливості комбінувати різні типи умов та ознак, такі правила можуть виявляти як відомі зразки ЗПЗ, так і їхні нові модифікації, для яких ще немає визначених сигнатур. Подібні правила є гнучкими та легко адаптуються до різних класів ЗПЗ.

Аналіз ентропії полягає у проведенні вимірювання рівня випадковості даних у файлі. Висока ентропія може свідчити про використання методів обфускації або шифрування, що часто застосовується у ЗПЗ [119]. Області файлу із високою ентропією можуть містити код, прихований від статичних методів виявлення. Такий

аналіз є швидким способом ідентифікації файлів, які потребують детальнішого дослідження.

Аналіз графа потоку виконання досліджує послідовність виконання програми у вигляді графа. Такий підхід може виявити підозрілі закономірності, наприклад надмірно складний або заплутаний потік керування [70], що може свідчити про обфускацію або наявність зловмисного коду. У ЗПЗ подібні методи використовуються для ускладнення розуміння логіки роботи програми та приховування її зловмисної функціональності.

Аналіз потоку даних дозволяє простежити, як дані рухаються через програму і які трансформації вони зазнають [18]. Цей метод особливо цінний для виявлення зловмисних дій, таких як витік конфіденційної інформації, маніпуляції з даними або несанкціонований доступ до ресурсів. Він дає змогу ідентифікувати підозрілі патерни обробки даних, що відрізняються від очікуваної поведінки програми.

Динамічні методи виявлення передбачають дослідження поведінки програми під час її реального або симульованого виконання в контрольованому та найчастіше ізольованому середовищі. Такі підходи дозволяють виявляти приховані загрози, що не проявляються під час статичного аналізу, але демонструють зловмисну поведінку під час фактичного виконання. Ці методи особливо ефективні проти ЗПЗ, які використовують поліморфізм, шифрування та складні техніки уникнення виявлення [63]. До найпоширеніших методів належать: аналіз поведінки в пісочниці; моніторинг системних викликів; аналіз пам’яті; моніторинг реєстру та файлової системи; трасування викликів API; моніторинг продуктивності. Розглянемо їх суть та особливості.

Аналіз поведінки в пісочниці передбачає виконання програми ізольованому середовищі з детальним моніторингом її активності. Досліджуються системні виклики, зміни в реєстрі, файлові операції, мережева активність тощо [41]. Такий підхід дозволяє отримати цілісну картину поведінки програми та виявити приховані зловмисні дії, які неможливо виявити статичним аналізом.

Моніторинг системних викликів здійснює спостереження за всіма запитами програми до операційної системи, аналізує їх параметри та послідовність виконання

[112]. Особлива увага приділяється рівню доступу й привілеям, які використовує програма. Це дозволяє не лише виявляти окремі підозрілі дії, а й розпізнавати складні зловмисні сценарії.

Аналіз пам’яті виконується в процесі роботи програми та досліджує вміст оперативної пам’яті динамічні зміни в структурі процесу, завантажені модулі та бібліотеки. Такий аналіз дозволяє зафіксувати стан програми в момент виконання й виявити зловмисний код [25], який може тимчасово розшифровуватись лише в пам’яті та бути відсутнім у статичному файлі на диску.

Моніторинг реєстру та файлової системи передбачає відстеження модифікацій у реєстрі та файловій системі [6]. Аналізуються шаблони доступу, створення та редагування файлів, особливо в незвичних місцях або із високою частотою. Також виявляються потенційно небезпечні дії, такі як зміна системних файлів або швидке шифрування даних.

Трасування викликів API зосереджується на аналізі функцій API, які викликає програма. Це дозволяє точно ідентифікувати використані функції операційної системи [76] та виявити такі зловмисні дії, як викрадення даних, встановлення з’єднань із віддаленими серверами або маніпуляція іншими процесами.

Моніторинг продуктивності аналізує споживання ресурсів системи, зокрема завантаженість процесора, використання пам’яті та інтенсивність операцій введення-виведення. Виявлення аномальних патернів [99], таких як різке збільшення споживання ресурсів, може свідчити про наявність ЗПЗ, що виконує дії які призводять до дестабілізації системи, бере участь у DDoS-атаках або здійснює іншу зловмисну активність.

Аналіз сучасних засобів виявлення загроз свідчить про активне використання та поєднання обох підходів одночасно, що загалом формує гібридний (комбінований) метод виявлення. На рис. 1.2 графічно представлено основні методи виявлення ЗПЗ.

Рисунок 1.2 – Класифікація підходів до виявлення ЗПЗ

На рис. 1.2, також, визначено деякі ключові гібридні методи, зокрема такі: машинне навчання; кореляційний аналіз. Обидва орієнтуються на пошук збігів результатів статичного й динамічного аналізу [108], які свідчать про зловмисну поведінку. Машинне навчання в такому контексті опирається на велику базу даних уже досліджених зразків ЗПЗ, дозволяючи виявляти складні патерни, тоді як кореляційний аналіз використовує сформовану базу знань із типових статичних та динамічних ознак, комбінації яких уже раніше призводили до виявлення ЗПЗ.

Підходи статично-спрямованого динамічного тестування та статичний аналіз із урахуванням динамічних результатів передбачають використання обох видів аналізу, проте в різному порядку. Перший підхід ґрунтується на попередньому статичному аналізі, завдяки якому визначають потенційно небезпечні аспекти програми й надалі цілеспрямовано перевіряють їх динамічними методами для підтвердження зловмисного прояву [8]. Другий же починається з динамічного аналізу, мета якого –

виявити приховані артефакти у пам’яті (наприклад, розпакований зловмисний код), що потім передаються на поглиблений статичний аналіз. Це дає змогу виявити ті елементи коду, які не проявляються під час первинного статичного аналізу.

Гібридний аналіз з профілюванням поведінки фокусується на тому, що саме програма робить під час виконання в операційній системі. У процесі аналізу фіксуються дії, характерні для ЗПЗ, на основі яких формується поведінковий профіль [79]. Далі отриманий профіль порівнюється з уже відомими зразками, що мали зловмисний характер. Загалом цей підхід дозволяє виявляти ЗПЗ навіть у разі сильно заплутаного коду. Крім того, метод профілювання дає змогу класифікувати виявлене ЗПЗ, що сприяє подальшому вдосконаленню методів виявлення нових загроз.

Усі розглянуті підходи є частинами організаційного підходу, який зазвичай визначається як багатоступеневий конвеєрний аналіз, в якому множина визначених методів аналізу поєднані в єдиний набір дій для опрацювання файлу. Класичний підхід передбачає виконання спочатку швидких видів аналізу, і при недостатній визначеності результату застосувати більш комплексні. Така схема передбачає, що щойно отримане достовірне визначення наявності зловмисного прояву визначає, що подальші етапи можна не виконувати, що економить як час так і обчислювальні потужності.

1.3. Застосування розподілених систем для виявлення інфікованих виконуваних програм з поліморфними властивостями

Розподілені обчислювальні системи застосовуються для розв’язання широкого спектру наукових і практичних завдань [48], зокрема моделювання складних фізичних процесів, оцінки ризиків, молекулярного моделювання, збору та обробки великих обсягів даних у системах Інтернету речей (IoT), тренування моделей штучного інтелекту тощо [29,74]. З огляду на постійне зростання складності атак, необхідні більш ефективні підходи до аналізу потенційних загроз. Успішний досвід використання розподілених систем у різних сферах сприяв їх активному залученню [58] фахівцями з кібербезпеки для підвищення ефективності виявлення існуючих і

нових загроз у сфері ІТ [94, 69]. Розподілені системи мають низку важливих переваг, які допомагають підвищити точність, швидкість і надійність процесу виявлення ЗПЗ [98, 56].

Зокрема, такі системи дозволяють детально аналізувати підозрілі програми [57]. Інфраструктура обчислювальної системи, що складається з множини вузлів [53], дає змогу пришвидшити виявлення нових загроз завдяки використанню великої кількості інструментів і методів аналізу, а також ізольованих середовищ [64]. Це забезпечує різнобічне дослідження ЗПЗ з глибоким логуванням виконуваних дій, включаючи мережеві запити, зміни у файловій системі, використання ресурсів тощо.

Окрім цього, останнім часом обчислювальні системи активно застосовують для обробки великих обсягів даних, необхідних для навчання моделей машинного навчання та штучного інтелекту. Використання різнорідних джерел даних сприяє підвищенню точності моделей, а розподілена інфраструктура дає змогу паралельного обробляти й аналізувати великі датасети, що суттєво пришвидшує процес навчання. Крім навчання моделей, обчислювальні потужності розподілених систем залучають для аналізу та моніторингу трафіку в корпоративних мережах. Динамічне горизонтальне масштабування обчислювальних компонентів дозволяє адаптувати систему до поточних навантажень, забезпечуючи ефективну реакцію на потенційні загрози. Завдяки цим можливостям розподілені системи можуть функціонувати не лише як окремий сервіс для підвищення рівня безпеки, а й інтегруватись в існуючі корпоративні засоби захисту [66], зокрема в SIEM-системи, системи управління подіями безпеки, міжмережеві екрани тощо.

У цілому, розподілені системи відкривають широкі можливості для виявлення різноманітних загроз. Їх здатність пришвидшувати обробку даних сприяє скороченню часу реакції на загрози, а також дозволяє накопичувати й інтегрувати знання про них в єдину систему. Такий підхід сприяє створенню більш надійної, гнучкої та масштабованої інфраструктури кібербезпеки, здатної ефективно протидіяти сучасним викликам у сфері ІТ [71].

З моменту початку використання обчислювальних систем для розв’язання різних завдань науковці запропонували низку способів організації об’єднаних

обчислювальних елементів для виконання спільної задачі. Таке об’єднання та його організація функціонування визначало створення централізованої системи. Усі запропоновані підходи враховували різні аспекти підключення елементів, їх автономність, використання інтерфейсів передачі даних, а також загальну організацію та функціонування системи. На сьогодні популярністю користується кілька типів таких систем, кожен із яких має свої особливості та, відповідно, набір завдань, для яких вона найкраще підходить. До основних типів розподілених обчислювальних систем, що мають практичне застосування, належать такі: кластерні; грід; граничні; хмарні.

Кластерні системи обчислень становлять одну із ключових моделей реалізації розподілених обчислень. Вони забезпечують об’єднання значних обчислювальних ресурсів [118] шляхом використання однорідних обчислювальних вузлів. Усі компоненти кластеру фізично близько розташовані та з’єднані високошвидкісними провідними інтерфейсами передачі даних, такими як Infiniband, що сприяє ефективному обміну великими обсягами інформації. Завдяки можливості легкого додавання нових вузлів такі системи характеризуються хорошою масштабованістю. Надійність досягається завдяки можливості оперативної заміни несправних елементів без зупинки обчислювального процесу. Це робить кластерні архітектури придатними для виконання складних обчислювальних задач, що потребують значних ресурсів. Керування процесами обчислень і перевірка їх коректності здійснюється централізовано. Крім того, у таких системах широко застосовуються засоби віртуалізації для оптимального розподілу ресурсів між окремими незалежними задачами.

Грід-обчислювальні системи об’єднують розподілені обчислювальні елементи [113], які можуть знаходитися в різних географічних регіонах. На відміну від кластерних систем, у яких переважають однорідні компоненти, грід-архітектури зазвичай включають різнорідні обчислювальні ресурси, які відрізняються як за апаратним забезпеченням, так і за встановленим програмним середовищем. Саме ця особливість зумовлює їхню класифікацію як гетерогенних обчислювальних систем. Для забезпечення взаємодії між вузлами у грід-середовищах найчастіше

застосовуються стандартні мережеві технології, зокрема протоколи TCP/IP. Завдяки такій мережевій гнучкості грід-системи легко масштабуються, дозволяючи динамічно підбирати та залучати ресурси відповідно до поточних вимог обчислювальних задач. Важливою характеристикою таких систем є високий рівень автономності їхніх компонентів: кожен обчислювальний елемент може незалежно приймати рішення щодо доцільності участі в обробці конкретного завдання, а також визначати обсяг ресурсів, які будуть надані. Це забезпечує гнучке, адаптивне та ефективне використання наявних обчислювальних потужностей, особливо в умовах динамічної зміни навантаження та доступності ресурсів.

Хмарні обчислювальні системи представляють собою сукупність інформаційно-технічних засобів, основним призначенням яких є надання користувачам доступу до обчислювальних ресурсів, програмного забезпечення, а також сервісів для зберігання та обробки даних. Ключовою характеристикою таких систем є можливість надання послуг у віддаленому режимі через мережу Інтернет без необхідності у фізичному володінні апаратним забезпеченням [9]. Завдяки цьому користувачі отримують змогу швидко масштабувати ресурси відповідно до поточних потреб, що особливо актуально для задач з динамічними вимогами до обчислювальної потужності. Хмарні сервіси можуть впроваджуватись у різних формах залежно від організаційних і технічних вимог [46]. Виділяють кілька моделей розгортання: публічна хмара, що забезпечує доступ до ресурсів усім зареєстрованим користувачам; приватна хмара, яка функціонує виключно на обладнанні однієї організації та відзначається високим рівнем контролю та захисту; спільна хмара, що об’єднує інфраструктуру кількох організацій з метою економії витрат та підвищення ефективності; гібридна хмара, яка поєднує в собі переваги попередніх моделей, дозволяючи гнучко управляти навантаженням та ресурсами. Залежно від рівня наданого контролю над інфраструктурою, користувачі можуть працювати з хмарними системами в межах трьох основних сервісних моделей: IaaS (Infrastructure as a Service) – модель, за якої користувачу надається доступ до базових обчислювальних ресурсів, таких як віртуальні машини, дискові сховища, мережеві налаштування та операційні системи; PaaS (Platform as a Service) – забезпечує не лише

інфраструктуру, а й готові середовища для розробки, бази даних, інструменти тестування та розгортання програмних рішень; SaaS (Software as a Service) – надає доступ до готових прикладних сервісів, що працюють через веб-інтерфейс, без потреби встановлення програмного забезпечення на локальні пристрої. Для забезпечення стабільного та безпечного функціонування хмарних обчислювальних систем необхідно вирішити низку ключових завдань. Серед них: організація надійного захисту даних та обчислень, а також розробка й впровадження ефективних механізмів передачі великих обсягів інформації з мінімальними затримками та втратами.

Граничні обчислення є порівняно новою обчислювальною парадигмою, що сформувалася у відповідь на стрімке зростання кількості IoT-пристроїв та сенсорних систем у повсякденному житті й промислових застосуваннях [85]. Основна ідея цього підходу полягає у зміні традиційної моделі обробки даних: замість централізованого зберігання та аналізу інформації, значна частина обчислювальних процесів переноситься ближче до джерел її генерації – безпосередньо на пристрої або у безпосередньо пов’язані з ними вузли. Швидке збільшення кількості IoT-пристроїв, таких як сенсори, камери спостереження, медичні прилади тощо, спричиняє експоненційне зростання обсягів даних, що необхідно обробляти для прийняття рішень або ініціювання дій. Передача всіх цих даних до віддалених центрів обробки створює відчутні затримки, пов’язані з обмеженою пропускною здатністю мережі та необхідністю централізованої обробки. Граничні обчислення вирішують цю проблему шляхом використання проміжних обчислювальних ресурсів, які здатні частково або повністю обробляти дані локально, перш ніж вони будуть надіслані до центрального вузла. Такий підхід є особливо важливим для систем, що функціонують у реальному часі або наближених до нього умовах [121], де навіть незначні затримки можуть мати критичні наслідки. Завдяки розміщенню керуючих та обчислювальних компонентів ближче до джерел даних, можливо швидко отримувати узагальнену інформацію про поточний стан певної ділянки чи процесу, виконуючи лише один локальний запит до керуючого елемента відповідного рівня. Граничні обчислення знаходять широке застосування у багатьох галузях, серед яких: інтелектуальні

системи на базі IoT, охорона здоров’я та дистанційна медицина, автономні транспортні засоби, сучасні телекомунікаційні інфраструктури, системи управління енергетикою, промислове виробництво тощо. Водночас ця парадигма висуває низку нових викликів, серед яких особливу увагу привертають: забезпечення надійного та безперебійного мережевого з’єднання, захист конфіденційних даних на етапах збирання, обробки та передавання, а також організація ефективного розподіленого управління обчислювальними ресурсами в умовах мінливого середовища.

Серед усіх розглянутих способів організації розподілених систем, можна визначити, як перспективні для розгляду, саме грід-системи, оскільки їхня властивість залучати велику кількість обчислювальних елементів потенційно дозволяє акумулювати значні обчислювальні ресурси. Прикладом таких систем є волонтерські системи, які залучають пристрої охочих користувачів на добровільних засадах. Такі системи використовуються для розв’язання наукових і технічних задач [99], що потребують значних обчислювальних ресурсів, наприклад BOINC, Folding@home, Einstein@home, де обчислення розподіляються між великою кількістю незалежних учасників. Завдяки цьому підходу стає можливим ефективне використанні ресурсів звичайних користувачів для моделювання складних процесів, обробки великих обсягів даних чи навіть пошуку оптимальних рішень прикладних задач різних наукових напрямків. З використанням цих систем проекти досягали пікових значень обчислювальної спроможності. Наприклад, Folding@home на початку 2020 року акумулював обчислювальні ресурси, що вп’ятеро перевищували потужність найпотужнішого на той час суперкомп’ютера IBM Summit [5]. Така обчислювальна потужність дала змогу науковцям виконувати розрахунки біохімічної науки. Seti@home відзначився внеском в аналіз радіосигналів з космосу, залучивши понад 5.2 мільйона людей у всьому світі [37]. На піку своєї продуктивності ця система досягала рівня потужності тогочасних суперкомп’ютерів. Einstein@home відзначилася численними успіхами наукових відкриттів в астрофізиці, зокрема виявленням нових пульсарів або скануванням записів детекторів LIGO [107] для пошуку періодичних сигналів. Метою проекту Climateprediction.net є дослідження та моделювання кліматичних змін, а також визначення діапазону їхніх можливих

проявів. Усі ці проекти суттєво вплинули на розвиток науки і довели важливість використання моделі грід-обчислень для наукових потреб. Такий підхід не лише прискорює у дослідженнях, а й підвищує обізнаність суспільства про науку, дозволяючи кожному зробити внесок у вирішення глобальних наукових завдань.

Попри значні переваги використання грід-систем обчислень, їхня ефективна реалізація вимагає ретельного врахування низки критичних факторів. До ключових завдань належать організація розподілу обчислювальних задач між учасниками, забезпечення надійності результатів обчислень та розробка відповідного програмного забезпечення. Усі ці завдання зумовлені динамічним характером [72] обчислювального середовища, що формується в грід-системах, через автономність залучених обчислювальних елементів. Кожен елемент має право самостійно визначати обсяг та час надання своїх ресурсів для виконання поставлених задач. Це створює виклик для системи, яка повинна автоматично адаптуватись до постійно мінливого стану доступних обчислювальних ресурсів.

Непередбачуваність поведінки обчислювальних елементів унеможливлює планування обчислень на декілька ітерацій наперед, оскільки центральний сервер не може гарантувати стабільність архітектури системи. Крім того, відсутність прямого контролю над кожним обчислювальним пристроєм, що бере участь в обчисленнях, створює додаткові виклики захисту результатів [3]. Система не може перевірити або контролювати апаратне та програмне забезпечення, що використовується, а також його зміни, що підвищує ризик спотворення результатів.

Враховуючи, що обчислення зазвичай базуються на розподілі малих задач, а загальний результат формується з їхніх окремих результатів, спотворення навіть одного результату може звести нанівець усі попередні обчислення. Такі спотворення можуть виникати з різних причин, зокрема через наявність ЗПЗ на пристроях користувачів. Для забезпечення широкої сумісності ПЗ необхідно враховувати різноманітність операційних систем, що використовуються користувачами, та застосовувати кросплатформені рішення [73]. Успішне вирішення цих завдань щодо ефективного використання акумульованих обчислювальних ресурсів дасть змогу досягти поставлених наукових цілей.

Вирішуючи питання розподілу завдань у волонтерських обчислювальних системах, було реалізовано та впроваджено низку алгоритмів, які враховують як особливості системи, так і специфіку обчислювальних завдань. В умовах динамічної архітектури системи використання статичних алгоритмів є недоцільним, оскільки вони не враховують можливість від'єднання обчислювальних елементів під час роботи. Водночас деякі напрацювання, які реалізовані в таких алгоритмах, можуть бути корисними за умови внесення додаткових модифікацій. Одним із класичних прикладів таких алгоритмів є множина жадібних алгоритмів: min-min; min-max; max-min; max-max. Усі вони оперують двома основними характеристиками: приблизним часом виконання завдання; продуктивністю обчислювального елемента.

Min-min надає пріоритет найшвидшим завданням, що є корисним у випадках, коли набір задач містить велику кількість простих обчислень. Цей алгоритм дозволяє швидко завершувати прості завдання [17], що може підвищити загальну ефективність розподілу ресурсів.

Min-max орієнтується на максимальне використання всіх доступних ресурсів системи. Однак такий підхід може уповільнювати виконання робочої ітерації, якщо набір завдань є неефективним з точки зору балансу навантаження [36].

Max-min розподіляє найскладніші завдання в першу чергу, що робить його ефективним у ситуаціях [86], коли складність задач є непередбачуваною. Це дозволяє уникнути затримок у виконанні найбільш ресурсномістких обчислень.

Max-max шукає найскладніше завдання та призначає його обчислювальному елементу з найнижчою продуктивністю. Через це його використання є досить ситуативним, і він застосовується рідко, оскільки може значно знизити ефективність розподілу ресурсів.

Окрім цього, ці алгоритмів мають модифікації, які дозволяють адаптувати їх до конкретних умов. До таких модифікацій належать динамічна адаптивність, оптимізація навантаження [27, 102], дублювання завдань, енергоефективність [60], обмеження за часом виконання та пріоритетний розподіл.

Також, в подібних системах знаходять застосування і інші підходи. Зокрема, алгоритми глибокого навчання, що вимагають великих обсягів даних для тренування

нейронних мереж, демонструють високу точність прогнозування та оптимізації. Еволюційні алгоритми [80], що базуються на принципах природнього відбору, шляхом ітеративного процесу мутації та схрещувань, знаходять оптимальні рішення в складних умовах. Ройові алгоритми [17], імітуючи колективну поведінку соціальних комах, забезпечують децентралізоване та адаптивне балансування навантаження. Стохастичні методи, використовуючи елементи випадковості, допомагають уникати локальних мінімумів [116], що є критично важливим для пошуку оптимального розподілу. Нарешті, баєсові алгоритми, що базуються на ймовірнісних моделях, дозволяють динамічно оновлювати прогнози продуктивності ресурсів, враховуючи нові дані, що надходять в реальному часі.

З точки зору безпеки обчислень для грід-обчислювальних систем характерним є використання моделі довіри. Ця концепція базується на визначенні рівня довіри до кожного учасника системи, якому належать її обчислювальні ресурси. Довіра допомагає оцінювати надійність обчислювальних елементів на основі їхньої історії виконання завдань, відповідності очікуваним результатам, часу роботи без збоїв тощо. Це дозволяє ефективно контролювати та перевіряти виконані обчислення на кожному елементі, запобігаючи помилкам і зловживанням, а також мінімізувати вплив скомпрометованих елементів на всю систему, що є критично важливим для її стабільності та надійності.

Модель довіри в контексті розподіленої обчислювальної системи представляється у вигляді програмного модуля, призначеного для управління довірчими відносинами між елементами системи та встановлення рівня довіри [22]. Цей рівень є ключовим фактором у контролі доступу до ресурсів. Саме модель довіри забезпечує правильність обчислень, в умові коли центральний сервер не може повністю покладатись на жоден з обчислювальних елементів. Моделі довіри в розподілених системах [62] використовують різноманітні підходи для оцінки надійності учасників. Репутаційний підхід оцінює елементи на основі минулих успіхів, тоді як голосування використовує колективне рішення для визначення довіри. Блокчейн забезпечує безпеку та прозорість через децентралізований реєстр, а теорія ігор встановлює правила співпраці. Поведінковий аналіз вивчає поведінку, а

машинне навчання використовує алгоритми штучного інтелекту для прогнозування надійності [65]. Ці методи дозволяють ефективно контролювати обчислення та запобігати зловживанням, забезпечуючи безпеку та стабільність системи. Багато сучасних моделей інтегрують кілька цих аспектів для підвищення ефективності, враховуючи специфіку системи [104]. Наприклад, основний механізм голосування може доповнюватись машинним навчанням та поведінковим аналізом для вирішення конфліктів, що виникають при розбіжності результатів обчислень. У таких випадках система використовує рейтинг, заснований на історії та поведінці елементів, щоб забезпечити правильність обчислень.

1.4. Постановка задачі дослідження

Для розв’язання задачі покращення ефективності функціонування грід-обчислювальних систем із автономними обчислювальними елементами для виявлення зловмисної передачі управління головним потоком в інфікованих програмах, що базується на концепції динамічного стану емулювання середовища відтворення програмних засобів, потрібно вирішити такі завдання:

1) провести аналіз методів організації та функціонування грід-обчислювальних систем, дослідити методи оцінювання ефективності залучення обчислювальних елементів із рівнем автономії та їх безпечного і коректного функціонування для організації виявлення інфікованих програм;

2) розробити формальний опис та моделі інфікованих програм, що використовують методи уникнення виявлення, а саме ідентифікацію/визначення емульованого середовища виконання та обфускацію коду виконання, що дозволить відтворити їх поведінку для опису їх впливу на цільову систему користувача;

3) розробити та представити формальний опис функціонування і модель грід-обчислювальних систем із використанням автономних обчислювальних елементів, що повинна враховуватись при організації захищеного та правильного (коректного) виявлення зловмисної поведінки в інфікованих програмах;

4) розробити метод синтезу засобів формування шаблонів поведінки інфікованих програм, які використовують методи уникнення виявлення, що залучають базові пісочницю та емулятор використовуючи множини станів емульованих центральних процесорів для виявлення зловмисної поведінки;

5) удосконалити метод організації грід-обчислювальних систем для оптимального розподілу завдань між залученими автономними гетерогенними обчислювальними елементами для виявлення зловмисної поведінки в інфікованих програмах;

6) удосконалити метод оцінювання довіри автономних обчислювальних елементів для оптимізації використання обчислювальних ресурів шляхом скорочення кількості повторних обчислень в грід-обчислювальних системах;

7) розробити грід-обчислювальну систему виявлення інфікованих програм, які використовують методи уникнення виявлення, для експериментального дослідження з метою покращення її характеристик та її подальшого впровадження для практичного застосування.

1.5. Висновки до першого розділу

Аналіз існуючих комерційних ЗПЗ програмного забезпечення показав, що застосування технологій емуляції та пісочниці є провідним підходом, однак потребує значних обчислювальних ресурсів. Цю проблему можливо вирішити шляхом використання грід-обчислювальних систем, які забезпечують доступ до великої кількості ресурсів. Водночас такі системи потребують впровадження спеціалізованих методів, спрямованих на подолання недоліків, притаманних автономним обчислювальним елементам, що, у свою чергу, забезпечить ефективне та надійне функціонування системи в цілому.

Основні результати розділу опубліковані у працях [88-89, 124, 126, 128, 129]

РОЗДІЛ 2.
СИСТЕМА ДИНАМІЧНОГО ЕМУЛЮВАННЯ ДЛЯ ВИЯВЛЕННЯ ПЕРЕДАЧІ УПРАВЛІННЯ У ПРОГРАМАХ ІНФІКОВАНИХ ЗЛОВМИСНИМ ПРОГРАМНИМ ЗАБЕЗПЕЧЕНЯМ

Використання ЗПЗ як засіб інфікування реального ПЗ, є одним із поширених методів розповсюдження зловмисного коду серед ПП. Сформоване ІП набуває додаткових властивостей, які виконують низку важливих функцій та допомагають як уникати виявлення з боку АПЗ, так і відтворювати код, що містить зловмисну активність. Особливу увагу приділяють методам уникнення виявлення, які мають широку класифікацію і базуються як на методах аналізу системи користувача, так і на методах впливу на неї. А комбінація цих методів, суттєво ускладнює процес виявлення ІПЗ.

Як ПЗ, так і ІП на ПП користувача представлено у вигляді множини файлів, що містять набори низькорівневих інструкцій, які передаються на виконання апаратною частиною. Виконання цих інструкцій змінює стан компонентів центрального процесора (ЦП), формуючи тим самим певну поведінку програми. У провідних наукових дослідженнях та практичному застосуванні технології емуляції та пісочниці розглядаються як основні та найефективніші методи дослідження поведінки виконання зразків ЗПЗ різного типу.

Використання технології емуляції та ізольованих середовищ накладає додаткові обчислювальні витрати. Враховуючи те, що кількість нових зразків ЗПЗ зростає, то актуальним буде застосувати КС для організації їх аналізу. РСО є окремим випадком КС, та характеризується централізованою архітектурою. Ключовими елементами РСО визначають центральний сервер (ЦС) та набір обчислювальних елементів ОЕ. Одним із поширених варіантів реалізації РСО є грід-обчислювальні системи, які використовують гетерогенні та автономні ОЕ. Серед їхніх основних переваг визначають легкість масштабування та відмовостійкість. Така КС зможе організувати використання ОЕ для проведення аналізу ІП.

2.1. Моделі, властивості та особливості виконання інфікованих програм

Клас загроз, що класифікуються як ІП представляють собою одну із ключових загроз, з якими стикаються користувачі в ІТ. Розробники ЗПЗ використовують їх, для розповсюдження зловмисного коду під виглядом звичного для користувача ПЗ. Подібне маскування, дозволяє розробникам ЗПЗ сподіватись на успішне застосування ІП, що дозволить застосувати зловмисний код на ПП користувача. І, в залежності від реалізованого зловмисного коду, його несанкціоноване виконання дозволяє досягти різноманітних цілей, до яких можна віднести такі: крадіжка даних; порушення роботи ПП; отримання контролю над ПП.

Зважуючи на потенційну небезпеку, інженери з сфери кібербезпеки розробляють та вдосконалюють рішення для протидії подібним загрозам, що базуються на застосуванні апаратних, програмних засобів, та їх комбінації. Одним із найбільш популярних методів захисту від ІП визначають АПЗ, які комбінують статичні та динамічні методи аналізу програм, із метою виявлення зловмисної активності в них. На противагу засобам захисту, розробники ЗПЗ розробляють та використовують різні методи захисту від виявлення, що змінюють вигляд та поведінку програми в залежності від середовища її виконання. Комбіноване застосування таких методів ускладнює процес виявлення ЗПЗ.

Методи протидії емуляції визначають як одні із основних методів захисту ЗПЗ, так як їх основна задача – виявлення використання АПЗ на ПП користувача. Покладаючись на динамічний аналіз, більшість АПЗ використовують ізольовані середовища із емуляцією для аналізу виконання ІП, при цьому забезпечують захист ПП користувача. Тому, зловмисники ЗПЗ також можуть впроваджувати стратегії виконання програм, вибір яких буде залежати від результатів аналізу середовища виконання. І за допомогою цих стратегій, ІП буде змінювати свою поведінку, що збільшить шанси як на уникнення від виявлення, так і на успішне застосування зловмисної дії на ПП користувача.

Особливість використання ІП, як засобу виконання зловмисних дій на ПП користувачів передбачає вибір цільової програми для її інфікування. Зазвичай,

розробники ЗПЗ обирають вже існуючі та широко відомі програми. Подібні програми є добре впізнаваними серед користувачів, не викликають підозр, і тому збільшується імовірність їх виконання на їх ПП. Процедура інфікування може бути здійснена різними методами, що супроводжуються модифікацією коду оригінальної програми. У процесі такої модифікації зловмисники впроваджують зловмисний код, захисні механізми для обходу систем виявлення та визначають стратегії виконання програми. Тому, після завершення інфікування, програма набуває додаткових властивостей, основна мета яких – забезпечити успішне виконання зловмисної дії. Зазвичай, це відбувається завдяки використанню прихованих команд, що запускають на виконання зловмисного коду. А результатом виконання зловмисного коду уже буде несанкціоноване отримання конфіденційної інформації, втручання в систему або безпекові налаштування, тощо. Крім того, в ІП закладають можливості для самостійного розповсюдження через локальні мережі, що дозволяє зловмисникам розгорнути масштаб атаки.

Із зростанням проблеми широкого використання ІП та наслідками їх зловмисної дії, провідні фахівці ІТ з питань кібербезпеки почали детально досліджувати їх особливості функціонування. Тому, для протидії зловмисної дії почали розробляти та і застосовувати різні підходи їх виявлення, що представлені у вигляді програмних та програмно-апаратних рішень. Розробники ЗПЗ також активно досліджують нові методи виявлення, та на основі цих знань формують нові стратегії проникнення в ПП, які із певною вірогідністю можуть проходити повз наявні захисні рішення. У сучасних комп’ютерних системах, такі рішення представлені у вигляді АПЗ, що виконують аналіз та перевірку виконуваних програм на ПП користувача. Враховуючи це, можна зробити висновок, що одним із ключових підходів до успішного застосування ІП, є використання засобів уникнення від виявлення.

Методи уникнення виявлення застосовують різні підходи, які базуються як на апаратному, так і на програмному рівнях забезпечення. Вони розвивались дуже активно та динамічно впродовж еволюції засобів захисту для нових зразків ІП. Для виконання поставленого завдання, застосовується велика кількість методологій, на основі яких формуються підходи для аналізу ПП або впливу на нього. До основних

підходів відносять: протидія аналізу, протидія вивченню, протидія пісочниць, поліморфізм, застосування руткітів та приховування виконання.

Ці методи застосовуються для забезпечення модифікації поведінки ІП (рис. 2.1), а також для динамічної трансформації її структури під час виконання. Також, застосовують методи, в яких використовують спеціалізовані алгоритми для пошуку та видалення артефактів в оперативній пам’яті або залишкових процесів в комп’ютерних системах. Усі вони направлені на зменшення шансу виявлення за допомогою АПЗ, відповідно до її основних методів роботи. Використання таких методів захисту дуже важливе і тому, що попри велику кількість існуючих ІП, вони можуть застосовувати однакові вразливості ПП для проведення зловмисної дії. І якщо, хоча б один з таких ІП буде виявлено, інформація про використану вразливість буде додана у відповідні бази даних АПЗ. І в результаті, усі інші ІП, які використовують подібні вразливості, буде простіше виявити.

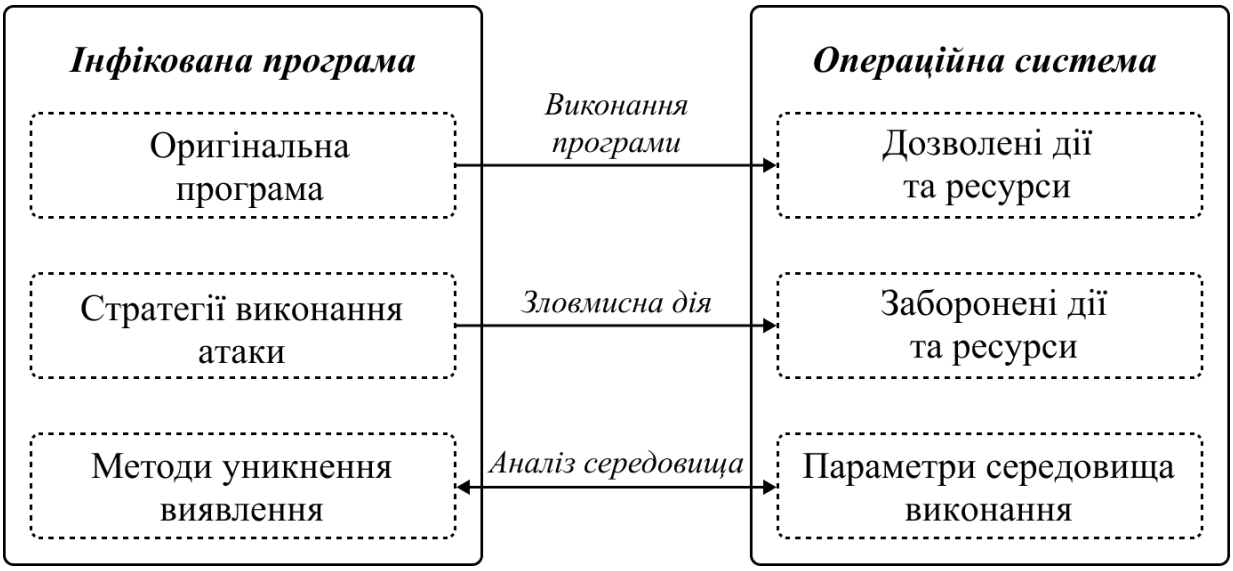


Рисунок 2.1 – Моделі інфікованих програм

Окрім методів уникнення від виявлення, зловмисники застосовують різноманітні стратегії виконання ІП. Вони можуть адаптувати процес виконання ІП в залежності від результатів аналізу системи за допомогою відповідних методів. Це дозволить обійти захист, який забезпечують АПЗ, системи виявлення вторгнень та інші засоби. Наприклад, застосування методів протидії емуляції дозволить визначити, чи потрібно приховувати свою діяльність. Адже зміна типу середовища виконання ІП

може свідчити про те, що АПЗ не виявило зловмисного прояву, і програма була ідентифікована як безпечна до застосування на ПП користувача.

У цілому, комбінація методів уникнення виявлення та стратегій виконання робить ІП більш стійкою до виявлення. Це дозволяє їй залишатися непоміченою протягом тривалого часу, значно підвищуючи ефективність у контексті реалізації зловмисних дій. Така прихована присутність у системі дає змогу здійснювати атаки більш приховано й ефективно. У результаті значно зростає ризик довготривалого компрометування системи та витоку критичної інформації.

Сучасні операційні системи часто використовують вбудоване або стороннє АПЗ для забезпечення захисту шляхом аналізу та моніторингу виконання всіх програм. Такі засоби є ключовими компонентами у забезпеченні конфіденційності та цілісності даних користувача. Переважна більшість АПЗ використовують технології пісочниць і емуляції як основні методи аналізу ІП. На основі отриманих результатів, АПЗ приймає рішення щодо безпечності запуску тієї чи іншої програми. Зважаючи на те, що більшість сучасних операційних систем за замовчуванням використовує АПЗ, застосування методів протидії емуляції у сучасному ЗПЗ є критичним для забезпечення його ефективного функціонування. Основна причина полягає в тому, що виявлення ознак аналізу або застосування емульованого середовища може свідчити про наявність процесу пошуку ЗПЗ в системі, що суттєво підвищує ймовірність його виявлення. Саме тому, розробники ІП активно досліджують питання виявлення емуляції, застосовуючи різні підходи, а потім активно використовують їх для уникнення виявлення. Виявлення емуляції можливе виконанню спеціальних перевірок, які включають пошук характерних файлів, процесів, інструкцій, а також аналіз характеристик апаратного забезпечення, специфіки функціонування операційної системи, часу виконання завдань і програмних артефактів. Деталізуємо та формалізуємо найбільш поширені методи протидії емуляції з метою їх врахування при розробленні нових методів виявлення ІП..

Зловживання CPUID позначимо $i_{ed_{ca}}$. ІП використовуючи cpuid, отримує детальну інформацію про поточний процесор і його виробника. Ця низькорівнева інструкція використовує вміст регістру EAX як вхідний параметр. Залежно від

переданого значення, cpuid повертає різну інформацію щодо процесора, яку розміщує у регістрах EAX, EBX, ECX та EDX. У випадку виконання в емульованому середовищі, значення, що повертаються, можуть відрізнятись від значень, отриманих на реальному апаратному забезпеченні. Ця відмінність дозволяє ІП виявити використання емуляції та, відповідно, змінити свою поведінку.

Семантичні атаки на центральний процесор позначимо $i_{ed_{sc}}$. Такий тип методів застосовує різні специфічні низькорівневі інструкції. До них відносять, інструкції які не мають належної відкритої документації, специфічні інструкції які працюють лише за певних умов та інструкції що виконуються із певними обмеженнями під час емуляції. Прикладами можуть бути, сімейство інструкцій MMX, різноманітні інструкції VMCALL для звернень до об'єктів віртуалізації, а також SMI інструкції для управління перериваннями. Усі описані інструкції поводять себе по різному, якщо порівнювати їх виконання у емульованому середовищі та на реальному апаратному забезпеченні.

Аналіз часових показників низькорівневих інструкцій позначимо $i_{ed_{it}}$. Методи, які направлені на аналіз часу виконання інструкцій, використовують інструкцію RDTSC. ІП, використовуючи такі методи, ініціює виконання підготовленого набору інструкцій, виконує замір його виконання, та аналізує із наявними даними. Ці методи покладаються на те, що емульоване середовище при виконанні деякого набору інструкцій буде мати деякі часові витрати. Значні відхилення від очікуваного часу вказують на здійснення емуляції.

Аналіз таблиць IDT та LDT позначимо $i_{ed_{ta}}$. ЗПЗ, що застосовує такий підхід використовує аналіз IDT (таблиця дескрипторів переривань) та LDT (локальна таблиця дескрипторів) за допомогою відповідних інструкцій SIDT та SLDT. Використовуючи ці інструкції, ЗПЗ має змогу отримати базову адресу IDT та селектор для LDT. В емульованому середовищі отримані адреси пам'яті будуть відрізнятись від адрес на реальному апаратному забезпеченні, що дозволить виявити емуляцію.

Зловживання затримками виконання програм позначимо $is_{ed_{id}}$. Основна мета таких методів полягає у затриманні виконання певної ділянки коду, яка зазвичай викликає набір інструкцій, що ініціюють зловмисні дії на хост. Реалізуючи такі

методи, розробники очікують, що інфікована програма точно буде аналізуватись, тому треба призупинити активність до того моменту, поки вона покине АПЗ. Такі методи реалізуються за допомогою різних інструкцій, що можуть створювати контрольовану затримку часу, наприклад NOP, HLT, PAUSE та RDTSC.

Аналіз часових характеристик хосту позначимо $i_{ed_{ht}}$. Такі методи направлені на аналіз загальних часових характеристик роботи хосту. Вони беруть до уваги загальний час роботи системи, та можуть ідентифікувати емуляцію, якщо він складає декілька секунд або хвилин. Сюди також відносять застосування мікро снів для виконуваних процесів із спеціалізованими таймерами. Ці таймери мають більшу точність, і результат заміру може виявити емуляцію у тому випадку, якщо пісочниця використовує спрощений варіант таймеру із метою оптимізації її роботи.

Перевірка апаратного забезпечення позначимо $i_{ed_{hc}}$. Такі методи орієнтуються на отримання іншої детальної інформації про апаратне забезпечення за допомогою мов програмування високого рівня. Це дозволяє отримати серійні номери та детальну інформацію таких пристроїв як: материнська плата, мережевого адаптера, інформацію про BIOS, конфігурацію пам’яті та інших периферійних пристроїв. Аналіз отриманої інформації у порівнянні із використанням реального обладнання може свідчити про застосування емуляції.

Пошук артефактів інспектування позначимо $i_{ed_{ia}}$. Метою таких методів є пошук та аналіз специфічних файлів, процесів, ключів реєстрів та системних артефактів на хості користувача. Серед файлів та процесів ІП може шукати ті, що належать різноманітним засобам аналізу або антивірусними програмами. Перевірка ключів реєстрів, також може зберігати інформацію про активність таких засобів, а аналіз системних файлів операційної системи може виявити незвичайні шаблони функціонування хосту. Зібрана інформація є основою для оцінювання можливості використання емуляції.

Нехай I_{ed} – це множина методів протидії емуляції, тоді усі розглянуті та проаналізовані методи задамо наступним чином:

$$I_{ed} = \{i_{ed_{ca}}, i_{ed_{sc}}, i_{ed_{it}}, i_{ed_{ta}}, i_{ed_{id}}, i_{ed_{ht}}, i_{ed_{hc}}, i_{ed_{ia}}\}, \quad (2.1)$$

де $i_{ed_{ca}}$ – елемент множини I_{ed}, який характеризує зловживання CPUID; $i_{ed_{sc}}$ – елемент множини I_{ed}, який характеризує семантичні атаки на центральний процесор; $i_{ed_{it}}$ – елемент множини I_{ed}, який характеризує здійснення аналізу часових показників низькорівневих інструкцій; $i_{ed_{ta}}$ – елемент множини I_{ed}, який характеризує здійснення аналізу таблиць IDT та LDT; $i_{ed_{id}}$ – елемент множини I_{ed}, який характеризує здійснення зловживання затримками виконання програм; $i_{ed_{ht}}$ – елемент множини I_{ed}, який характеризує здійснення аналізу часових характеристик середовища виконання; $i_{ed_{hc}}$ – елемент множини I_{ed}, який характеризує перевірку апаратного забезпечення; $i_{ed_{ia}}$ – елемент множини I_{ed}, який характеризує пошук артефактів інспектування.

Сформована множина методів протидії емуляції для приховування зловмисних програмних кодів в інфікованих програмах зазвичай використовуються на первинному етапі виконання ІП. Успішне виявлення емульованого середовища може задіяти різноманітні додаткові захисні засоби і реалізовані методи, одними із яких визначають як методи поліморфізму, за допомогою яких відбувається зміна ІП з точки зору структури її коду так і поведінки.

Застосування поліморфізму є одним із основних захисних механізмів для приховування інфікованих програм від виявлення шляхом аналізу програмного коду в ізольованому середовищі. Подібні методи досить часто використовуються з метою уникнення виявлення АПЗ, що використовують методи виявлення на основі сигнатур. Заради уникнення або зменшення шансу на виявлення ІП використовує ці методи, для зміни власної сигнатури та поведінки в рамках інфікованої програми. Це стає можливим завдяки модифікації інструкцій, що потім будуть передані на виконання операційною системою на апаратне забезпечення ПП. В результаті модифіковані версії інфікованих програм будуть мати властивість поліморфізму.

Окрім зміни поведінки, за допомогою таких методів є можливим організація передачі управління головного потоку виконання програми, що дозволить передати на виконання набір інструкцій, які містять зловмисний характер. Змінюючи порядок виконання, зловмисники зазвичай запускають фонові процеси в операційних

системах, а потім повертають потік виконання в звичний порядок. Використання інфікованої програми в такому варіанті, буде виглядати для користувача як звичайне виконання програми.

Ключовим напрямком методів для реалізації поліморфізму в програмах є зміна виконуваного коду за допомогою таких конкретних окремих особливостей поліморфізму як мутація, обфускація та шифрування. Ці методи виконуються автоматично, відповідно до визначених правил захисту від виявлення, що можуть бути пов’язані із аналізом середовища виконання. Враховуючи їх активне застосування в сучасних ІП, складність аналізу модифікованих програм, та потенційну небезпеку від них, розглянемо їх такі основні методи: методи використання додаткового коду; методи перепризначення регістру; методи заміни інструкцій; методи перемішування виконання підпрограм; методи транспонування коду; методи розділення коду.

Методи використання додаткового коду позначимо $i_{p_{ci}}$. Особливістю цих методів є використання додаткового коду для модифікації. Ці методи можуть бути поділені з врахуванням трьох основних стратегій, а саме: додавання інструкцій, що не впливає на результат виконання певної частини програми; додавання алогічних та беззмістовних інструкцій, наприклад у вигляді додаткових розгалужень або кроків алгоритмів, що ніколи не будуть виконані; комбінація обох вищезазначених підходів.

Методи перепризначення регістру позначимо $i_{p_{rr}}$. Використання таких методів передбачає перетворення ідентифікаторів констант, змінних та регістрів на інші подібні значення. В результаті перетворення, семантика зберігається. Такий підхід дозволяє зробити код менш передбачуваним, що у свою чергу ускладнює його виявлення або формування його поведінкового алгоритму.

Методи заміни інструкцій позначимо $i_{p_{ir}}$. Цим методам притаманне використання підмін певних інструкцій в загальному наборі виконуваних інструкцій. При цьому, замінені інструкції виконують такі самі дії що і оригінальні, і визначаються як інструкції-синоніми. Модифіковані програми, які створені за допомогою таких методів, зазвичай містять більшу кількість інструкцій.

Таким чином, визначені інструкції змінюють програмний код на рівні інструкцій, що призводить до зміни поведінки виконання самої програми. Хоча ці методи вважаються не складними, але їх комбінування дає необхідний результат зловмиснику. В Додатку А1 наявний приклад впливу таких методів на оригінальний програмний код.

Частим є, також, використання методів, які використовуються для модифікацій підпрограм (subroutines), що часто застосовуються у наборах інструкцій програм після її компіляції.

Методи перемішування виконання підпрограм позначимо $i_{p_{sr}}$. Вони характеризуються застосуванням на рівні виклику підпрограм. Розробники виділяють незалежні підпрограми, та змінюють порядок їх виклику. Така зміна ускладнює аналіз всієї програм та пошук поведінкових особливостей.

Методи транспонування коду позначимо $i_{p_{ct}}$. Ці методи характеризуються відносно нескладною формою застосування поліморфізму. Зазвичай, її використання пов’язане із зміною розміщення підпрограм в основній програмі. Розробники ЗПЗ часто вдаються до використання генераторів випадкових значень та інших різних математичних функцій для виконання зміни порядку інструкцій у вихідному коді.

Методи розділення коду позначимо $i_{p_{cs}}$. Основною метою таких методів є підміна вмісту розроблених підпрограм. При цьому їх функціональність залишається незмінною. При їх використанні часто виконується розбиття існуючого блоку коду на декілька дрібних підпрограм із відповідними переходами між ними та інструкціями оригінального коду.

Визначені методи поліморфізму на рівні підпрограм націлені на ускладнення при автоматизованому аналізу програм різноманітними АПЗ. В Додатку Г наведено приклад впливу застосування методів на основі поліморфізму на рівні підпрограм на зміну коду виконання оригінальної програми. При формуванні ЗПЗ, розробники зазвичай використовують декілька методів поліморфізму одночасно, що дає суттєвий результат в контексті приховування факту інфікування виконуваних програм.

Нехай I_p – це поліморфні техніки, тоді визначені методи для модифікації програмного коду виконання та зміни поведінки виконання, подамо так:

$$I_p = \{i_{p_{ci}}, i_{p_{rr}}, i_{p_{ir}}, i_{p_{sr}}, i_{p_{ct}}, i_{p_{cs}}\} \quad (2.2)$$

де $i_{p_{ci}}$ відображає методи використання додаткового коду; $i_{p_{rr}}$ відображає методи перепризначення регістру; $i_{p_{ir}}$ відображає методи заміни інструкцій; $i_{p_{sr}}$ відображає методи перемішування виконання підпрограм; $i_{p_{ct}}$ відображає методи транспонування коду; $i_{p_{cs}}$ відображає методи розділення коду.

Таким чином, методи поліморфізму застосовуються комплексно в рамках одного екземпляру ІП, та можуть мати різні варіації їх комбінації. Комбіноване застосування методі дозволяє значно ускладнити процес виявлення ІП за допомогою АПЗ.

Розробники ІП активно використовують поєднання методів базованих на поліморфізмі і методів протидії емуляції для уникнення їх виявлення АПЗ. Це дозволяє ІП динамічно адаптуватись до середовища, змінюючи свою структуру та поведінку, що ускладнює процес його аналізу. Такий підхід суттєво ускладнює процес виявлення ІП, оскільки вони можуть ефективно приховувати свою зловмисну поведінку. Сучасні системи антивірусного захисту застосовують гібридний аналіз, поєднуючи статичний та динамічний методи перевірок. Проте, техніки уникнення виявлення часто націлені на пошук вразливостей в цих методах, що дозволяє ІП успішно виконувати зловмисні дії.

Одним із важливих компонентів ІП є модифікації, що робить можливими виконання зловмисних дій. Одним із типових проявів таких дій є передача управління виконання головного потоку команди. Завдяки передачі, модифікована програма ініціює запуск зловмисних фонових процесів, а потім повертає виконання програм у нормальний стан. Враховуючи те, що ІП застосовують декілька методів одночасно для уникнення виявлення, а також знаючи їх особливості застосування можна сформувати стратегії виконання на ПП користувача. Ці стратегії будуть розглядаються в контексті двох розглянутих класів методів уникнення виявлення. Стратегію або розгортання виконання ІП подано у табл. 2.1. Показники для стратегій виконання ІП розглядатимемо в їх поєднанні. Таким чином, стратегія, яка

отримуватиме найбільше показників, тобто включених методів матиме найбільшу складність.

Таблиця 2.1

Стратегії виконання інфікованих програм із використанням методів уникнення виявлення емульованого середовища

Стратегія виконання	i_{st_p}	i_{st_r}	i_{st_p}	i_{st_c}	i_{st_p}	i_{st_a}	i_{st_p}	i_{st_h}
Виявлення емуляції	-	-	-	-	+	+	+	+
Використання поліморфізму	-	-	+	+	-	-	+	+
Зловмисна поведінка	-	+	-	+	-	+	-	+

Стратегію приховування позначимо i_{s_h}. Дана стратегія застосовується в той момент, коли ІП виявляє емульоване середовище, тоді застосовує поліморфну модифікацію коду та запускає на виконання код із зловмисними проявами.

Агресивну стратегію позначимо i_{s_a}. Агресивний режим застосовуються ІП, що виявляють емуляцію, але при цьому не мають чи не використовують методи поліморфізму і запускають виконання інфікованої програми.

Безпечну стратегію позначимо i_{s_m}. Подібні стратегії застосовуються ІП, що не використовують методи виявлення емуляції, але застосовують поліморфізм для модифікації коду перед його виконанням на хості користувача.

Звичайну стратегію позначимо i_{s_c}. Така стратегія притаманна ЗПЗ, яке не застосовує методи уникнення виявлення та передбачає виконання інфікованої програми на хості без її додаткової модифікації.

В усіх інших випадках, згідно представленої табл. 2.1, буде застосовуватись стратегія паузи, яку позначимо як i_{s_p}, під час якої ІП може виконувати перевірку середовища, але в цілому ніяк не проявляти зловмисної активності в системі. Щодо застосування методів базованих на технологіях поліморфізму, то вони також можуть застосовуватись, але лише для того, щоб змінювати власну сигнатуру, як захист від користувацьких перевірок ПП на наявність ІП на основі статичного аналізу. В цілому, такий режим застосовується, для того, щоб ІП було виконане в деяких умовах, що

забезпечить виконання зловмисного дії. Тому, нехай описані стратегії виконання для ІП, що застосовують методи протидії емуляції та поліморфізму це - I_s. То тоді розглянуті стратегії можемо задамо так:

$$I_s = \{i_{s_h}, i_{s_a}, i_{s_m}, i_{s_c}, i_{s_p}\}, \quad (2.3)$$

де i_{s_h} - стратегія приховування; i_{s_a} - агресивна стратегія; i_{s_m} - безпечна стратегія; i_{s_c} - звичайна стратегія, а i_{s_p} – стратегія паузи.

Враховуючи розглянуті особливості ІП, та їх проаналізовані методи уникнення виявлення, можемо узагальнено представити її модель наступним чином:

$$I_i = \{s, a, I_{ed}^i, I_p^i, I_s^i\}, \quad (2.4)$$

де I_i – певний екземпляр i ІП; s – цільова програма, яка була інфікована; a – зловмисна атака, що була додана зловмисником; I_{ed}^i, I_p^i – методи протидії емуляції та методи поліморфізму, що застосовується у ньому для уникнення виявлення I_s^i – набір стратегій виконання.

Будемо розглядати ІП, які використовують визначені методи, тому справедливими є такі співвідношення: $I_{ed}^i \in I_{ed}, I_p^i \in I_p, I_s^i \in I_s$. Окремі екземпляри ІП, враховуючи те, що вони не використовують усі методи уникнення від виявлення одночасно, то можемо подати, наприклад, наступним чином певний екземпляр i ІП: $I_i = \{s_i, a_i, \langle i_{ed_{ca}}, i_{ed_{ta}} \rangle, \langle i_{p_{rr}}, i_{p_{sr}} \rangle, \langle i_{s_a} \rangle\}$.

Таким чином, розроблені моделі, які використовують визначені методи уникнення виявлення і відповідають ІП з точки зору як поведінки, так і основним його компонентам. Розглянуті моделі враховують наявність цільової програми інфікування та програмного коду, що містить зловмисний прояв, а також комбінування методів уникнення від виявлення, і є основою для розроблення систем виявлення такого типу інфікованих програм.

2.2. Архітектура засобів формування поведінки виконання програми

Сучасні операційні системи на ПП користувачів використовують як вбудовані, так і додаткові методи захисту. Основним методом захисту є АПЗ, яке базується на статичному та динамічному аналізі. Для забезпечення такого аналізу АПЗ активно використовують технології пісочниці та емуляції програмного і апаратного забезпечення. Поєднання технологій є важливим та вкрай ефективним рішенням у питаннях виявлення ІП.

Використання емуляції дозволяє імітувати роботу апаратних та програмних засобів. Залежно від повноти та деталізації реалізації функцій, що емулюють ту чи іншу технологію, визначають повну або часткову емуляцію. Така реалізація потребує детального вивчення особливостей функціонування об’єкту емулювання, а реалізований засіб називають – емулятором. Одним із ключових елементів дослідження в рамках виявлення описаної моделі ІП є процесор. Для реалізації часткової емуляції процесору необхідно проаналізувати відомості про архітектуру процесору, його набір апаратних компонентів та особливостей їх взаємодії. Також, слід звернути увагу на процес виконання програм в операційній системі, подання інструкцій на процесор, та як ці інструкції впливають на процесор і його компоненти.

Окрім емуляції, АПЗ зазвичай використовують технологію пісочниці, щоб убезпечити ПП користувача від потенційного негативного впливу. Це досягається завдяки формуванню ізольованого середовища виконання підозрілих програм. Поєднуючи обидві технології, АПЗ запускає на виконання ІП в ізольованому середовищі із можливістю аналізувати процес його процес виконання. В такій конфігурації, пісочниця використовуючи емулятор, як засіб виконання інструкцій в середині ізольованого середовища може сформувати поведінку виконання програми базуючись на різних аспектах. По завершенню виконання програми та її аналізу пісочниця знищує ізольоване середовище. Такий підхід гарантує найбільшу ефективність із виявлення ЗПЗ згідно наукових та практичних досліджень.

Правильна реалізація процесу функціонування емулятора, в першу чергу залежить від дослідження та аналізу його основних компонентів та особливостей їх

функціонування, під час виконання поданих інструкцій. До основних компонентів відносять: регістри різного призначення; вказівник стеку; пам’ять; виконавчі блоки; блок керування; інтерфейс шини; порти введення та виведення. Усі ці компоненти задіяні в основних етапах обробки інструкції, зокрема: зчитуванні; декодуванні; виконанні.

Згідно аналізу визначених компонентів ЦП та особливостей їх функціонування, можна поділити їх на дві категорії, а саме: активні; пасивні. До активних будемо відносити такі, які змінюють свій стан під час виконання інструкцій. І саме аналіз зміни стану цих компонентів буде важливим в контексті виявлення наявної зловмисної активності в програмах. До таких компонентів віднесемо регістри різного призначення, а саме: основні регістри; регістри вказівники; строкові регістри; регістри загального призначення. Вони є важливою і невід’ємною частиною процесора, тому потребують розроблення для втілення в емуляторах.

Основні регістри позначимо множиною $R_b = \{r_{RAX}, r_{RBX}, r_{RCX}, r_{RDX}\}$. Регістри як елементи множини використовуються при виконанні основних арифметичних та логічних операцій, організації виконання циклів, виконанні складних математичних операцій, передачі аргументів функцій, операції вводу та виводу.

Регістри вказівники позначимо множиною $R_p = \{r_{RBP}, r_{RSP}\}$. Вони використовуються як базовий покажчик для посилань на локальні змінні, як вказівник на вершину стек, так і для керування динамічного розподілу пам’яті, що дозволить формувати локальні змінні.

Строкові регістри позначимо множиною $R_s = \{r_{RSI}, r_{RDI}\}$. Вони застосовуються як вказівники на вхідні та вихідні дані для деяких команд при переміщенні даних, а також для роботи із строковими даними.

Регістри загального призначення позначимо множиною $R_r = \{r_8 \dots r_{15}\}$. До них відносемо регістри, які не мають спеціального призначення, і їх основна задача це надання більших обсягів пам’яті для виконання більш складних операцій не втрачаючи ефективності виконання.

Окрім цього, сучасні процесори містять блоки пам’яті, які ще визначають як кеш-пам’ять різних рівнів L1, L2 та L3. Але, ці блоки в рамках виявлення ІП

включатись не будуть, враховуючи навіть те, що їх також можна визначити як активні. Основною причиною є те, що така пам’ять керується апаратною логікою і це відбувається автоматично без участі програмних функцій операційних систем. Процесори не мають інструкцій, які безпосередньо працюють із цим типом пам’яті. Ця пам’ять контролюється апаратними засобами ЦП, та буде наповнюватись даними із адресного простору на основі внутрішніх алгоритмів. Такий підхід дозволяє пришвидшувати процес виконання програм.

Натомість, запущені програми активно застосовують пам’ять програми, які незалежно та ізольовано зберігаються в оперативній пам’яті впродовж виконання програми. Для таких задач, у процесорі визначені відповідні інструкції. Оскільки пам’ять програми активно використовується при виконанні інструкцій, то аналіз її вмісту є важливим для виявлення зловмисної активності. Тому, визначимо ще один активний компонент пам’ять і позначимо його i_{mem}. Це буде загальна пам’ять програми, що зберігає великі обсяги даних у порівнянні із розміром регістрів. Вважатимемо, що в ній зберігаються проміжні дані для виконання наборів низькорівневих інструкцій процесора.

Щодо інших компонентів, то їх можна віднести до пасивних, а тому в контексті виявлення можуть бути проігноровані. Наприклад, виконавчі блоки характеризуються в цілому наявністю певного набору інструкцій, що можуть виконуватись процесором. І в рамках аналізу виконання програми, дослідження його поведінки є неважливим, враховуючи вплив реалізованих методів уникнення, які застосовуються в ІП.

Тому, якщо емулятор для виконання програм, що подаються у вигляді послідовного набору інструкцій, визначити як E, то його можна задати за формулою (2.5), беручи до уваги лише ті компоненти, які важливі з точки зору виявлення зловмисної активності.

$$E_c = \{R_b, R_p, R_s, R_r, \{i_{mem}\}\}, \tag{2.5}$$

де E_c – множина активних компонентів емулятора, елементами якої є множини регістрів та елемент пам’яті; R_b – множина основних регістрів; R_p - множина

регістрів вказівників; R_s – множина строкових регістрів; R_r – множина регістрів загального призначення; i_{mem} елемент, що характеризує загальну пам’ять програми та зберігає великі обсяги даних порівняно із розміром регістрів.

За виконання інструкцій в процесорі відповідає виконавчий блок, тому емуляція його роботи також важлива. Розроблений виконавчий блок забезпечує логіку виконання частини існуючих інструкцій. Всього розроблено декілька десятків інструкцій архітектури x86_64, що використовуються найчастіше. Так як, усі інструкції не були реалізовані, будемо вважати цю реалізацію частковою. Розроблені інструкції представлені у табл. 2.2 за категоріями.

Таблиця 2.2

Список розроблених інструкцій за категоріями

Арифметичні інструкції	add, sub, mul, imul, div, idiv, inc, dec, adc, sbb
Логічні інструкції	and, or, xor, not, shl, shr, sal, sar, rol, ror
Інструкції з плаваючою комою	fld, fstp, fadd, fsub, fmul, fdiv, fcom, fchs
Строкові інструкції	movsb, movsw, movsd, movsq, stosb, stosw, stosd, stosq, lodsb, lodsw, lodsd, lodsq, scasb, scasw, scasd, scasq
Інструкції потоку керування	jmp, je, jne, jg, jl, jge, jle, call, ret, loop, int, iret
Інструкції роботи із даними	mov, push, pop, lea, xchg, bswap, cmov
Системні інструкції	hlt, cpuid, rdmsr, wrmsr, in, out, cli, sti, lidt, sidt, lgdt, sgdt

Враховуючи завдання емулятора, визначимо наступні основні функції, які він повинен забезпечувати, а саме: e_{f_l}– завантажувати набор інструкцій на виконання, e_{f_e}– ідентифікація інструкції та подання її на виконання, e_{f_m}– моделювання роботи регістрів під час виконання інструкцій, e_{f_c} – очищення стану емулятора. Тоді, його основні функціональні особливості в контексті аналізу виконання програм, задамо так:

$$E_f = \{e_{f_l}, e_{f_e}, e_{f_m}, e_{f_c}\}, \quad (2.6)$$

де e_{f_l} – завантажувати набор інструкцій на виконання, e_{f_e} – ідентифікація інструкції та подання її на виконання, e_{f_m} – моделювання роботи регістрів під час виконання інструкцій, e_{f_c} – очищення стану емулятора.

Визначений набір функцій дозволить забезпечити базове та поступове виконання програм, які подаються у вигляді набору низькорівневих інструкцій. При цьому, емулятор буде імітувати роботу із регістрами та пам'яттю програми. Це дозволить сформувати середовище, для аналізу виконуваних програм, що дозволить виявити відмінності у виконанні тої чи іншої програми. А, такі відмінності можуть означати про наявність передачі управління виконання головної програми, для активації зловмисної активності.

Емулятор та його функціональні спроможності забезпечують поступове та емульоване виконання низькорівневих інструкцій. Відповідно до призначення інструкцій, їх виконання призводить до зміни стану активних компонентів емулятора. Проте, емулятор не визначає, яку саме програму слід виконувати, і також не здатний самостійно проводити аналіз. Для контролювання цим процесом застосовується пісочниця.

Розглянемо основні компоненти пісочниці, яку позначимо S_c, що необхідні для виконання поставленої задачі. Одним із ключових компонент визначимо блок комунікації, який позначимо s_{c_t} і через який користувач може надіслати ІП для подальшого аналізу його поведінки виконання. Наступним важливим компонентом, який позначимо $s_{c_{io}}$, є той, що відповідає за створення модифікованого ізольованого середовища. Модифікації середовищ будуть визначати характер поведінки виконання ІП, що дозволить виявити аномальну активність. А для забезпечення процесу виявлення, пісочниця застосовує компонент для виконання програми та її аналізу, який позначимо s_{c_a}, що використовує механізми для зчитування стану емулятора із певною періодичністю. Цей підхід також відповідає за накопичення інформації про стан емулятора впродовж виконання програми. Зібрані дані передаються до модуля формування звіту, який позначимо s_{c_r}, звідки відправляється об'єкту, що ініціював

перевірку. Варто зазначити, що пісочниця сама по собі не робить висновків щодо наявності зловмисної активності. Ця задача покладається на відповідний модуль АПЗ, який може бути представлено у вигляді окремої програми чи системи. Тому, представимо структуру пісочниці в контексті аналізу ІП на наявність зловмисної поведінки, так:

$$S_C = \{s_{c_t}, s_{c_{io}}, s_{c_a}, s_{c_r}\}, \tag{2.7}$$

де S_c – множина компонентів пісочниці; s_{c_t} – компонент, що відповідає за комунікацію; $s_{c_{io}}$ – компонент, що забезпечує модифіковане ізольоване середовище; s_{c_a} - компонент для виконання аналізу програми під час її виконання; s_{c_r} – компонент, відповідальний за формування звіту.

Архітектуру пісочниці, що використовує емулятор для аналізу ІП, який виконується в ізольованому середовищі та взаємодію її компонентів, подамо схемою на рис. 2.2.

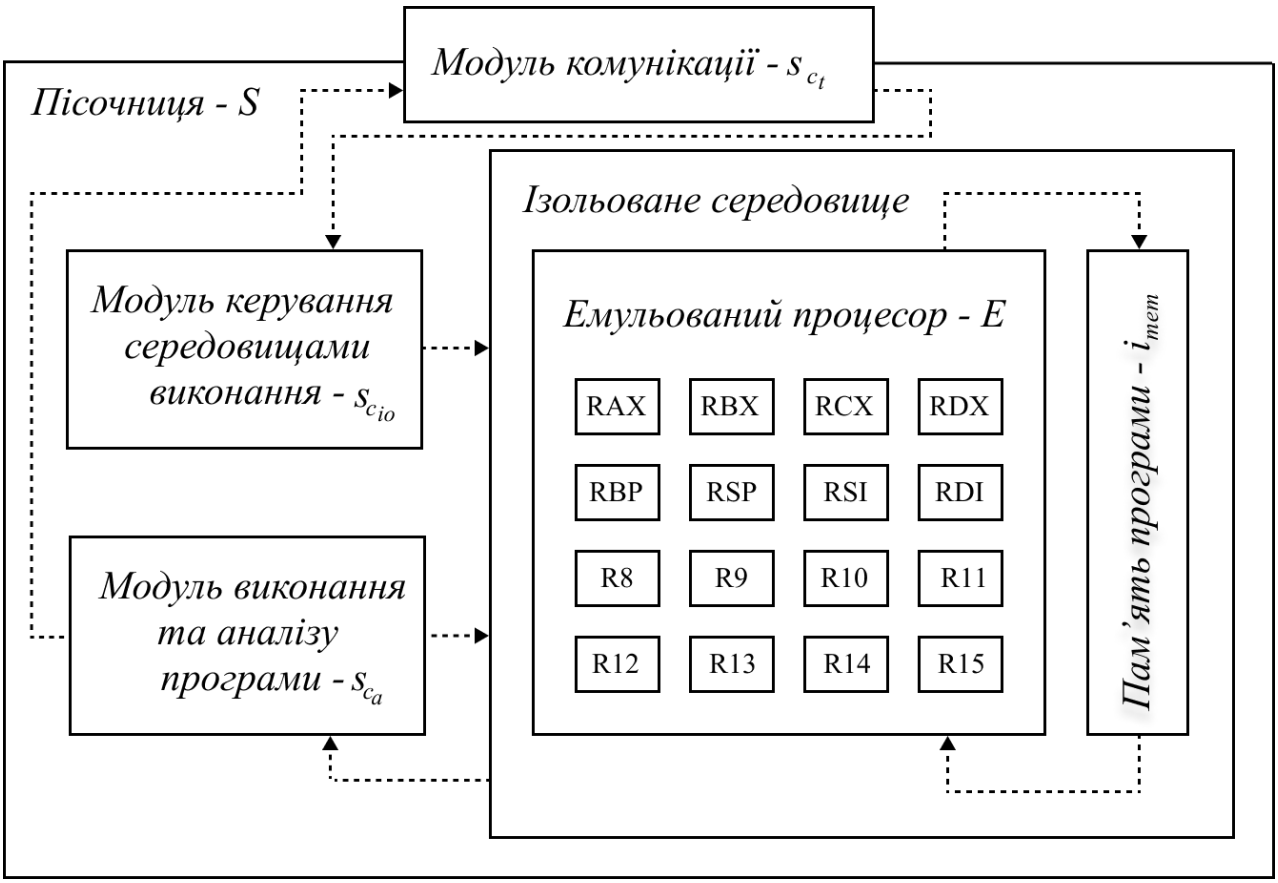


Рисунок 2.2 – Архітектура пісочниці для аналізу виконання програми

Зважаючи на основні модулі пісочниці та завдання, що на них покладаються, для забезпечення поставленого мети, визначимо її ключові функціональні особливості: отримання програми на перевірку; ініціалізація ізольованого середовища; передача на виконання отриманої програми; проведення аналізу виконання програми; формування звіту про аналіз.

Позначимо отримання програми на перевірку як s_{f_g}. Пісочниця отримує програму на аналіз від користувача, або від ЦС в залежності від реалізації.

Ініціалізацію ізольованого середовища позначимо s_{f_i}. Для безпечного проведення аналізу, пісочниця ініціює створення ізольованого середовища, що надає розширені можливості до аналізу та втручання в процес виконання програми.

Передачу на виконання отриманої програми позначимо s_{f_e}. Отримана програма передається в створене ізольоване середовище, де буде виконуватись за допомогою емульованого процесору.

Проведення аналізу виконання програми позначимо s_{f_a}. Використовуючи частковий доступ до апаратних та програмних компонентів засобів виконання програм, пісочниця виконує аналіз.

Формування звіту про аналіз позначимо s_{f_r}. Виконаний аналіз формує інформацію про виконання програми, на основі якої пісочниця може сформувати поведінку. Сформована поведінка подається у визначеному форматі об’єкту, що ініціалізував перевірку програми.

Виконання очищення (знищення) середовища виконання позначимо s_{f_d}. Пісочниця готує середовище для виконання наступної програми, тому повертає його у початковий стан, або формує нове.

Тому, якщо основні функції визначити як S_f, то набір функцій пісочниці, що необхідні для проведення аналізу програми на наявність зловмисної активності, задамо так:

$$S_f = \{S_{f_g}, S_{f_i}, S_{f_e}, S_{f_a}, S_{f_r}, S_{f_d}\}, \tag{2.8}$$

де s_{f_g} - отримання програми на перевірку; s_{f_i} - ініціалізація ізольованого середовища; s_{f_e} - передача на виконання отриманої програми; s_{f_a} - проведення аналізу виконання програми; s_{f_r} - формування звіту про аналіз; s_{f_d} - виконання очищення (знищення) середовища виконання.

Враховуючи розглянуті основні компоненти та функціональні особливості як емулятора, так і пісочниці, задамо узагальнене подання засобу для аналізу програм A так:

$$A = \{E_c, E_f, S_c, S_f\}, \tag{2.9}$$

де E_c – множина активних компонентів ЦП, елементами якої є множини регістрів та елемент пам’яті, яка визначена за формулою (2.5); E_f – основні функціональні особливості ЦП в контексті аналізу виконання програм, які задачі за формулою (2.6); S_c – множина компонентів пісочниці за формулою (2.7); S_f – основні функції пісочниці визначені формулою (2.8).

Отже, можна представити життєвий цикл роботи пісочниці як набір етапів, що послідовно виконуються для кожного екземпляра програми, яка підлягає аналізу.

Етап 1. Передача завдання на виконання. На цьому етапі засіб для аналізу програм A отримує від користувача чи системи, в залежності від його використання інфіковану програму I на аналіз.

Етап 2. Підготовка до виконання програми. Засіб для аналізу програм A ініціює створення пісочниці S, що в свою чергу ініціює створення емулятора для виконання інструкцій E разом із ізольованим середовищем виконання. Далі відбувається передача ІП I в емулятор E на виконання, для чого використовуються відповідні комунікаційні засоби.

Етап 3. Виконання переданої програми. Отримавши ІП I, емулятор E виконує дешифрування та поетапно виконує отриманий набір низькорівневих інструкцій. Під час виконання, E поступово змінює вміст множини активних компонентів ЦП E_c, і тим самим змінює стан кожного регістру і пам’яті, так і емульованого процесора в цілому. Коли усі інструкції опрацьовані, то виконується передача сигналу про завершення в пісочницю S.

Етап 4. Оцінка виконання. Пісочниця *S* виконує збір та аналіз даних про виконання програм, та формує звіт, який складається із сформованої поведінки виконаної програми та інших метрик, зокрема часових.

Етап 5. Фінальний етап. Засіб для аналізу програм *A* відправляє сформований звіт про аналіз виконання ІП *I* ініціатору аналізу, а також здійснює скидання усіх компонентів, що приймають участь у виконанні та аналізі програми, до початкового стану.

Графічне зображення процесу роботи *A* наведене схемою її етапів на рис. 2.3.

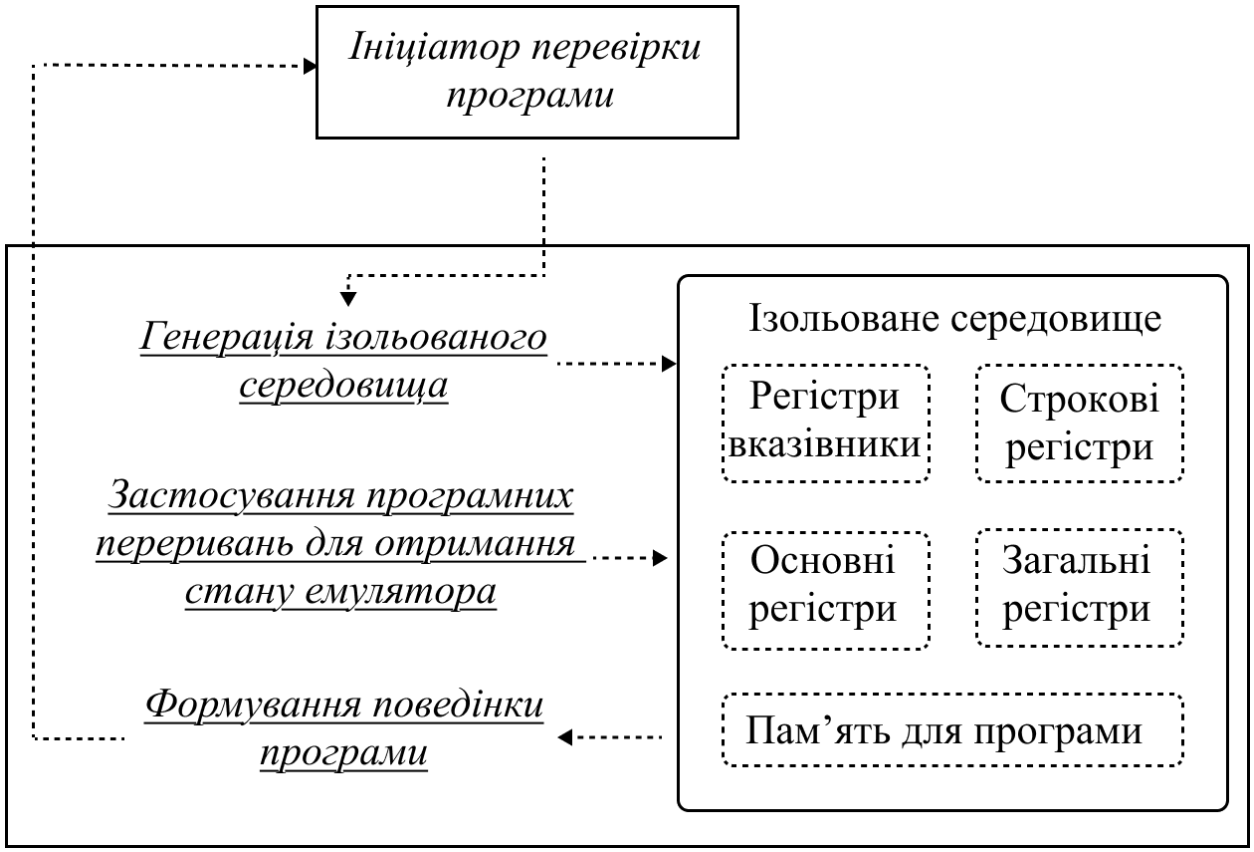


Рисунок 2.3 – Етапи аналізу поведінки програми

Таким чином, розроблено формалізовані компоненти та елементи пісочниці і запропоновано архітектуру засобів, які дозволять ініціювати процес виконання програми, що представлена у вигляді низькорівневих інструкцій. При цьому, програмна реалізація компонентів процесору дозволяє зчитувати вміст регістрів в обраний момент виконання програми. В цілому розроблені емульовані компоненти процесору та його функції можуть бути використані для аналізу функціонування програми та формувати її поведінку.

З точки зору інфікованої програми, якщо в неї закладені відповідні механізми протидії її дослідженню в емульованих середовищах, здійснюється класифікація середовищ виконання за трьома основними типами, а саме: невідоме середовище, тобто середовище, в якому розпочинається її виконання; ізольоване середовище, тобто успішно виявлене середовище, яке застосовує концепцію ізоляції за допомогою відповідних методів маскування; цільове середовище, тобто середовище, після аналізу якого яке визначається як безпечне до використання зловмисного програмного коду.

Так як успішність визначення емульованого середовища залежить від набору методів протидії емуляції, то ІП може трактувати цільовим, як безпосередньо ПП користувача, так і ізольоване середовище, що створюється АПЗ, для аналізу нової програми. Найбільш поширена стратегія виконання ІП полягає в очікувані, а саме очікування що ізольоване середовище стане цільовим. Це відбувається в тому випадку, коли АПЗ виконало перевірку ІП, і не виявило ознак зловмисного прояву, та визначило його як безпечне до використання, тим самим виводячи його із власного ізольованого середовища. Поетапний процес виконання ІП, в такому випадку, зобразимо на рис. 2.4.

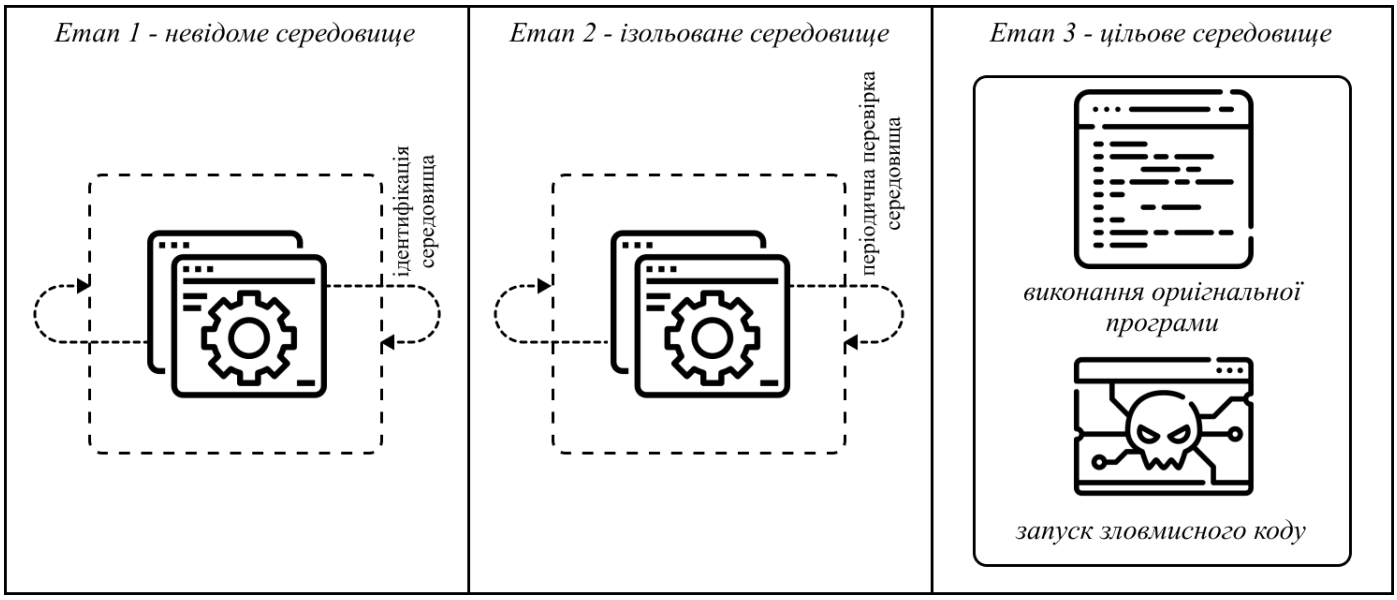


Рисунок 2.4 – Функціонування ІП в на хості із АПЗ

Тому, характер виконання ІП буде повністю залежати від набору методів протидії емуляції, що він застосовує, та точності реалізації функціонування самого

ізольованого середовища. При цьому, характер виконання буде уже змінюватись за допомогою методів, що базуються на методах поліморфізму. Враховуючи такі особливості у виконанні, доцільно сформувати поняття поведінки, а також пов’язати її до методів уникнення виявлення та характеристиками середовища, де ця виконувана програма виконується.

Визначимо середовище виконання як H, набір його характеристик, стосовно яких відбувається дослідження щодо присутності емуляції позначимо H_c, а h – безпосередньо характеристика. Під характеристикою, будемо вважати чи правильно була реалізована функціональна особливість емулятора відповідно до методів виявлення емуляції, яку застосовують ІП. Тоді, задамо середовище виконання так:

$$H_c = \{h_{ca}, h_{sc}, h_{it}, h_{ta}, h_{id}, h_{ht}, h_{hc}, h_{ia}\}, \quad (2.10)$$

де h_{ca} – елемент множини H_c, який характеризує правильність реалізації CPUID; h_{sc} – елемент множини H_c, який характеризує наявність аномалій виконання інструкцій при семантичних атаках; h_{it} – елемент множини H_c, який відповідає за наявність спотворень часових параметрів при виконанні визначених інструкцій; h_{ta} – елемент множини H_c, який відповідає за коректність реалізації функціонування таблиць IDT та LDT; h_{id} – елемент множини H_c, який відповідає за наявність засобів, що відповідають за створення затримок у виконанні програм; h_{ht} – елемент множини H_c, який відповідає за наявність засобів, що відповідають за модифікацію часових характеристик ПП; h_{hc} – елемент множини H_c, який відповідає за появу артефактів в пам’яті, що відповідають не реальному апаратному забезпеченню; h_{ia} – елемент множини H_c, який відповідає наявність артефактів інспектування системи.

У формулі (2.10) представлені параметри середовища, які стоються лише досліджених методів анти-емуляції. Візьмемо для прикладу характеристику правильності реалізації інструкції CPUID h_{ca}, яка відповідає за те, в якому вигляді буде подана інформація про процесор поточного ПП. ІП може мати визначений набір правильних позначень, які має повернути середовище, при застосуванні такої техніки. На основі таких перевірок і визначається наявність емуляції. Слід враховувати, що ІП можуть і помилково трактувати середовище на наявність емуляції, або її відсутність.

В залежності від типу перевірки, ІП можуть аналізувати і часові показники, зіставляючи їх із пороговими значеннями, перевіряти можливість виконання тих чи інших функцій, тощо. Тому, якщо ІП досліджує середовище виконання, а його поведінка динамічна і залежить одночасно як від характеристик середовища, так і від набору технік протидії емуляції, то справедливим буде сформувати наступне:

$$B: I \xrightarrow{S_{fe}} H \tag{2.11}$$

де B – поведінка виконання ІП в середовищі виконання; I – ІП, поведінка якої буде проаналізована; S_{fe} – функціональна особливість пісочниці, що передає на виконання ІП; і H – середовище виконання ІП.

Якщо, ж представити стан середовища виконання як – E_{state}, що формується до початку виконання програми, та враховує лише активні компоненти задані у формулі (2.5). Виконання програми спричинить зміну цих вмісту цих компонентів, і як наслідок – зміну стану. А, $\grave{E_{state}}$ - стан середовища після завершення виконання програми. Тоді, поведінку можемо задати наступним чином:

$$B: E_{state} \xrightarrow{H \wedge I} \grave{E_{state}}, \tag{2.12}$$

де B – поведінка виконання ІП, E_{state} – початковий стан середовища виконання, H – середовище виконання ІП, I – ІП, яка буде проаналізована, $\grave{E_{state}}$ – вихідний стан середовища виконання, після виконання у ньому ІП.

Формалізація сформованої поведінки важлива в контексті визначення наявності зловмисного прояву в ІП. Відштовхуючись від процесу виконання звичайної програми, в якій не використовуються методи базовані на поліморфізмі, її поведінка буде однаковою в усіх середовищах, так як не використовується методів перетворення коду виконання, на відміну від ІП, що активно застосовують подібні техніки, для уникнення виявлення.

Таким чином, розроблено архітектуру засобів, які залучають базові пісочницю і емулятор, що забезпечує часткову емуляцію виконання програм представлених у вигляді набору інструкцій в ізольованому модифікованому середовищі виконання. Маючи контроль над ізольованим середовищем, стає можливим формувати поведінку

виконання програми, використовуючи програмі переривання, на основі зміни стану компонентів ЦП, що визначені як активні.

2.3. Модель грід-обчислювальних систем для виконання аналізу програм

Застосування РСО в різних сучасних задачах є необхідним в багатьох професійних сферах. Це дозволяє ефективно використовувати розподілені ресурси для виконання складних обчислювальних задач. З точки зору організації таких систем виділяють чималу кількість систем, що відрізняються способом її організації та критеріями вибору елементів системи для виконання обчислювальних задач. Одним із провідних напрямків організації подібних систем визначають грід обчислювальні системи. Такі системи характеризуються залученням різнорідних (гетерогенних) обчислювальних елементів як за програмним, так і за апаратним забезпеченням, та вирізняються відносною легкістю масштабування. Це дозволяє легко збільшувати обчислювальну потужність системи за допомогою нових елементів, при цьому не вносячи кардинальні зміни в її архітектуру. Одним із успішних прикладів застосування грід-систем у сучасних умовах є програмна інфраструктура BOINC, яка залучає обчислювальні потужності користувачів для виконання складних задач на добровільній основі. BOINC використовують для вирішення декількох десятків наукових задач, залучаючи обчислювальні ресурси на добровільних засадах через спеціальний клієнтський застосунок. Такий підхід є раціональним і в контексті аналізу поведінки виконання програм, так як може забезпечити додаткову кількість обчислювальних елементів, що можуть бути використані для генерації ізольованих середовищ, та виконувати аналіз виконання ІП. Використання саме грід обчислювальної буде доцільне ще з точки зору її особливостей функціонування та організації.

Так як грід системи залучають гетерогенні обчислювальні елементи, для деяких завдань розподіленого обчислення, це може викликати незручності в довгостроковому плануванні обчислень. Використовуючи ж, обчислювальні елементи (ОЕ) для аналізу виконання програм, ця особливість не буде критично

відігравати роль. В цьому випадку кожен ОЕ, буде виконувати поставлену задачу (виконання та аналіз ІП) самостійно і за той час, який йому необхідний, враховуючи його власну продуктивність.

Ще однією ключовою характеристикою ОЕ в грід обчислювальних системах є їх автономність, що дозволяє користувачу визначати час та активність взаємодіяти із системою. Тому, при організації системи потрібно враховувати цю особливість, як на рівні алгоритму розподілу завдань, так і на рівні формування завдання. В такому випадку, завдання повинні бути незалежні один від одного, щоб не порушувати порядок виконання головної програми, яка використовує ці частини обчислень на різних етапах виконання. В інакшому випадку, доводиться проводити обчислення повторно, на декількох ОЕ. Поставлена задача аналізу не має залежностей такого роду, і в разі якщо один із ОЕ буде відключено, а значить не направить результати виконання, то цю частину можна буде направити на виконання на інший ОЕ.

Тому, при використання подібних типів систем, враховуючи низьку зв’язність обчислювальних задач та автономність вимагає перед її розробниками створювати алгоритми її функціонування такі, що передбачають можливе відключення ОЕ з системи. А з іншої сторони, такий алгоритм повинний враховувати нові ОЕ, які підключаються до системи. Це значно покращує, як здатність системи до масштабування, так і її відмовостійкість в контексті виконання обчислювальних задач.

Окремим завданням такої системи є забезпечення захисту обчислень, і це також визначається її типом. Застосування персональних пристроїв користувачів в якості ОЕ, може викликати ризики з точки зору правильності виконання задач. Це пов’язано зокрема із тим, що системі не належить їх ні програмне, ні апаратне забезпечення, а значить система не має контролю над ними. Через це, система не може гарантувати, що під час виконання обчислень на кожному із елементів, не відбувалось певних дій, через які результат обчислення був би спотвореним. Тому, застосовують різні підходи, зокрема моделі довіри які базуються на рейтинговій системи, яка дозволяє оцінити ймовірність правильності проведеного обчислення.

Централізована грід обчислювальна система складається з декількох ключових компонентів, що забезпечують її функціонування. Основним визначають центральний сервер, який відповідає за процес виконання задач, що подаються на виконання користувачами, а задачі виконуються на обчислювальних елементах. Розглянемо структуру обох типів компонентів, а також їх функціональні особливості, що забезпечать розподілений процес аналізу ІП.

Визначимо керуючим елементом системи – програмну частину ЦС O. Для забезпечення правильності функціонування системи, він має використовувати визначений набір програмних компонентів, до яких будуть входити: модуль для взаємодії із користувачем o_{ui}, що надає можливість користувачу завантажувати потенційні ІП для проведення аналізу виконання; модуль планування обчислень o_{ts}, основна задача якого полягає у формуванні завдань для активних ОЕ системи, що буде враховувати їх обчислювальну потужність та час виконання поставлених завдань; модуль комунікації всередині системи o_{sr}, який забезпечує комунікацію між сервером та ОЕ, для передачі завдань на виконання, отримання результатів виконання аналізу та виконання сервісних запитів у системі; аналітичний модуль o_{am}, мета якого аналізувати результати виконаних завдань для ОЕ; модуль захисту обчислень o_{tm}, який застосовує необхідні сервісні дії для зменшення шансу появи спотворення результатів що пов’язані через автономність ОЕ.

Тоді, архітектуру програмної частини ЦС задамо так:

$$O = \{o_{c_{ui}}, o_{c_{ts}}, o_{c_{sr}}, o_{c_{am}}, o_{c_{tm}}\} \tag{2.13}$$

де $o_{c_{ui}}$ – модуль для взаємодії із користувачем, $o_{c_{ts}}$– модуль планування обчислень, $o_{c_{sr}}$– модуль комунікації всередині системи, $o_{c_{am}}$– аналітичний модуль , $o_{c_{tm}}$ – модуль захисту обчислень.

Запропоновані модулі програмної частини ЦС O, при належному рівні реалізації, дозволять сформувати систему, яка буде надавати інтерфейс користувачу для завантаження файлів на аналіз, а також керувати процесом виконання аналізу на доступних ОЕ в поточний момент часу. Усі ІП, які подаються на аналіз, розміщуються в черзі, яка керується модулем для планування обчислень $o_{c_{ts}}$. Окрім цього, він

використовує інформацію яку надає модуль захисту обчислень, для захищеного планування виконання завдання по аналізу ІП. Модуль захисту обчислень в свою чергу, отримує результати виконання аналізу, з точки зору їх коректності. Для цього результати аналізу порівнюються із результатами довірених ОЕ, що дозволить сформувати рейтинг кожного учасника. Рейтингова система учасників дозволить пришвидшити результат аналізу, за допомогою зменшення повторних виконань завдань. Графічно представимо програмну частину ЦС на рис. 2.5.

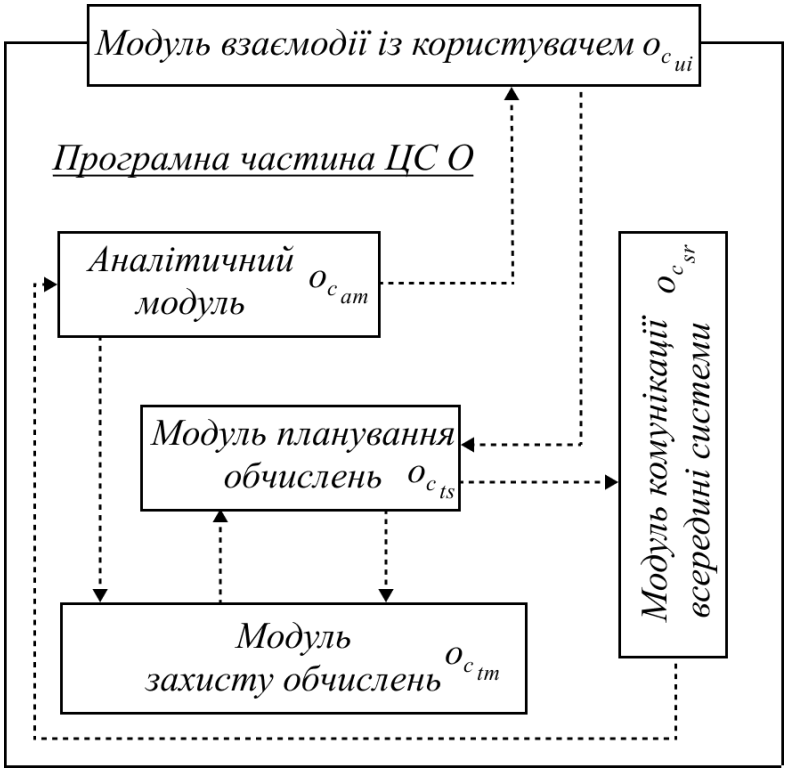


Рисунок 2.5 – Архітектура програмної частини ЦС

З огляду на ключові компоненти програмної частини ЦС та їх функціональні можливості, можна визначити основні функції, що йому притаманні.

Позначимо отримання програми на аналіз від користувача як o_{f_g}. Через відповідний модуль взаємодії із користувачем надається можливість передавати потенційні ІП для аналізу, яка буде згодом поміщена в чергу на опрацювання.

Передачу сервісних запитів до ОЕ позначимо як $o_{f_{sm}}$. Використання внутрішніх сервісних повідомлень використовується в багатьох внутрішніх процесах

функціонування системи, таких як: оновлення даних модуля захисту обчислень, зміна мережевого протоколу взаємодії перевірка доступності, тощо.

Планування наступної робочої ітерації позначимо як $o_{f_{sh}}$. Застосування автономних ОЕ для організації розподіленого процесу обчислення накладає необхідність виконувати планування на кожну робочу ітерацію враховуючи динамічність топології самої системи.

Позначимо передачу програми на аналіз як $o_{f_{st}}$. Сформовані та розподілені завдання відправляються на виконання до усіх ОЕ.

Аналіз виконаних завдань позначимо як $o_{f_{ta}}$. Він відбувається після кожної ітерації виконання завдання у випадку, якщо усі заплановані перевірки визначеного ІП уже завершились. Аналіз може бути не проведений, якщо кількість ОЕ не було достатньо, щоб опрацювати усі необхідні перевірки за одну ітерацію, або модуль захисту обчислень виявив сумнівність отриманих результатів і потребує повторного виконання певної частини аналізу ІП.

Оновлення даних модуля захисту обчислень позначимо як $o_{f_{utm}}$. Успішність використання цього модуля залежить від багатьох факторів, зокрема історичних даних про поведінку ОЕ та їх успішність з точки зору правильності виконання завдання. Нові дані використовуються для коригування алгоритму роботи модуля довіри.

Позначимо формування результату для користувача як $o_{f_{ur}}$. Якщо усі задачі, що пов’язані із аналізом одного ІП були виконані, то формується результат аналізу для користувача.

Розглянуті функції демонструюсь основні процеси, які відбуваються в системі на рівні програмної частини ЦС та підключених до нього ОЕ, тоді якщо O_f – основні його функції, то їх задамо так:

$$O_f = \{o_{f_g}, o_{f_{sm}}, o_{f_{sh}}, o_{f_{st}}, o_{f_{ta}}, o_{f_{utm}}, o_{f_{ur}}\}, \quad (2.14)$$

де o_{f_g} – отримання системою ІП від користувача, $o_{f_{sm}}$ – виконання сервісного запиту до ОЕ системи, $o_{f_{sh}}$ – планування наступної робочої ітерації, $o_{f_{st}}$ – відправка завдань

активним ОЕ системи, $o_{f_{ta}}$ – аналіз виконаних завдань після робочої ітерації, $o_{f_{utm}}$ – оновлення даних модуля довіри , $o_{f_{ur}}$ – формування результату аналізу користувачу.

Більшість цих функцій ініціюються завдяки подіям, які виникають під час роботи усіх або частини компонентів системи. Для більш цілісного представлення роботи системи розглянемо також структурні та функціональні особливості ОЕ.

Для проведення аналізу та формування поведінки виконання ІП, буде доцільно застосувати спеціальне програмне забезпечення. Щоб залучити його в розподілену систему, необхідно розробити архітектуру системи, що включатиме такий застосунок, в якому будуть виконуватись функції мережевої взаємодії у системі, користувацький інтерфейс ОЕ та визначений набір сервісних функцій. Інтерфейс програмної частини ОЕ тут використовується для візуалізації користувачу, який надав свої обчислювальні ресурси системі для проведення обчислень. Тому, якщо визначити ПЗ для ОЕ як, U, тоді компоненти програмної частини ОЕ можна задати як множину U_c. Тому, архітектуру ОЕ задамо наступним чином:

$$U_c = \{u_{c_{sc}}, u_{c_{ci}}, u_{c_{ap}}, u_{c_{ui}}\}, \tag{2.15}$$

де $u_{c_{sc}}$ – модуль комунікації із програмною частиною ЦС O, $u_{c_{ci}}$ - модуль керування застосунком для аналізу ІП S, $u_{c_{ap}}$ – модуль зчитування характеристик середовища виконання, та $u_{c_{ui}}$ – графічний інтерфейс програмної частини ОЕ.

Враховуючи визначені архітектуру та модулі програмної частини ОЕ та основні задачі, які на нього покладені в контексті формування поведінки виконання програми в системі, сформуємо функції, які він повинний забезпечувати.

Комунікацію із серверним застосунком позначимо як $u_{f_{sc}}$. Використовується, як для отриманні ІП, щоб виконати аналіз його поведінки виконання, параметрів його виконання так і для передачі результатів опрацювання ІП, а також для інформування про свій поточний статус.

Позначимо зчитування інформації про середовище виконання застосунку U як $u_{f_{ie}}$, яке виконується завжди при запуску застосунку, а також періодично під час його роботи. Використовується для збору сервісної інформації, як для модулю планування обчислень, так і модуля захисту обчислень.

Позначимо ініціалізацію пісочниці S як $u_{f_{ea}}$, який забезпечує виконання аналізу отриманого ІП в модифікованому ізольованому середовищі, шляхом передачі усіх параметрів йо виконання.

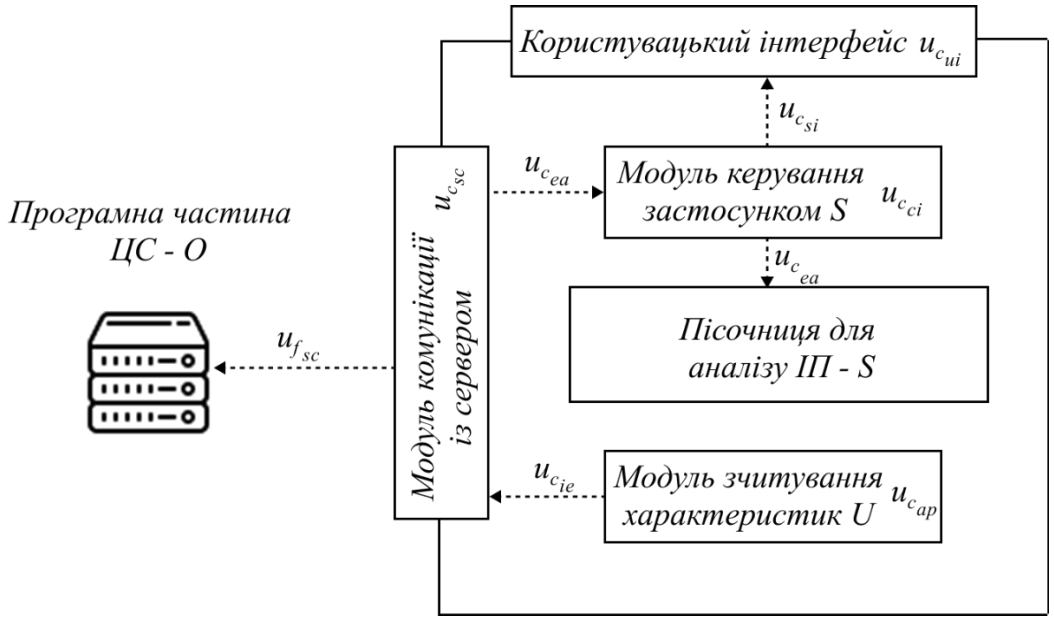
Представлення інформації користувача ОЕ $u_{f_{si}}$ представляє користувачу ОЕ детальну інформацію про його проведену роботі із аналізу програм, що виражається у різних параметрах, таких як час роботи, кількість проаналізованих екземплярі, його поточний рейтинг та інше.

Тоді, якщо визначити основні функції ПЗ ОЕ як U_f, тоді буде справедливим наступне:

$$U_f = \{u_{f_{sc}}, u_{f_{ie}}, u_{f_{ea}}, u_{f_{si}}\}, \tag{2.16}$$

де $u_{f_{sc}}$ – комунікація із програмною частиною ЦС O, $u_{f_{ie}}$ – зчитування інформації про середовище виконання програмної частини ОЕ U, $u_{f_{ea}}$ – ініціалізація пісочниці S для аналізу ІП, $u_{f_{si}}$ – представлення інформації користувачу ОЕ.

Процес залучення програмної частини ОЕ для використання в розподілених системах зображено на рис. 2.6 і демонструє застосування описаних функціональних особливостей застосунку.



Рисунку 2.6 – Схема потоків виконання аналізу в програмній частині ОЕ

Розглянуті компоненти системи та особливості їх функціонування, при відповідній організації, дозволять сформувати робочу систему обчислень із використанням гетерогенних, за апаратним та програмним забезпеченням ОЕ. При цьому, реалізація програмної частини ОЕ має бути виконана крос-платформними програмними засобами, або із засобами під кожну цільову ОС потенційних ОЕ. А при реалізації програмної частини ЦС, слід звернути увагу на засоби, що дозволять реалізувати підтримку підключення великої кількості одночасно підключених ОЕ. Архітектуру системи задамо схемою на рис. 2.7.

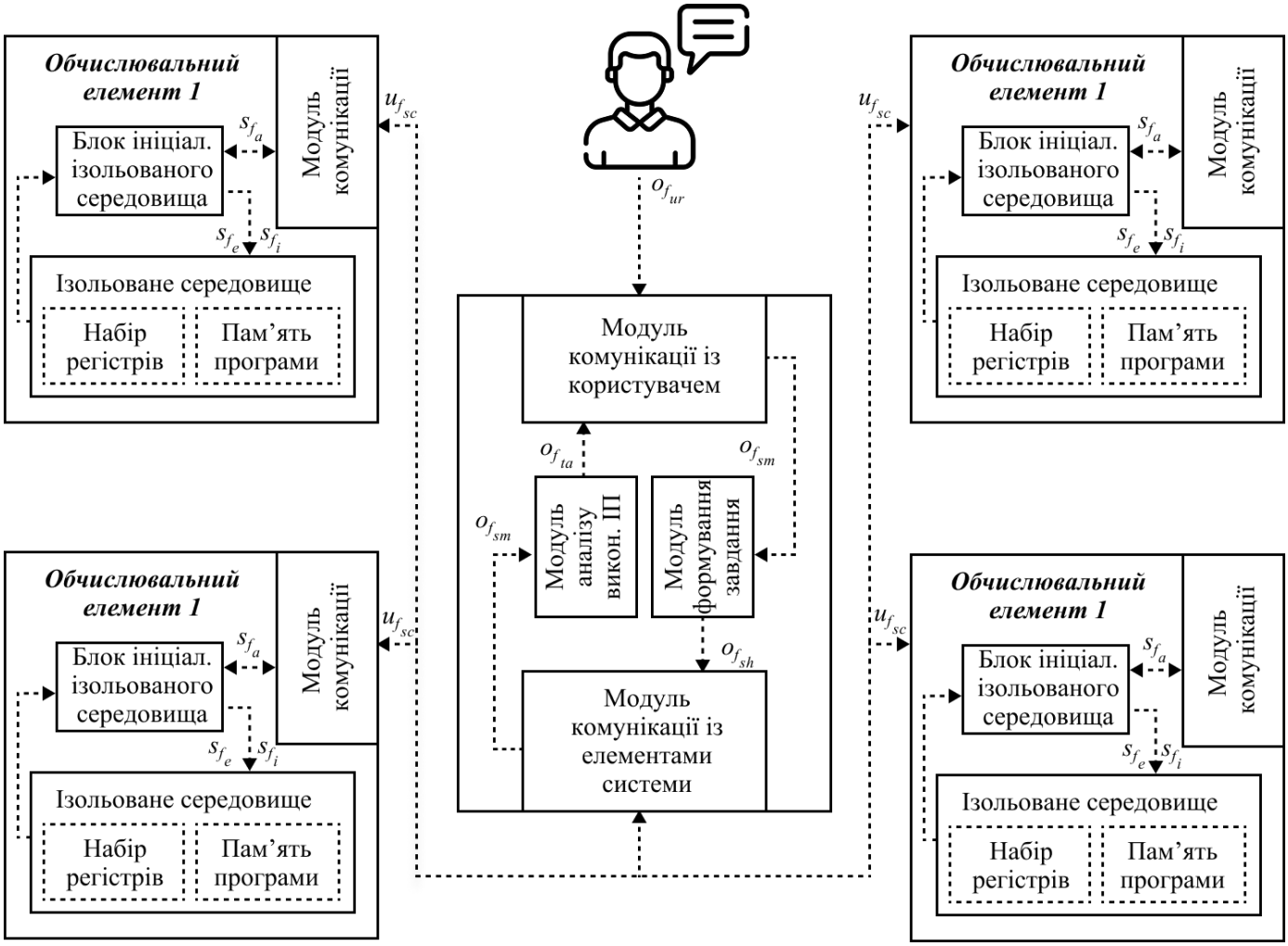


Рисунок 2.7 – Архітектура системи

Централізованість системи передбачає ініціювання усіх процесів тільки з сторони серверного застосунку. Основний потік його виконання працює ітеративно і, також, базується на реагуванні на події, що змінюють його стан. Основний ітеративний процес стосується проведення аналізу виконуваних програм, який ще

можна охарактеризувати як циклічний. Програма виконується постійно за умови, що черга ІП сформована користувачами не пуста. На початку кожної ітерації, сервер аналізує стан черги ІП та список активних ОЕ в поточний момент часу, та формує список задач на виконання. В рамках однієї ітерації, сервер може віддати на аналіз один або декілька зразків ІП, і це число буде залежати напряму від кількості активних ОЕ та кількості необхідних перевірок в контексті одного зразку ІП. При плануванні обчислень програмна частина ЦС, також, враховує і складність задачі, що виражається у вигляді коефіцієнту. Це дозволяє мінімізувати часи простою ОЕ, враховуючи їх апаратну особливість. Коли завдання на ітерацію сформоване, усі ОЕ отримують свій визначений список зразків ІП та набір необхідних параметрів для проведення аналізу програми. Після цього сервер переходить в очікуючи режим, та починає працювати над сервісними задачами, такими як оновлення інформації в бази даних відомих йому ОЕ, опрацювання екземплярів ІП, що надійшли від користувачів, опрацювання результатів роботи ОЕ. По завершенню робочої ітерації, усі ОЕ відправляють сформовані поведінки на ЦС та знищують ізольоване середовище, а потім переходять у режим очікування, та чекають на нові завдання. На стороні серверу ж, виконується збір сформованих поведінок, які сервер сортує за екземплярами. Для зменшення часу простою ОЕ, аналіз поведінок має другорядне значення з точки зору планування роботи системи, тому далі сервер виконує сервісне опитування усіх ОЕ, і цикл повторюється. Тому, аналіз одного ІП буде тривати більше ніж одну робочу ітерацію. Сюди, також, входять застосункові обчислювальні витрати, що виникають через автономну природу роботи ОЕ. Програмна частина ЦС повинна враховувати ситуації, коли ОЕ взяв в роботу аналіз ІП, і за якоїсь причини не повернув результати. Такі частки будуть відправлені повторно на опрацювання у наступні робочі ітерації програми.

Інші події виникають в системі без прив’язки до часу. До таких подій можна віднести завантаження користувачем файлу на перевірку системою o_{f_g}, що ініціює оновлення відповідно черги завдань. До такого ж типу подій можна також віднести і залучення нових ОЕ в розподілені обчислення. В такому випадку ОЕ будуть обмінюватись відповідними повідомленнями $o_{f_{sm}}$ для проходження ідентифікації,

після якої він буде доданий до множини активних ОЕ. На початку робочої ітерації, ця множина використовується для планування обчислень і безпосереднього залучення ОЕ в них. При першому підключенні ОЕ до системи, ЦС буде ініціювати виконання тестових завдань, щоб визначити його рівень довіри та його обчислювальну потужність. Зібрана інформація трансформується у набір різних коефіцієнтів, що дозволяють прогнозувати загальну обчислювальну потужність системи на наступну ітерацію виконання завдання. Це дозволяє оптимізувати час на планування завдання в умовах автономних ОЕ. Також, ця інформація використовується для модулю довіри системи, який відслідковує зміни в апаратному та програмному забезпеченнях, що стосується одного ОЕ.

Розглянутий опис особливостей функціонування грід-обчислювальних систем виявлення зловмисного прояву в інфікованих програмах потребує формалізації та деталізації, в особливості основних проміжних підходів, що визначають ключові етапи її роботи. Тому, потрібно розглянути наступні завдання, які стоять перед системою, а саме:

1) формування множини синтезованих засобів аналізу програм із модифікованими ізольованими середовищами для виявлення інфікованих програм із урахуванням принципів зміни виконання поведінки на програмно-апаратному рівні через застосування методів уникнення виявлення;

2) використання методу організації роботи ОЕ, що гарантують правильність проведених обчислень в системі враховуючи їх гетерогенність та динамічно змінювану топологію системи;

3) використання методів оцінки довіреності автономних ОЕ із використанням моделі довіри для її оцінки, на основі записів про успішність виконання поставлених завдань та поведінкові особливості кожного з них.

Таким чином, розроблено модель централізованих грід-обчислювальних систем, особливістю якої є врахування вимог до залучення автономних та гетерогенних обчислювальних елементів для забезпечення виконання задач із перевіркою на коректність в динамічному середовищі виконання і яка дає змогу залучити під’єднані обчислювальні елементи для аналізу поведінки виконання

інфікованих програм, забезпечуючи розподілений процес виявлення зловмисного прояву в інфікованих програмах.

2.4. Висновки до другого розділу

Проблема виявлення інфікованих програм пов’язана із застосуванням різноманітних методів уникнення виявлення, що наділяють їх динамічною поведінкою виконання. Для представлення їх особливостей функціонування були представлені моделі, які визначають саме ті зразки ІП, які використовують методи протидії емуляції та обфускації для уникнення від виявлення. Додатково, розглянуто стратегії виконання, що можуть застосовуватись в залежності від визначеного факту наявності ознак емуляції в середовищі виконання.

Для вирішення цього завдання було розроблено засіб та його архітектуру, що складається із базових емулятора та пісочниці, тим самим забезпечуючи часткову емуляцію виконання програм. Також, було розглянуто функціональні особливості як емулятора, так і пісочниці, що дозволяють їм спільно функціонувати.

Розроблено модель централізованих грід-обчислювальних систем. Її особливістю є врахування вимог до залучення автономних та гетерогенних обчислювальних елементів для забезпечення виконання задач із перевіркою на коректність в динамічному середовищі виконання. Вона дає змогу залучити під’єднані обчислювальні елементи для аналізу поведінки виконання інфікованих програм, забезпечуючи розподілений процес виявлення зловмисного прояву в інфікованих програмах.

Уведений опис особливостей функціонування грід-обчислювальних систем виявлення зловмисного прояву в інфікованих програмах потребує формалізації та деталізації в частині основних проміжних підходів, що визначають ключові етапи роботи.

Основні наукові результати другого розділу опубліковані в [89, 91, 124, 126, 127]

РОЗДІЛ 3
МЕТОДИ ТА ЗАСОБИ ОРГАНІЗАЦІЇ РОЗПОДІЛЕНОГО ВИЯВЛЕННЯ ЗЛОВМИСНОЇ ПОВЕДІНКИ

Успішність функціонування розподіленої системи залежить від багатьох факторів, одним із ключових з яких визначають внутрішні алгоритми, які встановлюють правила роботи усіх елементів і компонентів системи та взаємодію між ними. Алгоритми повинні враховувати особливості середовища їх виконання, головну мету функціонування системи, а також переваги та недоліки усіх її компонентів. Результуючий набір алгоритмів, повинен забезпечувати визначену поведінку системи, а результат його роботи має забезпечити працездатність та надійність визначену відповідно до завдання.

Поставлене завдання щодо розроблення методу виявлення зловмисної поведінки в ІП розподіленими системами представляється у вирішенні декількох основних задач, а саме: розроблення методу для організації функціонування розподілених систем; розроблення методу забезпечення захищеного виконання та аналізу різних зразків ІП на базі розподілених систем; реалізувати додаткову підсистему захисту обчислень.

Вибір грід-системи як типу розподілених систем супроводжується рядом викликів для її практичного застосування. Такі системи використовують гетерогенні та автономні ОЕ для вирішення поставленої задачі, що потребує використання алгоритмів, які зважають на ці особливості. Зокрема, вирішувати питання, як вільного підключення до системи будь-яким ОЕ в будь-який проміжок часу, так і його відключення. Метод розподіленого виявлення ІП, в якому реалізовані методи уникнення виявлення, повинен базуватись на детальному аналізі процесів, які відбуваються на програмно-апаратному рівні під час його виконання. Це дозволить сформувати поведінку виконання ІП для подальшого визначення присутності зловмисної дії. Зважаючи на автономність ОЕ, необхідно в рамках системи залучати додаткові захисні підсистеми, що будуть забезпечувати точність обчислень, на основі

додаткової ідентифікації програмного та апаратного забезпечення та аналізу їх поведінки в системі.

3.1. Метод синтезу засобів формування шаблонів поведінки інфікованих програм

Проведений аналіз функціонування ІП із використанням методів уникнення виявлення визначає динамічну поведінку їх виконання в залежності від особливостей середовища виконання. Тому, наступним кроком буде дослідження впливу таких методів на процес виконання ІП і на програмно-апаратну складову середовища виконання. Визначені методи виявлення емуляції дозволяють задати ключові характеристики середовища виконання, на які ІП будуть звертати увагу при виконанні. Тому, враховуючи визначене середовище виконання H (формула (2.10)), можна розглянути використання його модифікованої версії у вигляді множини M, для яких буде правильним наступне твердження:

$$M = \{x \in H_c \mid x - \text{застосовані модифікації}\}, \tag{3.1}$$

де M – модифіковане середовище, x – множина застосованих модифікацій застосованих для характеристиках середовища виконання, H_c – множина характеристик середовища виконання.

Елементи цієї множини визначають саме кількість модифікацій характеристик, які були зроблені для одного середовища, в рамках досліджених методів виявлення емуляції. Такі модифікації будуть впливати на поведінку виконання ІП, адже вони спеціально провокують на застосування методів уникнення від виявлення. Це викликано тим, що модифікації впливають поведінку емульованого програмно-апаратного забезпечення середовища виконання програм.

Проаналізувавши множини методів виявлення емуляції I_{ed} (формула (2.1)) та множину модифікацій середовища виконання M (формула (3.1)) встановлено, що їх записи відповідні один до одного, так як сформовані на одному і тому самому наборі проаналізованих методів уникнення виявлення. Враховуючи те, що елементи цих

множин відповідні один одному, то будемо їх вважати еквівалентними і визначимо їх як ознаки емуляції.

При виконанні ІП, в ізольованому середовищі можна сформувати поведінку B, яка може змінюватись відповідно динамічної природи ІП. Зважаючи на проведений аналіз зміни поведінки програм після інфікування сучасними версіями ЗПЗ та їх методи поліморфізму, можна встановити, що результуюча поведінка виконання в різних модифікованих середовищах буде відмінною. Так як поведінка буде залежати від набору методів виявлення емуляції I_{ed_i} деякого ІП I_i в деякому модифікованому середовищі із використаними модифікаціями M_j, то поведінка буде залежати від $M_j \cap I_{ed_i}$. Це пов'язано із тим, що в одному випадку емуляція була виявлена ($M_j \cap I_{ed_i} = \emptyset$), а в іншому – не була виявлена ($M_j \cap I_{ed_i} \neq \emptyset$). Використовуючи деяку кількість таких модифікованих середовищ, можна встановити про наявність зловмисного прояву в ІП, який був поданий на аналіз.

Для прикладу візьмемо умовний $I_1 = \{s_1, a_1, \langle i_{ed_{sc}} \rangle, \langle i_{p_{ci}}, i_{p_{sr}} \rangle, \langle i_{s_a} \rangle\}$, а також два модифікованих середовища $M_1 = \langle i_{ed_{sc}}, i_{ed_{ta}} \rangle$ і $M_2 = \langle i_{ed_{ht}}, i_{ed_{hc}} \rangle$. Тоді, в результаті виконання такої ІП у різних середовищах, можна отримати наступне за формулою (3.10):

$$B_{I_1} \begin{cases} B_1 : E_{state} \xrightarrow{I_1 \subset M_1} E^1_{state} \\ B_2 : E_{state} \xrightarrow{I_1 \subset M_2} E^2_{state} \end{cases}, \tag{3.2}$$

де B_{I_1} - набір поведінок виконання ІП I_1; B_1 та B_2 – поведінки виконання сформовані в модифікованих ізольованих середовищах M_1 та M_2; E_{state} – початковий стан емулюваного ЦП в ізольованому середовищі; E^1_{state} та E^2_{state} – стани ЦП після виконання в відповідних M_1 та M_2.

В цьому випадку, встановлено, що поведінки $B_1 \neq B_2$, так як і стани $E^1_{state} \neq E^2_{state}$, що буде означати відмінний процес виконання і що це буде свідчити про наявність зловмисного прояву.

Для іншого прикладу, візьмемо $I_2 = \{s_2, \emptyset, \emptyset, \emptyset, \emptyset\}$, що власне буде означати неінфіковану програму, та два модифікованих середовища $M_1 = \langle i_{ed_{sc}}, i_{ed_{ta}} \rangle$ та $M_3 =$

$\langle i_{ed_{sc}}, i_{ed_{ht}}, i_{ed_{hc}} \rangle$. В такому випадку, керуючись (3.2) результатом виконання та формування поведінки ми отримаємо E^1_{state} та E^3_{state}. В цьому прикладі, буде характерне те, що поведінки $B_1 = B_3$, так як стани виконання програм будуть $E^1_{state} = E^1_{state}$, які будуть рівні один до одного, що буде визначати програму як безпечну до використання, адже під час її виконання, не відбулось спроб змінити її поведінку, а значить не були застосовані маскувальні методи.

Враховуючи таку особливість виконання, можна сформувати метод виявлення зловмисної поведінки ІП, який буде складатись із наступних кроків:

Крок 1. Формування набору модифікованих ізольованих середовищ. Так як виконання ІП чутливе особливостей середовища виконання, для аналізу одного зразка необхідно представити набір різних модифікованих середовищ виконання. Такі середовища повинні відрізнятись згідно із набором параметрів, чутливих до методів виявлення емуляції. В рамках одного ІП, буде визначена наступна множина:

$$M_{I_i} = \{M_1, M_2, \dots M_n\}, \tag{3.3}$$

де M_{I_i} – множина модифікованих середовищ для ІП I_i яка складається з n модифікованих середовищ M, кожне з яких є відповідною множиною модифікацій.

Подібний розподіл дозволить спровокувати ІП на використання методів обфускації, що у свою чергу змінить поведінку виконання програми в цілому.

Крок 2. Передача ІП та параметрів ізольованого середовища на виконання. Отримавши необхідні параметри, відбувається формування унікального модифікованого середовища виконання за допомогою пісочниці S, де ІП буде виконана із використанням аналізу. Архітектура описаної пісочниці та її особливості функціонування дозволяють контролювати стан деяких компонентів емульованого процесора. Це дозволить сформувати поведінку програми, що включає в собі аналіз лише активних компонентів.

Крок 3. Виконання програми в ізольованому середовищі для формування поведінки виконання програми. Аналіз особливостей функціонування процесора визначає, що виконання інструкцій на процесорі змінює стан деяких його компонентів. В контексті аналізу виконання програми будемо розглядати наступні

компоненти, а саме регістри які зручно представити у вигляді наступних категорій: основні регістри – $R_b = \{r_{RAX}, r_{RBX}, r_{RCX}, r_{RDX}\}$, регістри вказівники – $R_p = \{r_{RBP}, r_{RSP}\}$, строкові регістри – $R_s = \{r_{RSI}, r_{RDI}\}$, регістри загального призначення – $R_r = \{r_8 \dots r_{15}\}$, а також пам'ять програми – i_{mem}. Також, в контексті аналізу програми, її пам'ять можемо представити в такому вигляді (3.4):

$$i_{mem} = \{(k_i, v_i) \mid i \in N\} \tag{3.4}$$

де i_{mem} – пам'ять програми, а k – ключ змінної, v – значення змінної, i – індекс змінної в пам'яті, I – індексна множина що разом формує впорядковану пару ключ-значення.

Враховуючи те, що усі обрані компоненти мають свій стан, який може змінюватись під час виконання інструкцій, можемо і сформувати стан емулятора E_{state}, що буде представляти собою комбінацію станів компонентів:

$$E_{state} = \left\{R_b, R_p, R_s, R_r, \{k_i : v_i\}\right\}, i = \overline{1, n}, \tag{3.5}$$

де E_{state} – стан емулятора в визначений проміжок часу; R_b – вміст основних регістрів; R_p – вміст регістрів вказівників; R_s – вміст строкових регістрів; R_r – вміст регістрів загального призначення; $\{k_i : v_i\}$ – пам'ять програми, яка представлена у вигляді словника «ключ-значення», яка має n записів.

Виконання програми буде змінювати стан емулятора (формула (3.5)) із кожною наступною виконаною інструкцією (рис. 3.1), а множина таких станів буде визначати характер роботи всієї програми в цілому, а також вплив на активні компоненти емульованого ЦП E_c, і як наслідок буде формувати вплив на ПП користувача. Дослідження зміни цього стану може дозволити виявити зловмисну поведінку, що відбувається під час виконання ІП.

На рис. 3.1 наведено приклад частини інструкцій які використовуються в програмах. Після виконання представлених інструкцій стан емулятора можна подати в такому вигляді $E_{state} = \{\{15,0,0,0\}, \{0,0,0,0\}, \{0,0,0,0\}, \{0,0,17,0,0,0,0,0\}, \{\emptyset\}\}$. Виконання наведених інструкцій для прикладу призвели до зміни стану лише в групі основних регістрів R_b та регістрів загального призначення R_r, а усі решта регістрів залишились із значеннями, які емулятор ініціював при створенні ізольованого

середовища. Колекція, яка виступає за збереження змінних, також залишилась порожньою, так як в наданому наборі інструкцій, які відповідають за їх створення або використання.

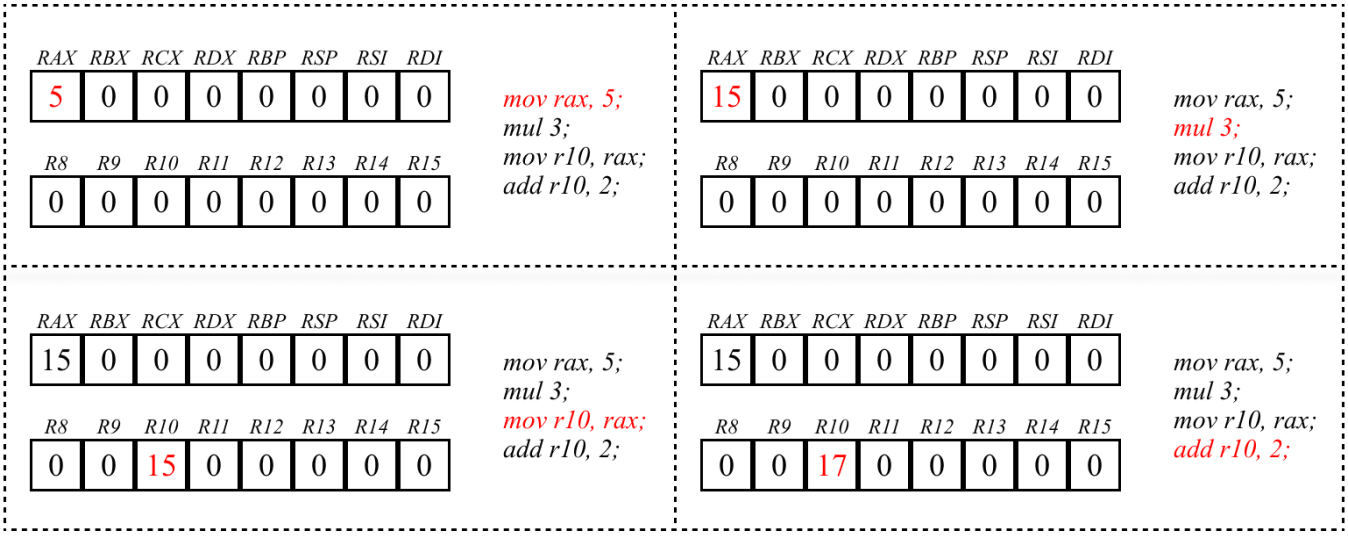


Рисунок 3.1 – Вплив виконання інструкцій на стан емулятора

Після виконня інструкцій, що представлені на рис. 3.2 для прикладу, загальний стан емулятора можна також подати так (3.6):

$$E_{state_j} = \{15,0,0,0,R_p,R_s,0,0,17,0,0,0,0,0,\{\}\}, j = 1 \dots m \quad (3.6)$$

В цьому прикладі j буде виступати кількістю виконаних інструкцій. Але, якщо проводити формування подібних записів та накопичувати їх, то їх сукупність буде формувати поведінку програми, так як вони фіксують усі зміни, що відбуваються у регістрах та пам’яті виконуваної програми.

Крок 4. Формування поведінки виконання програми. Для отримання інформації про стан емулятора на певному етапі виконанні програми доцільно застосувати програмні переривання, які слугують механізмом переривання звичайного потоку виконання будь-якого ПЗ. Вони характеризуються тим, що при їх виконанні процесор зберігає його поточний стан, тобто значення що зберігають регістри. За допомогою них виконуються зазвичай операції введення та виведення, системні виклики та обробка помилок. Хоча під час виконання обробних подібних переривань може змінювати поточні значення регістрів, але ці значення відновлюються, як тільки обробник завершує свою роботу. Такий механізм дозволяє процесору відновлювати нормально виконання програм. І в цілому, з точки зору процесора та його процесу

обробки інструкцій таких переривань і не існували. А також, програмні переривання відповідальні за забезпечення багатозачної роботи системи.

Розглянемо частину таких векторів на прикладі, що зображено на рис. 3.1.Застосовуючи переривання, в емульованому середовищі стає можливим формувати вплив завантажених інструкцій на процесор, і якщо на виконання була подана частина інструкцій розміром n, тоді для зручності стан кожного регістру можна представити у вигляді окремого вектору:

$$\vec{r}_{RAX} = \begin{pmatrix} 5 \\ 15 \\ 15 \\ 15 \end{pmatrix}; \vec{r}_{R10} = \begin{pmatrix} 0 \\ 0 \\ 15 \\ 17 \end{pmatrix}, \tag{3.7}$$

де $\vec{r}_{RAX}$ та $\vec{r}_{R10}$ – вектор регістрів RAX та R10 відповідно із значеннями, які вони набувають під час виконання програми.

Тоді, відповідно, за уже визначеними категоріями сформуються наступні множини векторів: для основних регістрів – $R_{b_{state}} = \{\vec{r}_{RAX}, \vec{r}_{RBX}, \vec{r}_{RCX}, \vec{r}_{RDX}\}$; для регістрів вказівників $R_{p_{state}} = \{\vec{r}_{RBP}, \vec{r}_{RSP}\}$; для строкових регістрів $R_{s_{state}} = \{\vec{r}_{RCI}, \vec{r}_{RDI}\}$; для регістрів загального призначення $R_{r_{state}} = \{\vec{r}_{R8}, \dots, \vec{r}_{R15}\}$.

Загальний процес зміни стану емулятора під час виконання, представимо у вигляді множини так:

$$E_{state} = \{R_{b_{state}}, R_{p_{state}}, R_{s_{state}}, R_{r_{state}}, i_{mem}\}, \tag{3.8}$$

де $R_{b_{state}}, R_{p_{state}}, R_{r_{state}}, R_{r_{state}}$ – множини векторів основних регістрів, регістрів вказівників, строкових регістрів, регістрів загального призначення відповідно, а i_{mem} – колекція змінних, які зберігаються в пам’яті.

Таке представлення дозволить аналізувати усю виконану програму на ОЕ, що буде корисним в процесі її аналізу, із метою виявлення передачі управління. Але, враховуючи те, що програми містять велику кількість інструкцій, доцільно привести множину векторів станів компонентів процесора до певного значення, яке буде простіше аналізувати. При цьому, таке приведення повинно застосовувати підходи, що будуть гарантувати унікальність отриманого значення. Для цього зручно

використати метод хешування для більшості перетворень під час формування такого значення, а алгоритм має бути підібраний таким чином, щоб максимально мінімізувати шанс на появу колізії. Так як, перетворення стосується усіх векторів та пам’яті програми, представимо його процес виконання покроково.

Крок 4.1. Виконати сумування усіх значень в різних станах емулятора для кожного окремого регістру.

Крок 4.2. Обчислити суму отриманих значень в попередньому кроці, та сформувати його текстове представлення.

Крок 4.3. Виконати обхід пам’яті виконуваної програми, яка містить пари значень, що відповідають за збереження змінних, які використовувались під час виконання програми у відповідній частині пам’яті, та сформувати їх сума.

Крок 4.4. Представлене значення пам’яті також перетворити у текстове представлення.

Крок 4.5. Текстові представлення станів набору регістрів та пам’яті об’єднати та передати на функцію хешування.

Представлені перетворення представимо наступним чином:

$$B = hash\left(toText\left(\sum_{i=0}^{n} (\sum_{j=0}^{m} \vec{r}_j)) + toText(id_k + val_k) \right) \right), \tag{3.9}$$

де B – поведінка ІП; $hash()$ – метод, який застосовує SHA-3 хешування; $toText()$ – метод, що перетворює числові значення у текст; $\vec{r}$ – вектор регістру для формування поведінки; i – кількість виконаних інструкцій; j – кількість регістрів в процесорі; k – кількість пар значень, які зберігаються в пам’яті в кінці виконання всієї програми.

Розглянемо використання такого методу на прикладі частини наборів інструкцій (Додаток Г), а саме порівняємо виконання «оригінального набору» та «метод заміни інструкцій». Якщо проаналізувати обидва набори, то в результаті їх виконання отримаємо однакові значення, але характер зміни станів окремих регістрів буде демонструвати різницю. Розглянемо таку зміну, яка відбувається для регістру RAX:

$$\vec{r}_{RAX_o} = \begin{pmatrix} 4 \\ 12 \\ 17 \end{pmatrix}; \vec{r}_{RAX_{ir}} = \begin{pmatrix} 4 \\ 8 \\ 12 \\ 17 \end{pmatrix}, \quad (3.10)$$

де $\vec{r}_{RAX_o}$ – вектор зміни стану регістра RAX під час виконання оригінального коду; $\vec{r}_{RAX_{ir}}$ – вектор зміни стану регістра RAX під час виконання модифікованого коду.

Такий приклад показує, що застосування формули (3.9) буде правильним для формування результуючого значення поведінки. Відмінність векторів буде мати вплив на фінальне значення поведінки, що покаже наявність модифікації, і при порівнянні зможе бути визначено, що програма була модифікована під час виконання, а значить зловмисний прояв присутній. Порожні інструкції, які часто використовуються як простий метод зміни сигнатури ЗПЗ, також, можна відслідковувати. Можна застосувати обчислення виконаних інструкцій, які передаються на процесор в рамках однієї програми. Таке число можна також враховувати при формуванні поведінки.

Окрему увагу слід приділити кількості станів, які використовуються в рамках аналізу виконання програми. Враховуючи те, що кількість інструкцій для сучасних програм суттєва, то фіксувати стан після кожної виконаної інструкції не видається можливим. Тому, потрібно проводити такий аналіз після певного числа виконаних інструкцій, що суттєво зменшить кількість програмних переривань під час аналізу. Це також важливо з точки зору захисних властивостей вірусів, які можуть аналізувати швидкість виконання інфікованої програми на хості, та припинити аномальну поведінку, визначивши що в процес виконання програми аналізується.

Крок 5. Виявлення зловмисної поведінки в ІП. Розподіливши ІП для виконання в множині модифікованих ізольованих середовищах B_{I_i} та ініціювавши їх виконання із аналізом, отримаємо набір їх поведінок, який задамо так:

$$B_{I_i} = \left\{ b_{u_{1_i}}, b_{u_{2_i}}, \dots b_{u_{n_i}} \right\}, \quad (3.11)$$

де B_{I_i} – множина поведінок ІП I_i; u_n – визначений ОЕ який використовувався для формування поведінки; n – кількість ОЕ, що приймали участь в аналізі ІП; b – сформована поведінка.

Окрім цього, доцільно враховувати і час виконання програми із урахуванням продуктивності ОЕ, що дозволить сформувати множину коефіцієнтів K_{I_i}, так:

$$K_{I_i} = \left\{k_{u_{1_i}}, k_{u_{2_i}}, \dots k_{u_{n_i}}\right\}, \tag{3.12}$$

де K_{I_i} – множина коефіцієнтів продуктивності виконання ІП I_i; u_n – визначений ОЕ, який використовувався для виконання ІП; n – кількість ОЕ, що приймали участь в аналізі ІП; k – коефіцієнт продуктивності.

Для визначення наявності зловмисної поведінки, потрібно застосувати обидва набори значень з формул (3.11) та (3.12). Їх використання пов'язані із наступними твердженнями: поведінка виконання в різних модифікованих середовищах має бути однаковою; коефіцієнт продуктивності не повинен відрізнятись більш ніж на 10% від контрольного значення k_b. Їх можна задати так:

$$\text{якщо } \begin{cases} \forall bx, by \in B_{I_i}, bx = by \\ \forall kx \in K_{I_i}, \left|\dfrac{kx - k_b}{k_b}\right| \leq 0.10 \end{cases} \rightarrow \text{ то не виявлено}, \tag{3.13}$$

де bx та by це одна сформована поведінка виконання ІП, B_{I_i} – множина поведінок сформованих після виконання ІП, kx – коефіцієнт продуктивності виконання ІП, k_b – контрольне значення продуктивності виконання ІП.

Якщо визначена умова істинна, то вважається що проаналізована ІП не має зловмисної активності. Друге правило базується на додатковому значенні k_b, яке обчислюється для кожного зразка ІП, що поданий на аналіз із використанням довіреного ОЕ або самим ЦС. В усіх решті випадках, проаналізована програма буде вважатись як такою, що містить зловмисну поведінку.

Розроблений метод використовує основні функціональні можливості представленого застосунку для аналізу виконання програми, а результатом його виконання буде сформоване текстове представлення числа, яке характеризує процес виконання і перетворене у відповідне текстове представлення за допомогою хеш-

функції для подальшого порівняння. Графічне представлення методу його кроками зображено на рис. 3.2.

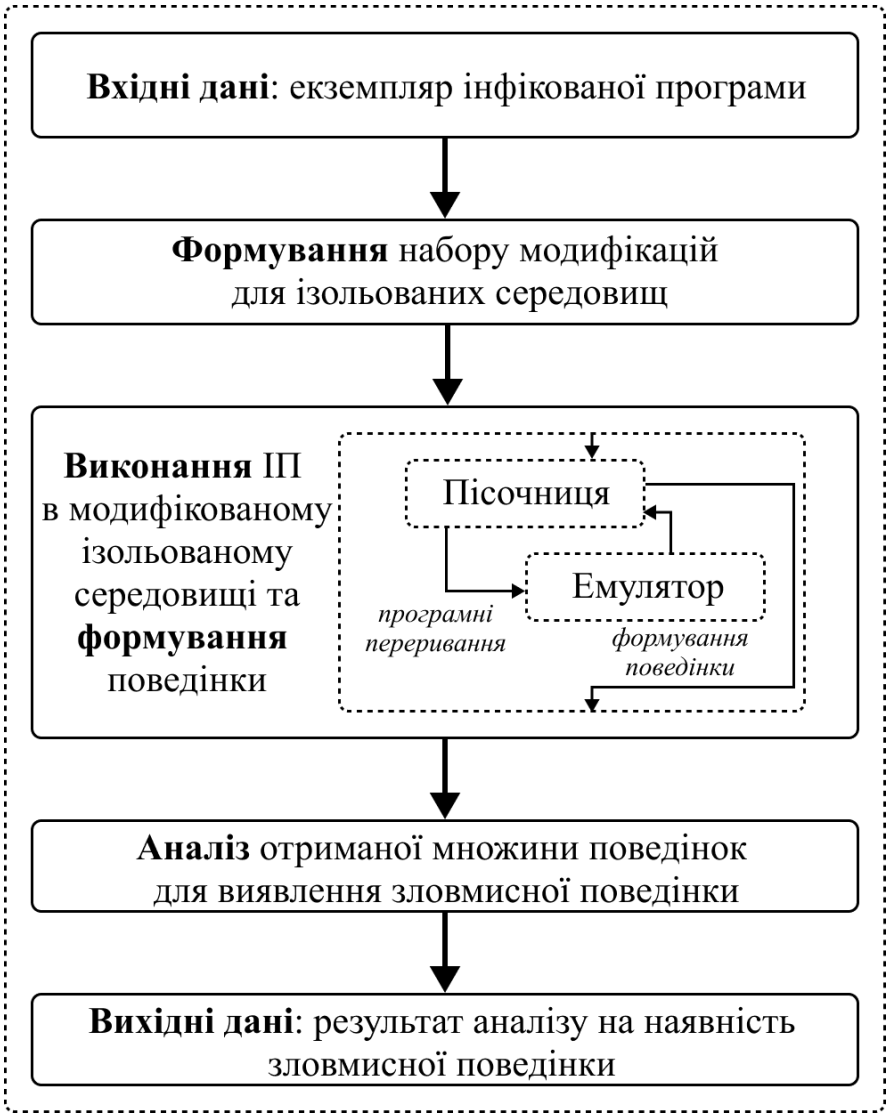


Рисунок 3.2 – Метод виявлення зловмисної поведінки в ІП

Беручи до уваги множину методів уникнення від виявлення та їх стратегії приховування зловмисної поведінки застосування запропонованої системи розподілених обчислень буде доцільним. Для задачі виявлення, ОЕ можуть бути використані як середовище для генерації модифікованих ізольованих середовищ для подальшого аналізу поведінки ІП у них.

Таким чином, розроблено метод синтезу засобів формування шаблонів поведінки інфікованих програм, особливість якого полягає в залученні пісочниці для їх виконання у наборі створюваних модифікованих ізольованих середовищ за допомогою виконання програмних переривань та базового емулятора із визначеним

набором реалізованих низькорівневих інструкцій, що дає змогу отримувати з них шаблони поведінки на множинах станів емульованих центральних процесорів. Таке застосування синтезованих засобів формування шаблонів поведінки інфікованих програм використовується як пастки з метою виявлення зловмисної поведінки з урахуванням особливостей методів уникнення від виявлення, які реалізовані зловмисниками.

3.2. Метод організації функціонування грід-обчислювальних систем для розподіленого виявлення зловмисної поведінки в інфікованих програмах

Забезпечення коректної роботи розподілених обчислювальних систем залежить від численних процесів, які виконуються в їх межах. Це охоплює процеси, що виконуються в середині серверного та клієнтських застосунків, а також процеси їх комунікації, що базуються на визначених правилах взаємодії. Так як, запропонована система характеризується як централізована, то усі процеси серверного застосунку які пов’язані із комунікацією із елементами системи можна визначити як керівні, так як вони ініціюють роботу інших елементів. Сюди буде входити більша частина функцій, зокрема це функції передачі сервісного запиту, а також передача програми на аналіз. Такі функції будуть ініціювати відповідні процеси на стороні ПЗ ОЕ. Інші процеси в системі мають засновану на подіях природу та виконуються коли наступає відповідна ситуація в системі. І усі ці процеси в цілому можна визначити як такі, що мають синхронну та асинхронну природу.

Головний потік процесу, який відповідає за організацію наявних обчислювальних елементів для виконання поставленого завдання. Цей процес виконується синхронно та ітеративно. Окрім цього, для своєї роботи програмна частина ЦС працює із трьома множинами різних об’єктів. До них віднесемо множину підключених ОЕ до системи – O_{au}, множину задач від користувачів, що очікують на виконання – O_{ut} системою, а також множину O_{at}, та множину задач, які виконуються.

Величина O_{au} представляє собою набір унікальних програмних об’єктів, які ідентифікують для серверного застосунку кожен підключений активний ОЕ в системі.

В цій множині поміщені тільки елементи, які в поточний момент знаходяться в режимі паузи, тобто очікують коли вони отримають задачу на виконання. Крім того, ця множина містить і інформацію про продуктивність кожного ОЕ. Тому, представити O_{au} можна наступним чином:

$$O_{au} = \{(e_1, p_1) \dots (e_i, p_i)\}, \quad (3.13)$$

де, e – програмний інтерфейс для роботи із ОЕ, p – коефіцієнт продуктивності ОЕ, а i – кількість активних ОЕ в поточний момент часу.

Величина O_{ut} містить в собі усі завдання, які потрапляють в систему від користувачів. Фактично, ця множина містить набір файлів, які можуть мати зловмисну поведінку і потребують аналізу, на думку користувача.

Величина O_{at} це множина завдань, які перебувають у роботі в поточний час роботи системи. При цьому, кожен елемент даної множини є також множиною, що складається із підзадач, які потрібно виконати для завершення вибраної поточної задачі із черги O_{at}. Підзадачі є незалежними одна від одної, але необхідно, щоб усі вони були виконані для завершення основного завдання. Розбиття задачі на частини відбувається за допомогою планувальника задач $o_{c_{ts}}$ (формула (2.13)).

В контексті поставленої задачі аналізу та виявлення ІП, можна визначити, що O_{ut}, складається із зразків ІП, які надають користувачі для аналізу. Коли система вибирає задачу в роботу, то виконується формування множини задач, які розміщуються у O_{at}, враховуючи запропонований метод виявлення зловмисної поведінки, адже він передбачає використання визначеної кількості модифікованих ізольованих середовищ. Окрім цього алгоритм, також, враховує автономність ОЕ, тому для виконання кожної задачі із O_{at} буде заплановане повторне її виконання на різних ОЕ. Це дозволить отримати множину результатів виконання однієї задачі, що дозволить прийняти рішення про правильність його виконання. Тому, фактично коли програмна частина ЦС вибирає задачу із O_{ut} в черзі O_{at} вона буде мати наступний вигляд:

$$t_{at_{I_i}} = \begin{pmatrix} \{t_{I_i}, M_{I_{i_1}}, c_{I_i}, u_{id_1}\} & \cdots & \{t_{I_i}, M_{I_{i_n}}, c_{I_i}, u_{id_1}\} \\ \vdots & \ddots & \vdots \\ \{t_{I_i}, M_{I_{i_1}}, c_{I_i}, u_{id_m}\} & \cdots & \{t_{I_i}, M_{I_{i_n}}, c_{I_i}, u_{id_m}\} \end{pmatrix}, \quad (3.14)$$

де $t_{at_{I_i}}$ – задача, яка пов'язана із аналізом ІП I_i; $M_{I_{i_n}}$- множина унікальних характеристик для модифікованого ізольованого середовища; n – кількість сформованих середовищ; c – обчислювальна складність поточної задачі; u_{id_m}- ідентифікатор ОЕ, який виконував аналіз програми; m – кількість ОЕ залучених для перевірки визначеного ІП в рамках одного унікального ізольованого середовища.

Кожна із таких підзадач, що представлена у вигляді відповідної множини, яка на етапі створення отримує усі елементи окрім u_{id_m}. Це значення залишається порожнім, поки сама підзадача не буде виконана, і відіграє важливе значення в забезпеченні правильності виконання обчислювального процесу на рівні системи. Графічно, розподіл задач зображено на рис. 3.3.

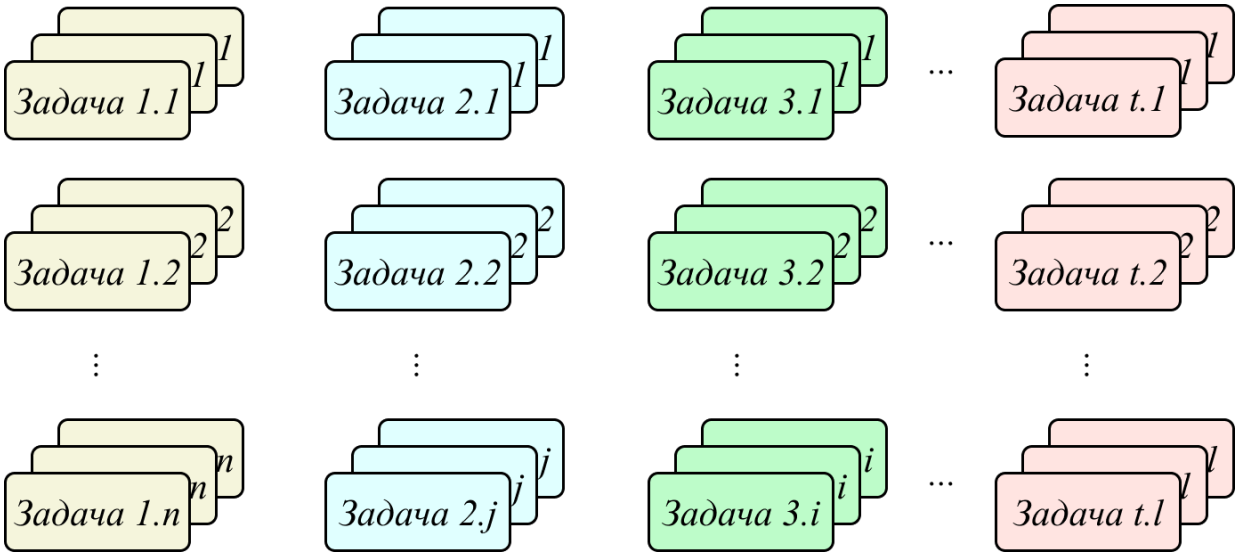


Рисунок 3.3 – Графічне представлення активних задач

В запропонованій системі, виконання обчислень відбувається із використанням ітеративних робочих сесій, які мають обмеження по часу. Розглянемо його у вигляді основних кроків

Крок 1. Отримання списку активних ОЕ. На початку робочої ітерації, центральний сервер використовує множину O_{au}, яка містить усі підключені активні ОЕ на поточний момент часу. Надсилає тестове сервісне повідомлення

використовуючи програмний інтерфейс e_i. Таким чином, здійснюється перевірка підключення усіх ОЕ, які готові до виконання обчислень.

Крок 2. Аналіз черги поставлених задач та застосування множини активних ОЕ. Програмна частина ЦС розпочинає роботу із аналізу множини активних задач O_{at}. На цьому етапі управління на себе бере $o_{c_{ts}}$, який проводить первинну оцінку на достатність задач, щоб заповнити усі активні ОЕ задачами у відповідності до визначеної тривалості робочої ітерації. Для оцінки використовуються така умова:

$$\sum_{n}^{i=1} c_{at} \geq p_{min} * m * t, \tag{3.15}$$

де c_{at}– це набір усіх активних підзадач ($c_{at} \in C, C = \{c \mid \{t_I, M_I, c_I, u_{id}\} \in O_{at}, u_{id} = 0\}$); n – кількість активних підзадач; p_{min} – це найменша продуктивність серед усіх активних ОЕ на поточний момент часу ($p_{min} \in P, P = \{p \mid \{e, p\} \in O_{au}\}$; m – кількість активних ОЕ; t – час робочої ітерації.

У випадку, якщо умова не справджується, то центральний сервер буде вибирати задачі із O_{ut} та формувати підзадачі поки умова не буде істинною, окрім випадку якщо O_{ut} – порожня. Як тільки умова буде істинною, то алгоритм переходить на наступний крок. При цьому, неявний пріоритет виконання завдань користувачів O_{ut} буде зберігатись. Це спричинено тим, що в активному списку підзадач O_{at} зберігаються тільки задачі, які поступово вибираються із відповідної черги завдань користувачів, і додаються туди тільки тоді коли умова в формулі (3.15) не істинна.

Крок 3. Пошук найкращого варіанту розподілу завдань поточної робочої ітерації. Вирішуючи задачу пошуку ефективного розподілу задач, необхідно врахувати наступні фактори, а саме: кожна підзадача та ОЕ мають обчислювальну складність та продуктивність відповідно; деякі підзадачі не можуть виконуватись на визначених ОЕ, так як це суперечить заходам, які забезпечують коректність проведених обчислень. Слід також враховувати і кількість підзадач та ОЕ в системі, це може запропонувати більш ефективний підхід. Концепція використання грід-обчислювальних систем в сучасних реаліях не передбачає надто великої кількості ОЕ в рамках однієї системи, так як може викликати суттєві часові та обчислювальні

втрати на службові та операційні заходи. У випадках, коли система залучає надто велику кількість ОЕ, то їх розбивають на підсистем. Підсистеми, в такому випадку, краще піддаються управлінню, а їх формування дещо схоже на загальні принципи побудови програмно-конфігурованих мереж. І в такому разі, система отримує обрис одного із типів грід-систем, а саме граничних систем обчислень. Тому, визначивши те, що кількість ОЕ в системі не буде суттєвою, можемо сказати і те саме про кількість активних підзадач. В результаті, одним із найкращих варіантів, враховуючи усі динамічні аспекти застосованих розподілених обчислень, є використання модифікованого жадібного алгоритму розподілу.

Спочатку алгоритм виконує сортування множини активних підзадач O_{at} в порядку від найбільшого до найменшого за складністю виконання. Далі, створюється тимчасова множина зайнятості кожного активного ОЕ - $Y_{au} = \{y_{au_1}, y_{au_2}, \dots y_{au_n}\}$ із початковими значеннями 0, а її розмір еквівалентний кількості активних ОЕ в поточний момент часу. Далі, алгоритм вибирає із першу задачу із списку O_{at} першу підзадачу, та робить оцінку виконання її на кожному активному ОЕ із O_{au} і використанням:

$$y_{au_{I_i}} = \frac{c_{I_i}}{p}, \tag{3.16}$$

де $y_{au_{I_i}}$- оціночна продуктивність виконання задачі I_i, зі складністю виконання c_{I_i}; p – продуктивність поточного ОЕ.

Таким чином, система отримає множину значень, які представляють теоретичне часове представлення часу виконання поточної задачі на кожному доступному ОЕ. Так як запропонований алгоритм в першу чергу намагається розподілити задачі так, що загальний час виконання системи (в рамках поточної ітерації) був мінімальним, то він виконає проходження по усій множині Y_{au} та виконає попереднє сумування отриманого часу $y_{au_{I_i}}$із кожним значенням множини. При цьому, алгоритм шукає мінімальну результуючу суму за формулою (3.17) для того, щоб визначити ідентифікатор ОЕ для призначення задачі до виконання:

$$u_{id} = \arg \min_{k \in \{1 \dots n\}} \left(y_{au} + y_{au_{I_i}} \right), \forall y_{au} \in Y_{au}, \quad (3.17)$$

де u_{id} – шуканий ідентифікатор призначення задачі; n – кількість активних ОЕ; $y_{au} \in Y_{au}$ – елемент множини, який визначає поточну завантаженість кожного активного ОЕ; $y_{au_{I_i}}$ - теоретичний час виконання задачі поточної задачі.

При цьому, алгоритм враховує u_{id} серед виконаних або призначених дублікатів задач, таким чином, щоб він не потрапив до того самого елементу. У випадку, якщо таке співпадіння буде знайдене, алгоритм пропустить цей варіант та обере найліпший за таким самим показником. Виконуючи призначення множині ОЕ, задачі будуть потрапляти спершу до тих, що мають вищий коефіцієнт продуктивності, але потім алгоритм збалансує навантаження грунтуючись на результаті формули (3.17).

Застосовуючи даний алгоритм, можуть з’являтись випадки, коли деякі ОЕ не отримають жодного завдання. Це спричинено зокрема тим, що в черзі активних задач знаходиться мала кількість різних зразків завдань на виконання. Збільшення кількості різних задач в обробці дозволить мінімізувати відсоток ОЕ в простої, але і в деяких випадках може сповільнити швидкість опрацювання цілого завдання, що подається користувачем.

Крок 4. Збір результатів виконання. Так як алгорити планування сформований таким чином, щоб рівномірно розподілити навантаження між множиною ОЕ, то і час виконання у задач має бути в цілому однаковий. Тобто, час простою ОЕ між фактом виконання поставленого завдання та початком нової ітерації невеликий, хоча і буде унікальним для кожного ОЕ для кожної нової ітерації. Як тільки ОЕ завершує виконання своєї частини роботи, він формує результат виконання і надсилає засобами мережевої комунікації його до серверного застосунку. При отриманні такого повідомлення, програмна частина ЦС ідентифікує задачу та виконавця. Це дозволяє оновити O_{at}, визначивши як результат виконання так і ідентифікатор ОЕ, який виконав задачу.

Етап збору результатів виконання відбувається протягом часу, який визначається на основі двох значень, а саме визначений час робочої ітерації а також

не обов’язковий коефіцієнт, який встановлюється адміністратором. Даний коефіцієнт в цілому призначений для збільшення тривалості ітерації для врахування операційних та мережевих витрат.

Коли час ітерації завершується, то програмна частина ЦС приймає що усі завдання, які не були виконані (тобто не було надіслане відповідне сервісне повідомлення від ОЕ) вважаються не виконаними. Навіть якщо ОЕ пізніше спробує надіслати результат виконання, то він відкине його результат. Також, по завершенню цього часу, сервер завершує поточну ітерацію та переходить до кроку 1, так як топологія в розглянутій системі може динамічно може змінюватись.

Розглянемо ще процеси, які виконуються при появі відповідних умов в системі, що також відіграють важливу роль в успішному функціонуванні системи. До таких процесів відноситься: поява нового ОЕ в системі; аналіз виконаної задачі; формування звіту про виконання аналізу програми.

Поява нового ОЕ в системі. Система в цілому залучає нових учасників в якості ОЕ із використанням спеціального застосунку, U (рис. 2.6) який має відповідний модуль та визначені налаштування для створення мережевого підключення із серверним застосунком. Так як алгоритм функціонування системи та розподілу завдань між ОЕ опирається на визначений коефіцієнт продуктивності кожного із них, програмна частина ЦС спробує також сформувати це значення. Для цього, він використовує набір тестових завдань, які він надсилає усім новим ОЕ на виконання. Отримавши відповідь від кожного з ОЕ програмна частина ЦС оцінює час виконання, та спираючись на еталонне значення часу виконання робить оцінку його продуктивності. Окрім цього, буде проведена оцінка і результату виконання тестового завдання, якщо відповідь збігається, то тоді ЦС вважає новий ОЕ придатним до використання і залучає його до списку активних ОЕ O_{au}. При цьому додає усю необхідну інформацію для подальшої взаємодії із ним, що формує собою програмний інтерфейс взаємодії.

Поява нової задачі від користувача. Така подія також не відбувається синхронно, але її подія запускає ряд важливих процесів функціонування системи. При появі нового файлу, програмна частина ЦС зробить його експериментальну оцінку

складності виконання, шляхом його виконання на довіреному ОЕ. І після його виконання, формує відповідний коефіцієнт, який потім буде використовуватись під час процесу розподілу завдань. Разом із сформованим коефіцієнтом, поточна задача спершу поміщається в O_{ut}. Далі, відбувається первинна робота над задачею, основною метою якої на розподіл на множину підзадач, що мають бути виконані для завершення всієї задачі. Сформовані підзадачі фіксуються уже в черзі O_{at} із необхідними дублікатами для забезпечення коректності виконання поставленого завдання.

Здійснимо аналіз виконаної підзадачі та задачі. Кожного разу, коли поточна ітерація функціонування системи завершується, виконані підзадачі відповідно позначаються в загальній черзі активних завдань. І як тільки розпочинається нова ітерація виконання завдань, програмна частина ЦС в асинхронний спосіб виконує пошук завершених як підзадач так і задач. Щодо підзадач, виконується пошук такої, для якої усі дублюючі були виконані, і якщо такі знаходяться, то алгоритм перевіряє усі результати. Так як усі підзадачі тут однакові, то і результат виконання має збігатись. У випадку, якщо результат не збігається, то підзадача вважається не правильно виконаною і повертається на повторне виконання, враховуючи усі дублюючі її частини. Такий підхід дозволить виявити ОЕ, які намагаються скомпрометувати результати роботи системи, що може бути викликано, як втручанням в роботу програмної частини ОЕ так і діями вірусних програм. Коли аналіз підзадач завершено, застосовується пошук задач, для яких усі підзадачі уже виконані і які пройшли перевірку на правильність виконання. Якщо така задача знайдена, то програмна частина ЦС аналізує результати виконаних підзадач та формує результат про виконану роботу, який потім відправляється у вигляді звіту користувачу, що ініціював виконання задачі.

Розроблений метод подано графічно на рис. 3.4 з врахуванням його основних кроків.

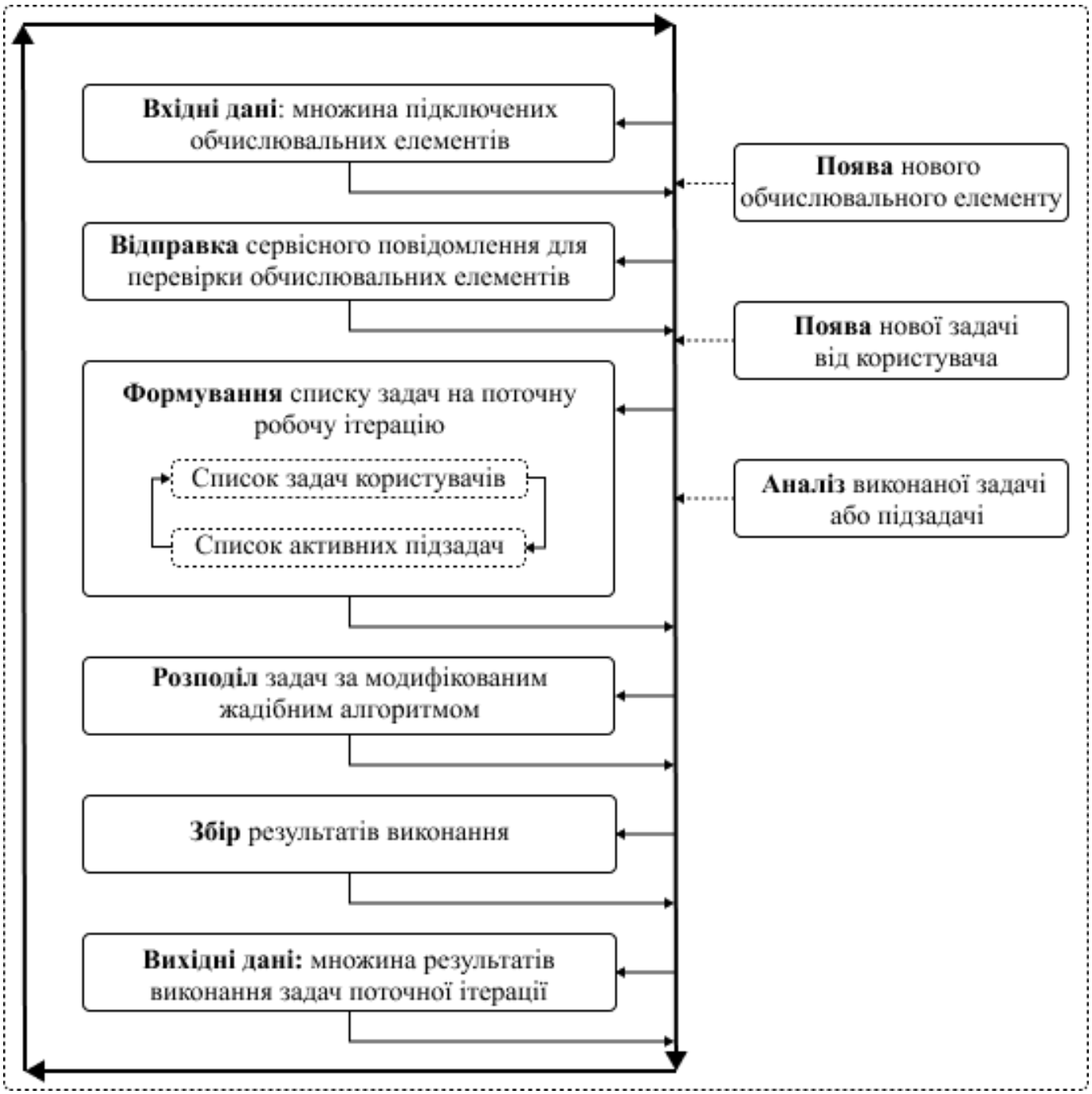


Рисунок 3.4 – Схема кроків методу розподілу задач в розподілених системах

Таким чином, удосконалено метод організації функціонування грід-обчислювальних систем, в основі якого застосовано жадібний алгоритм для оптимізації навантаження між автономними гетерогенними обчислювальними елементами та використано додаткову чергу активних задач, що забезпечує збалансоване виконання поставлених задач в розподілених системах із динамічно змінюваною топологією для розподіленого виявлення зловмисної поведінки в інфікованих програмах.

3.3. Метод оцінювання довіри автономних обчислювальних елементів грід-обчислювальних систем

Використання моделі довіри в грід системах обчислень дозволяє підвищити ефективність використання обчислювальних ресурсів ОЕ. Залучення такого підходу може також допомогти зменшити кількість повторних обчислень, уже виконаних задач, що є критичним з точки зору забезпечення правильності виконання обчислень в системах такого типу. А також, моделі довіри досить часто опираються на голосування, яке проводиться між елементами, що виконали поставлену роботу, щодо правильності виконання однієї задачі, зазвичай опираючись на однаковий результат усіма учасниками. Якщо, ж серед такої групи хоча б один результат виявився не таким, центральний сервер мусить втрутитись в роботу, та призначити повторне виконання, або перевірку довіреними ОЕ.

Модель довіри є програмним підходом, основна задача якого полягає в управлінні довірчими відносинами між компонентами системи, а його використання є критичним з огляду на тип розподілених систем та особливості визначеного ОЕ. Автономність кожного ОЕ, може становити загрозу для великих обчислювальних задач, які за допомогою алгоритмів розділяються на множину менших, які потім призначаються учасникам системи. Результат всієї задачі залежить від правильності виконання кожної частини, тому навіть декілька виконаних некоректно підзавдань спотворить результат всієї задачі, а також зменшить ефективність використання ресурсів ОЕ. Таким чином, система яка має забезпечувати правильне виконання розподілених обчислювальних задач, і яка не може довіряти жодному з її елементів, при використанні моделі довіри може вирішити визначені проблеми.

В основі сучасних моделей довіри лежить застосування різноманітних підходів, зокрема таких, як репутація, голосування, технологія блокчейн, теорія ігор, аналіз поведінки та методи машинного навчання. Сучасні запропоновані рішення зазвичай покладаються на декілька таких підходів, комбінуючи їх для отримання кращого результату в залежності від особливостей об’єктів, для яких модель довіри впроваджується.

В контексті поставленої задачі, для реалізації моделі довіри необхідно застосувати одночасно декілька факторів довіри. Його робота повинна базуватись на механізмах голосування, а для вирішення можливих конфліктів під час голосування, застосувати елементи нечіткої логіки та поведінкових підходів. Конфліктами визначаються ситуації коли окремі ОЕ, що відповідальні за обчислення одних і тих же завдань повертають різні результати. У таких випадках система повинна використати рейтинг, що базується на історії уже проведених обчислень для кожного елемента та враховувати його поведінку.

Класична модель довіри оперує бінарними ознаками: «довіряю»; «не довіряю». Використовуючи подібну логіку з точки поставленої задачі, можемо визначити, що такий статус для ОЕ, визначає чи можна центральний сервер приймати результати виконаних обчислень. Але, для такого підходу необхідно визначення набору ключових факторів, які будуть використання для формування припущення про довіреність до визначеного ОЕ. А, усі ці фактори, мають базуватись на інформації, що доступна для аналізу. Для її збору, треба зважати на інформацію, яку можна зібрати, використовуючи основні та додаткові бібліотеки мови програмування, що використовувалась для реалізації системи. Обрана мова програмування, для реалізації цього проекту дозволяє збирати часткову інформацію, як про апаратне, так і про програмне забезпечення ОЕ, для подальшого формування його поведінки. Оцінювання поведінки проводиться після кожного виконаного завдання, що дозволяє оновлювати рейтинг довіри до ОЕ постійно, і тримати в актуальному стані. Для оцінювання поведінки, використовується два набори даних:

$$U_{sb} = \{b_{os}, b_{cpua}, b_{cpum}, b_{cpus}, b_{ram}, b_{dir}, b_{mac}\}, \quad (3.18)$$

де U_{sb} – це набір інформації, що об’єднує відомості про програмне та апаратне забезпечення; b_{os} – версія операційної системи; b_{cpua} – архітектура ЦПУ; b_{cpum} – модель ЦПУ; b_{cpus} – базова швидкість ЦПУ; b_{ram} – кількість оперативної пам’яті ОЕ; b_{dir} – шлях домашнього каталогу; b_{mac} – MAC - адреса.

$$U_{db} = \{b_{taskc}, b_{taskp}, b_{tasks}, b_{taskr}, b_{prod}\}, \quad (3.19)$$

де U_{db} – це набір інформації, що містить показники, які характеризують поведінку ОЕ під час виконання завдання; b_{taskc} – обчислювальна складність виконаної задачі; b_{taskp} – час, який витрачено на її виконання; b_{tasks} – час, який витрачено на передачу задачі до ОЕ; b_{taskr} – час, який витрачено на передачу результату, коли задача була уже завершена; b_{prod} – коефіцієнт продуктивності виконання поточної задачі.

Враховуючи те, що рівень довіри зазвичай набуває значень в межах від 0 до 1, то потрібно розділити цей діапазон між різними ролями, які будуть призначатись ОЕ. Зважаючи на усі виклики поставлені перед системою, пропонується виокремити такі ролі.

Довірений обчислювальний елемент передбачає, що система ніколи не буде ініціювати застосункові перевірки отриманих результатів. Він вважається правильним за замовчуванням. Такі ОЕ можуть використовуватись для перевірки інших результатів обчислення.

Звичайний обчислювальний елемент, коли система ніколи не довіряє результатам обчислення таких ОЕ, тому вона буде надсилати дублікати завдань іншим ОЕ системи, щоб забезпечити правильність виконання.

Новий обчислювальний елемент це, коли такі ОЕ розпочинають роботу в системі зі певним визначеним адміністратором рівнем довіри, і центральний сервер буде надсилати тестові завдання для оцінки їхньої обчислювальної потужності, доти поки вони не отримають необхідний рівень, який дозволить приймати участь в обчисленнях. Рівень довіри такого ОЕ за замовчуванням є нижчим за звичайний ОЕ.

Ненадійний обчислювальний елемент надає таку роль ОЕ в тому випадку, якщо допущено багато помилок при обчислення завдань, які були виявлені під час голосування. Система визначає їх як скомпрометовані, та вимагає відновлення їх рівня довіри для подальшого залучення в обчислення, шляхом надсилання застосункових завдань, які будуть перевірятись на правильність, але самі результати обчислень не будуть прийняті до уваги.

Подібна класифікація дозволить виділити довірених ОЕ, що працюватимуть без перевірок, тим самим вивільняючи ресурси для інших задач. Також, необхідно забезпечити можливість коригування порогових значень рівнів довіри між ролями,

що дозволить адміністратору керувати часткою ресурсів, що будуть спрямовані на повторні обчислення. Додаючи такий модуль довіри в грід-обчислювальну систему, центральний сервер має забезпечити збір необхідної інформації для формування рівня довіри. Деталізоване подання грід-обчислювальної системи (рис. 2.7) із використанням програмної частини ЦС (формула (2.13)) та обчислювальними елементами із визначеними ролями можемо представити графічно, зокрема на рис. 3.5.

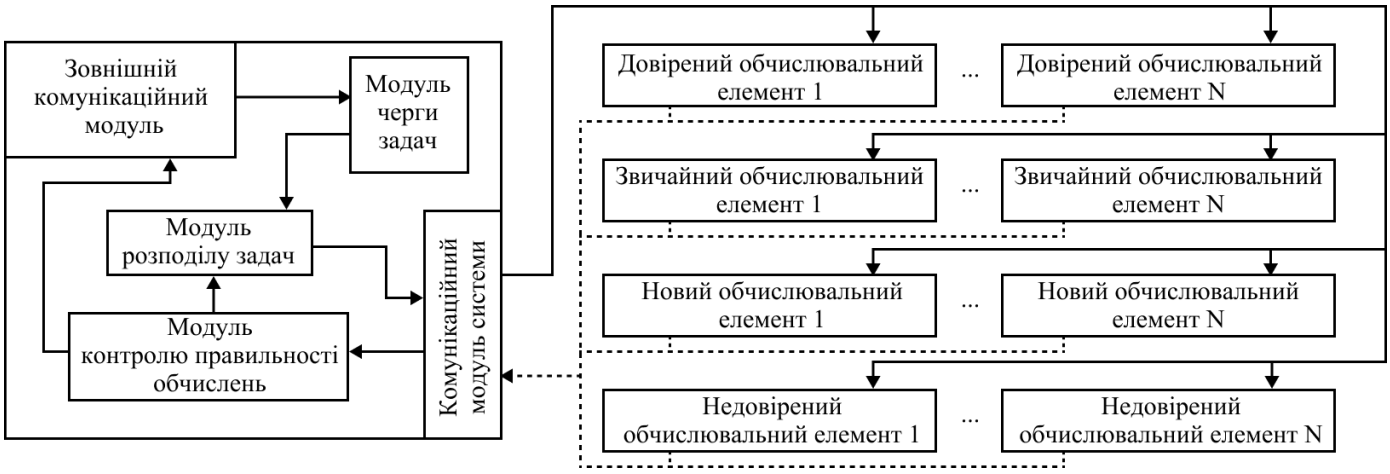


Рисунок 3.5 – Архітектура розподіленої системи з рольовим розподілом обчислювальних елементів

Представимо метод оцінювання довіри автономних обчислювальних елементів у вигляді кроків функціонування.

Крок 1. Попередня підготовка та нормалізація параметрів. Для формування правильного рейтингу, який буде враховувати попередню інформацію (відомості про участь в попередню робочу ітерацію) необхідно підготувати та класифікувати визначені значення, а тільки після того виконати нормалізацію, що дозволить визначити чи був ОЕ скомпрометований після поточної ітерації обчислень. Для цього, необхідно розділити пропонується усі параметри за їх природою та способом подальшої обробки. Це важливо, так як усі параметри мають різне призначення і потребують відповідного підходу до нормалізації. Зокрема, виділимо нечислові параметри. До них віднесемо ті, що характеризують апаратне та програмне забезпечення (формула (3.18)), та використовуються у визначенні збігу поточних

характеристик із зафіксованим профілем користувача. Для цих параметрів нормалізація відбувається за принципом бінарної відповідності так:

$$b_{sb_{i_n}} = \begin{cases} 1, & b_{sb_i} = b_{sb_{iet}} \\ 0, & b_{sb_i} \neq b_{sb_{iet}} \end{cases}, b_{sb_i} \in U_{sb}, \tag{3.20}$$

де $b_{sb_{i_n}}$ – результуюче нормалізоване значення; $b_{sb_{iet}}$ – еталонне значення, отримане на початку роботи із системою або після успішного виконання завдання.

Інша категорія параметрів (формула(3.19)) це параметри які відображають характер виконання завдань, і для того, щоб врахувати складність завдань, усі значення попередньо приводяться до складності виконаної задачі c із O_{at} (формула (3.14)). Після цього застосовується метод мінімуму-максимуму (формула (3.21)) для проведення нормалізації:

$$b_{dba_i} = \frac{b_{db_i}}{c_{I_i}}; b_{dba_{i_n}} = 1 - \frac{b_{dba} - b_{db_{i_{min}}}}{b_{db_{i_{max}}} - b_{db_{i_{min}}}}, \tag{3.21}$$

де b_{dba_i}- приведений параметр, який відповідає за поведінку виконання завдання; b_{db_i} – параметр з U_{db}; $b_{dba_{i_n}}$- нормалізований параметр; $b_{dba_{i_{min}}}$ та $b_{dba_{i_{max}}}$- мінімальне та максимальне значення обраного параметру.

Крок 2. Формування оцінок довіри. Далі, на основі отриманих нормалізованих значень виконується формування двох окремих групових оцінок: оцінка стабільності конфігурації ОЕ (формула (3.22)) та оцінка поведінки обчислень вузла (формула (3.23)):

$$t_{sb} = \frac{1}{m}\sum_{i=1}^{m} b_{sb_{i_n}}, \tag{3.22}$$

де t_{sb} – оцінка стабільності ОЕ; $b_{sb_{i_n}}$ – нормалізований параметр характеристик ОЕ множини U_{sb}; m – кількість параметрів.

При формування стабільності конфігурації потрібно використати середнє значення визначених показників. Такий підхід часто використовується, для оцінювання саме постійних параметрів або малозмінних параметрів. Щодо оцінки поведінки обчислень вузла, то тут пропонується використати множину вагових

коефіцієнтів для кожного параметру, що дозволяє визначати важливість того чи іншого параметра в загальній оцінці. Така оцінка формує інтегральну оцінку довіри, і формує узагальнене значення довіри проведених обчислень:

$$t_{db} = \frac{\sum_{i=1}^{m} w_j * b_{dba_{i_n}}}{\sum_{i=1}^{m} w_j}, \tag{3.23}$$

де t_{db} – оцінка поведінки ОЕ; w_j – вага, для кожного зваженого параметру; $b_{sb_{i_n}}$ – нормалізований параметр характеристик ОЕ множини U_{db}; m – кількість параметрів.

Крок 3. Обчислення рівня довіри ОЕ. Сформовані значення використовуються для підрахунку уже фінального рівня довіри кожного елемента:

$$u_{tl} = w_s * t_{sb} + w_d * t_{db}, \tag{3.24}$$

де u_{tl} – результуючий рівень довіри; w_s – вага оцінки стабільності ОЕ; t_{sb} – значення оцінки стабільності ОЕ; w_d – вага оцінки поведінки ОЕ; t_{sb} – оцінка поведінки ОЕ.

Врахування усіх цих характеристик є важливим, так як користувачі зазвичай не часто змінюють характеристики своїх ПП (як апаратне, так і програмне забезпечення), тому якщо підключення здійснюється із незвичними характеристиками, велика імовірність того, що обліковий запис користувача було викрадено. Збір інформації щодо поведінки виконання поставлених задач для виявлення відхилення від визначеної норми важливий, так як це може вказувати на наявність, наприклад, ЗПЗ на ОЕ, що потенційно може спотворити результати обчислень. Тому, така оцінка дозволить визначити скомпрометований ОЕ також.

Крок 4. Динамічне коригування рейтингу ОЕ. Під час роботи системи та проведених обчислень усіма типами ОЕ, система оцінить їх правильність та ефективність і буде коригувати поточний рейтинг для кожного вузла. Правильність виконання збільшує рейтинг, а виконання завдання із відмінною відповіддю або в деяких випадках навіть прострочене завдання призводить до його зменшення. Для коригування рейтингу застосуємо таку формулу:

$$u_{tl} = u_{tl_p} + l * u_{tl} * l_k, \tag{3.25}$$

де u_{tl} – рівень довіри; u_{tl_p} – рівень довіри ОЕ за попередню робочу ітерацію; l – набуває значення 1 або -1, та залежить від успішності виконання завдання поточної робочої ітерації; l_k – коефіцієнт встановлений адміністратором.

Передбачається, що адміністратор встановить основні порогові значення значення, щоб допомогти системі визначити роль кожного вузла, який бере участь в виконанні задач. Гнучкість коефіцієнтів, дозволить визначити динаміку переходу із однієї ролі до іншої, кожним активним ОЕ, особливо якщо застосувати різний коефіцієнт для переходу від однієї ролі ОЕ до іншої. Запропонований метод зображено графічно схемою кроків на рис. 3.6.

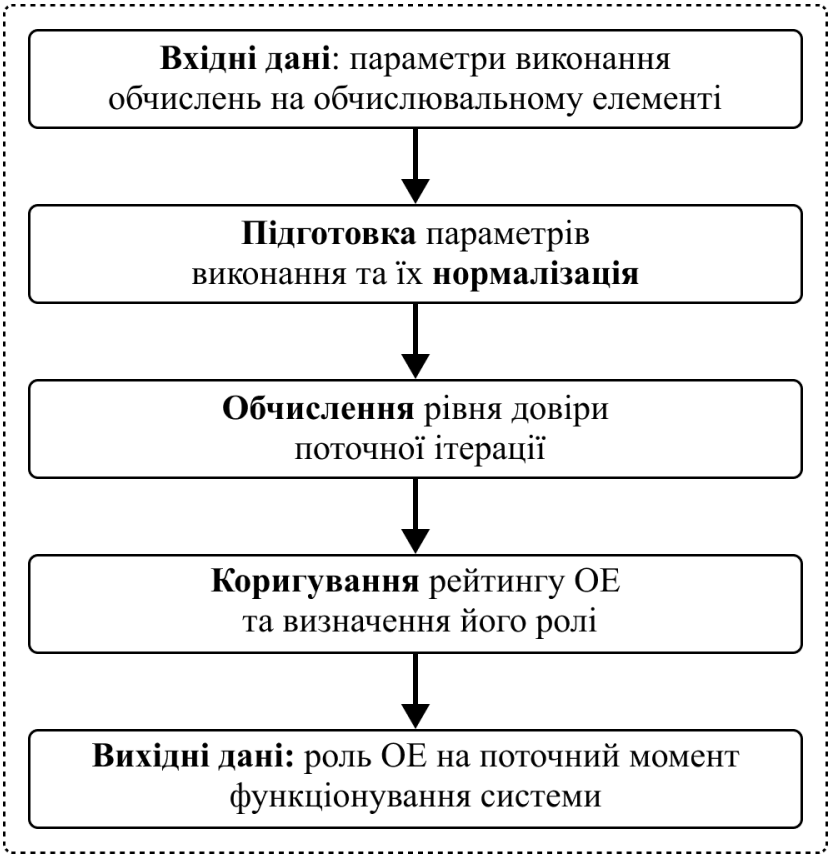


Рисунок 3.6 –Схема кроків методу з коригування ролі обчислювального елементу в розподілених системах

Таким чином, удосконалено метод оцінювання довіри автономних обчислювальних елементів, в якому використано механізми призначення ролей із застосуванням елементів нечіткої логіки для оптимізації обчислювальних ресурсів шляхом скорочення кількості повторних обчислень у системах, що функціонують в динамічному середовищі.

3.4. Висновки до третього розділу

Організацію грід-обчислювальної системи для виявлення зловмисної поведінки в інфікованих програмах забезпечено розробленими трьома методами.

Розроблено метод синтезу засобів формування шаблонів поведінки інфікованих програм, особливість якого полягає в залученні пісочниці для їх виконання у наборі створюваних модифікованих ізольованих середовищ за допомогою виконання програмних переривань та базового емулятора із визначеним набором реалізованих низькорівневих інструкцій, що дає змогу отримувати з них шаблони поведінки на множинах станів емульованих центральних процесорів. Застосування синтезованих засобів згідно розробленого методу забезпечує формування шаблонів поведінки інфікованих програм і використовується як пастки з метою виявлення зловмисної поведінки з урахуванням особливостей методів уникнення від виявлення, які реалізовані зловмисниками.

Удосконалено метод організації функціонування грід-обчислювальних систем, в основі якого застосовано жадібний алгоритм для оптимізації навантаження між автономними гетерогенними обчислювальними елементами та використано додаткову чергу активних задач. Це забезпечує збалансоване виконання поставлених задач в розподілених системах із динамічно змінюваною топологією для розподіленого виявлення зловмисної поведінки в інфікованих програмах.

Удосконалено метод оцінювання довіри автономних обчислювальних елементів, в якому використано механізми призначення ролей із застосуванням елементів нечіткої логіки для оптимізації обчислювальних ресурсів шляхом скорочення кількості повторних обчислень у системах, які функціонують в динамічному середовищі.

Основні наукові результати третього розділу опубліковані в [89, 90, 91, 126, 127, 128, 129]

РОЗДІЛ 4
РОЗПОДІЛЕНА СИСТЕМА ВИЯВЛЕННЯ ІНФІКОВАНИХ ПРОГРАМ ТА РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТІВ

Розглянемо архітектуру реалізованого ПЗ для організації функціонування грід-обчислювальної системи із використанням автономних обчислювальних елементів. Створене ПЗ складається із двох програмних застосунків, для центрального серверу та для обчислювального елементу. Створена система застосовує усі визначені практики, а також використовує усі запропоновані методи, як для організації розподілених обчислень, так і аналізу виконання ІП спираючись на розподілений підхід. Усі методи в коді програмного застосунку центрального сервера реалізовані як окремі модулі інтегровані в ітеративний життєвий цикл застосунку. Для підтвердження досягнення ефективності запропонованих рішень необхідно провести експериментальні дослідження розподіленої системи за показниками ефективності у відповідності до поставлених задач.

Для стабільної роботи розробленого програмної частини ЦС рекомендується використовувати систему з процесором не нижче Intel Core i5 (8-го покоління) або AMD Ryzen 5 (2-го покоління). Обсяг оперативної пам’яті повинен складати не менше 8 ГБ, що забезпечить достатній резерв додатка із супутніми системними службами. Для хостингу застосунку на сервері рекомендується використовувати віртуальний сервер із щонайменше 2 vCPU та 4 Гб оперативної пам’яті. Наявність SSD для обох випадків є дуже рекомендованою, використання якого підвищить чутливість і стабільність роботи застосунку. Рекомендований канал зв’язку має бути із пропускною здатністю в 100 Мбіт/с, що забезпечить належну швидкість обміну файлами та іншого службового трафіку між центральним сервером та ОЕ. Серверна частина ПЗ розподіленої системи може бути зібрана та запущена на будь-якій популярній настільній версії операційної системи, що підтримує Java Virtual Machine.

4.1 Реалізація програмного забезпечення розподіленої системи виявлення інфікованих програм, що застосовують методи уникнення виявлення

4.1.1 Опис експериментальної установки

Реалізація програмного застосунку для центрального сервера передбачає використання різних програмних модулів, що відповідають за різні процеси, які відбуваються під час його функціонування. Проте, з точки зору адміністратора, після відповідної компіляції, усі вони формують єдиний застосунок, який використовується на його операційній системі. За допомогою нього, відбувається уся мережева та організаційна координація процесів в системі, що застосовується для аналізу ІП на автономних ОЕ.

В якості основи, для розробки даного серверного застосунку було обрано мову програмування Java. Основними критеріями її вибору були: надійні механізми захисту внутрішніх компонентів через застосування модифікаторів доступу; підтримкою строгої типізації, що є критичним фактором в питаннях розподілених обчислень, оскільки допомагає забезпечити консистентність даних та передбачувану взаємодію між усіма учасниками системи; підтримкою крос-платформеності, що забезпечується Java Virtual Machine (JVM); широкими можливостями для реалізації мережевої взаємодії, а також підтримці паралелізму та багатопоковості, продуктивність яких додатково оптимізується завдяки Just in Time (JIT) компіляції.

Під час реалізації серверного застосунку, окрім стандартних бібліотек мови Java, було використано JavaFX – бібліотеку для побудови графічних інтерфейсів настільних застосунків, яка в даному випадку слугувала для створення панелі адміністратора з метою керування процесом обчислень та його відслідковування, а також збору статистичних даних. Інтерфейс побудовано з використанням декларативної мови розмітки FXML. Для зберігання критичних даних, зокрема інформації про авторизацію, історії виконаних завдань та поведінкових характеристик ОЕ для модуля довіри, використовувалась система керування базами даних PostgreSQL разом із відповідним драйвером для з Java. З огляду на характер

обчислень і способи підключення обчислювальних елементів було застосовано асинхронний підхід управління потоками на центральному сервері з використанням віртуальних потоків та технології Project Loom, що забезпечує легку масштабованість і високу пропускну здатність системи. Для реалізації мережевої взаємодії використовувався ServerSocket, який забезпечує надійне та контрольоване з'єднання.

Реалізована програмна частина ЦС функціонує як настільна програма, що здійснює ініціалізацію всіх необхідних програмних модулів для забезпечення його функціонування. Збірка застосунку виконувалась з використанням технології Maven, яка генерує єдиний JAR-файл для запуску в середовищі настільної операційної системи, при умові якщо там були встановлені пакети JDK або JRE. Після запуску програми адміністратор отримує можливість розпочати роботу системи, внаслідок чого створюються мережеві об'єкти, що забезпечують підключення ОЕ відповідно до визначених мережевих параметрів, визначених у налаштуваннях. Віконна форма серверного застосунку зображена на рис 4.1.

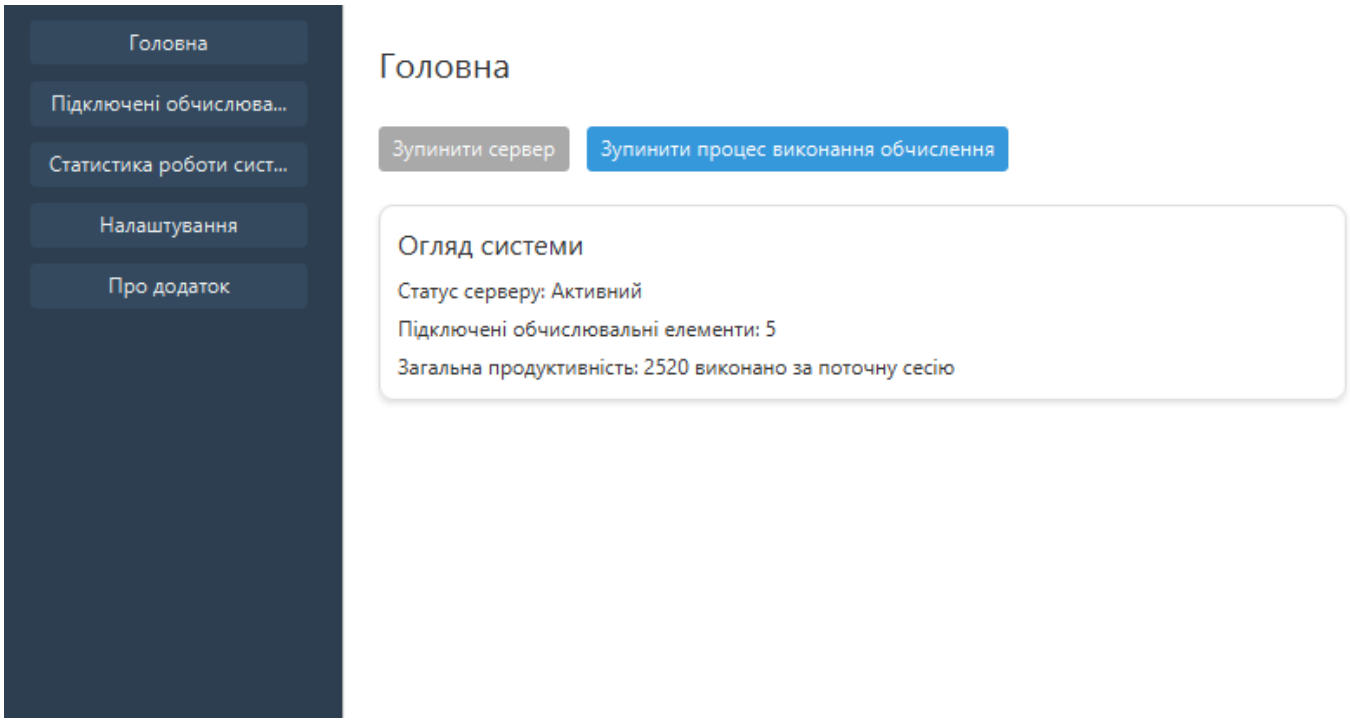


Рисунок 4.1 – Початковий екран програмної частини ЦС.

Для реалізації програмного застосунку, призначеного для ОЕ, були використані переважно ті самі технології, що і для центрального сервера, зокрема Java, JavaFX, FXML та засоби роботи з мережевими сокетами (Socket). Із допомогою JavaFX

сформований графічний інтерфейс, що дає змогу візуалізувати поточний стан програми та її активність в системі. Крім основних засобів, програмна частина ОЕ використовує бібліотеку oshi-core, яка призначена для зчитування інформації про апаратне та програмне забезпечення середовища. Зокрема, збираються дані, які необхідні для роботи запропонованого модуля довіри в розділі 3. Також, у проекті використовується бібліотека jackson-databind, що забезпечує зручну роботу із даними у форматі JSON. Вона використовується переважно в проміжному опрацюванні даних, під час функціонування застосунку. Аналогічно до серверного застосунку, для керування зовнішніми залежностями, завантаженням додаткових бібліотек та формування фінального JAR-файлу використовується система Maven. На початковому екрані застосунку (рис. 4.2) користувачеві надається спеціальна форма для проходження авторизації в системі. Після успішної авторизації центральний сервер надсилає відповідне сповіщення до застосунку ОЕ, внаслідок чого він переходить в режим очікування завдання на виконання. У цей момент активуються усі інші елементи графічного інтерфейсу, що надає можливість користувачу керувати процесами, пов’язаними з виконанням обчислень.

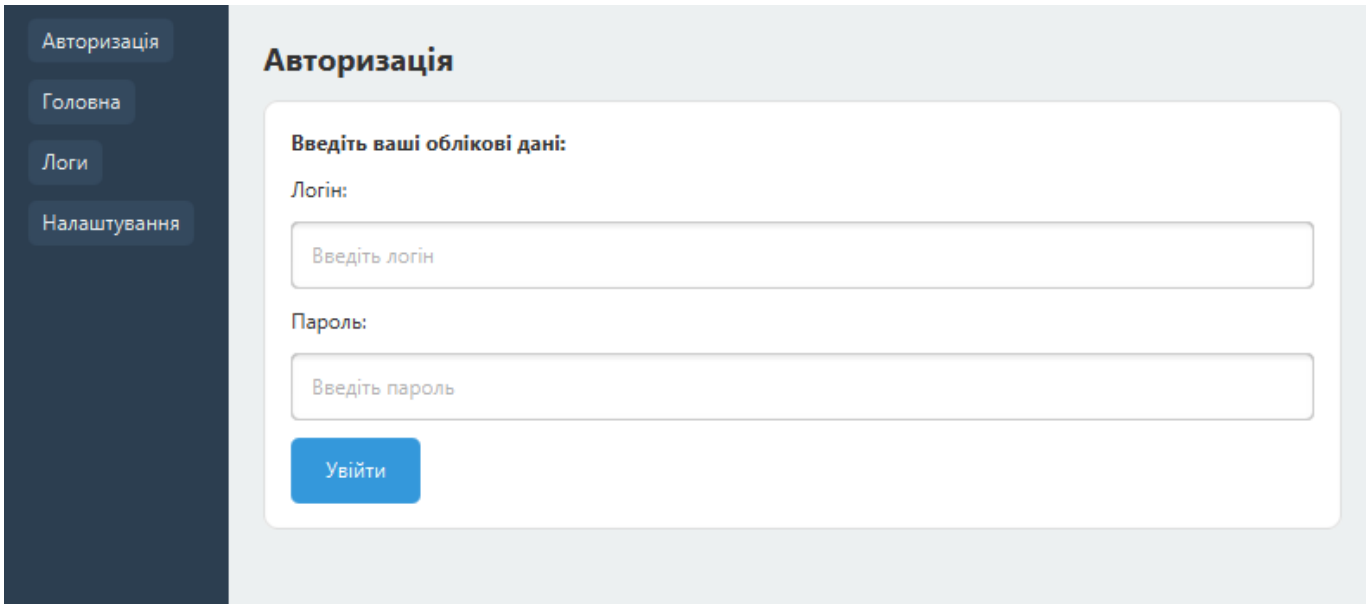


Рисунок 4.2 – Початковий екран програмної частини ОЕ

Взаємодія серверного застосунку з множиною клієнтських застосунків ОЕ формує повнофункціональну грід-обчислювальну систему, яка в разі наявності

активних задач виконує обчислення ітеративно та в автономному режимі, без необхідності постійного втручання як адміністратора, так і користувачів.

4.1.2 Функціонування розподіленої системи виявлення

Коли програмна частина ЦС перебуває в активному стані, а адміністратор активує його основний функціонал – зокрема мережеві засоби, які відповідають за очікування підключень ОЕ – каскадно запускаються інші важливі програмні компоненти. Серед них: модулі зчитування доступних ІП для аналіз, формування наступної робочої ітерації, а також створення списку дублюючих завдань. Для моніторингу процесу виконання обчислень було розроблено два спеціалізованих екрани додатку адміністратора, а для перемикання між ними здійснюється через інтегроване меню.

Екран «Підключені обчислювальні елементи» (рис. 4.3), відповідає за відображення адміністратору інформації про поточний стан підключених ОЕ системи. Верхня частина представляє собою таблицю, яка відображає список ОЕ із зазначенням їхнього ідентифікатора, який формується під час реєстрації, його станом, режим роботи (що, визначається за допомогою моделі довіри), кількості виконаних задач, а також MAC-адреса. Це дозволяє оцінити поточну активність системи.

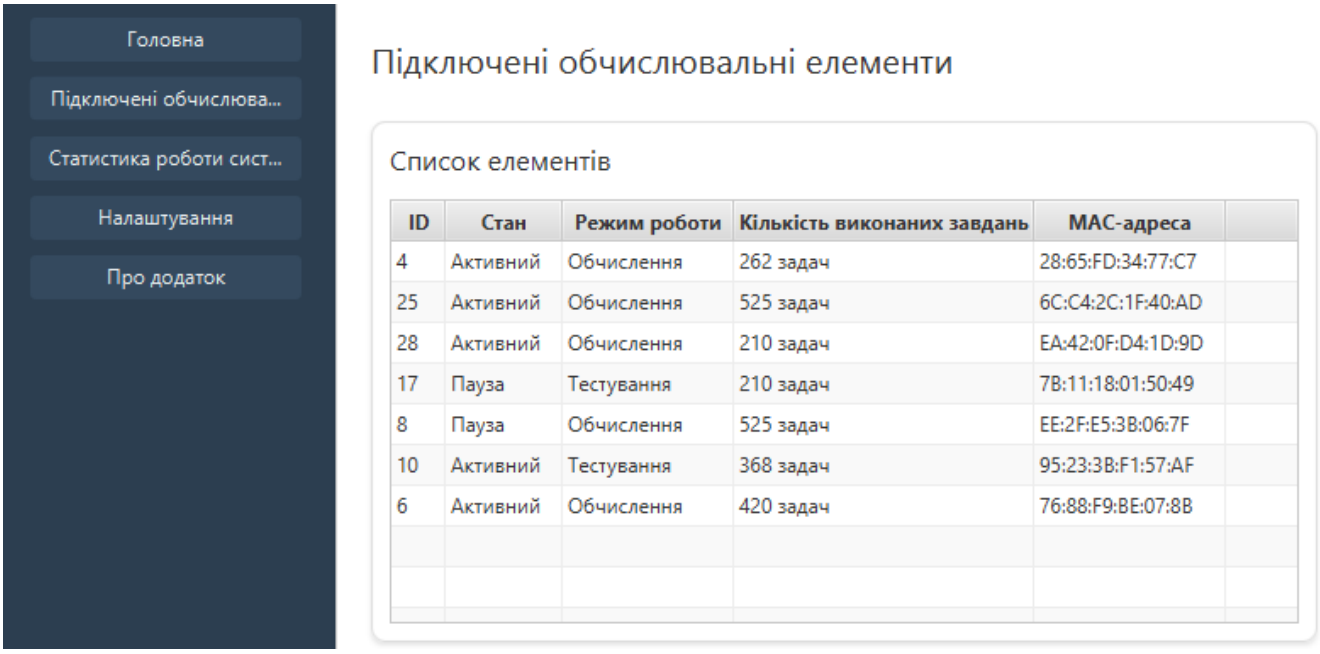


Рисунок 4.3 – Екран «Підключені обчислювальні елементи» програмної частини ЦС

Окрім цього, тут представлено результати продуктивності активних ОЕ, що виражаються через кількість виконаних задач за поточну робочу сесію. Відповідні показники візуалізуються у вигляді стовпчикової діаграми, яка дозволяє наочно порівняти продуктивність усіх активних ОЕ, а також в разі необхідності – внести зміни в процес розподілу завдань.

Наступний екран «Статистика роботи системи» (рис. 2.4), призначений для моніторингу ефективності функціонування обчислювальної системи за поточний період. У верхній частині інтерфейсу представлені узагальнені показники робочої сесії, а саме: загальна кількість виконаних задач, продуктивність виконання задач та ефективність використання обчислювальної потужності. Ці значення дозволяють оцінити ступінь завантаження системи та її результативність.

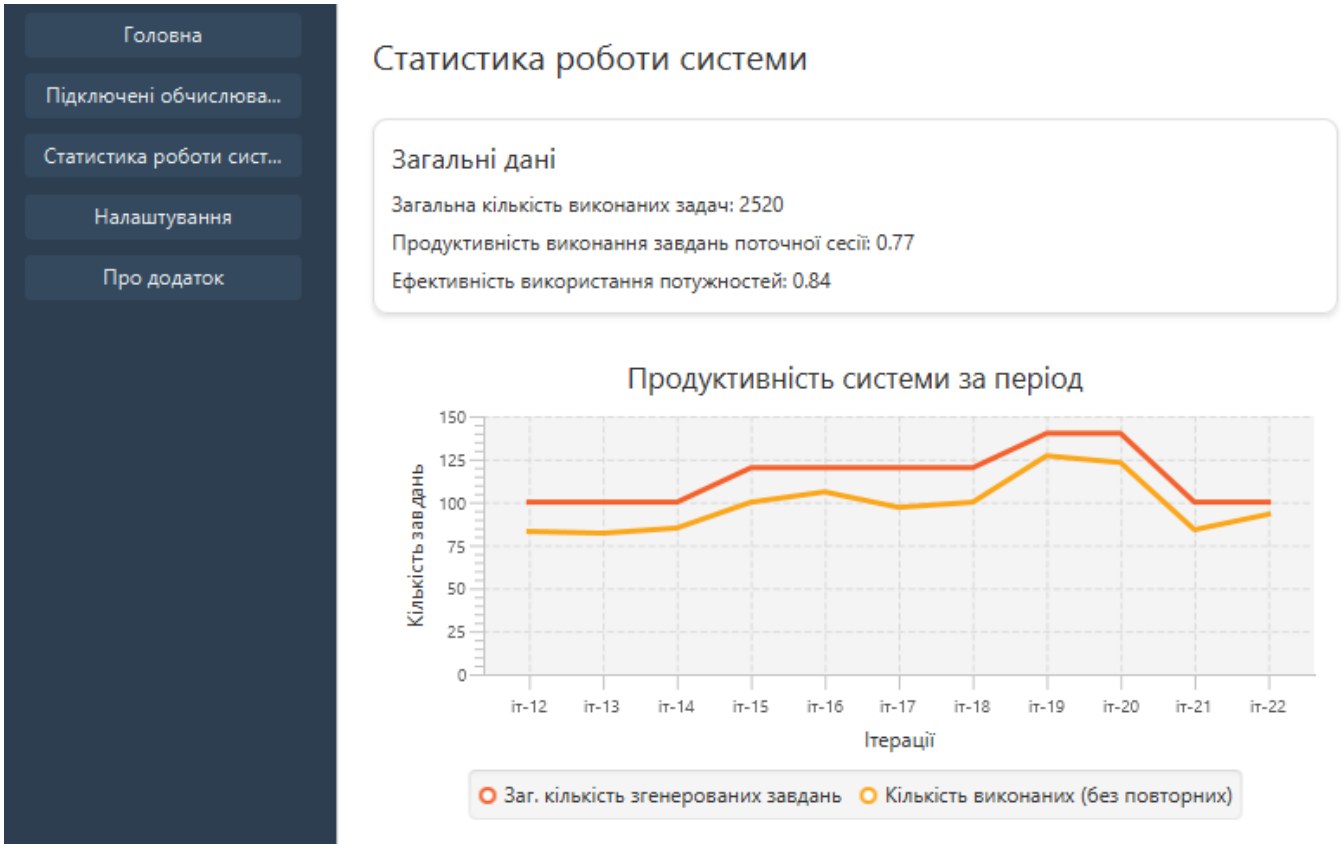


Рисунок 4.4 – Екран «Статистика роботи системи» програмної частини ЦС

Нижче подано лінійну діаграму, яка демонструє динаміку продуктивності системи впродовж послідовних робочих ітерацій. Візуалізація побудована на основі двох наборів даних. Перший відображає кількість задач, що були передані до розподіленої системи на виконання, а другий – кількість задач, успішно виконаних

без необхідності додаткової перевірки чи повторного призначення. В ідеальному випадку обидві криві збігаються, що свідчить про відсутність розбіжності між ОЕ під час одночасного виконання одних і тих самих задач. У такому випадку ОЕ під час узгодження результатів не вступають у конфлікт, що виключає необхідність повторних обчислень. В цілому, така візуалізація дозволяє виявити потенційні втрати продуктивності, що пов’язані із невиконання окремих завдань.

Наступний екран ілюструє розділ «Налаштування» (рис. 4.5), який надає адміністратору можливість конфігурувати параметри виконання обчислень у системі з акцентом на контроль їхньої коректності. У разі активації режиму «Без повторень», центральний сервер не виконує жодних перевірок результатів обчислень. Такий підхід є доцільним під час розгортання системи на власних гетерогенних ОЕ, де відсутні сумніви щодо достовірності обчислень.

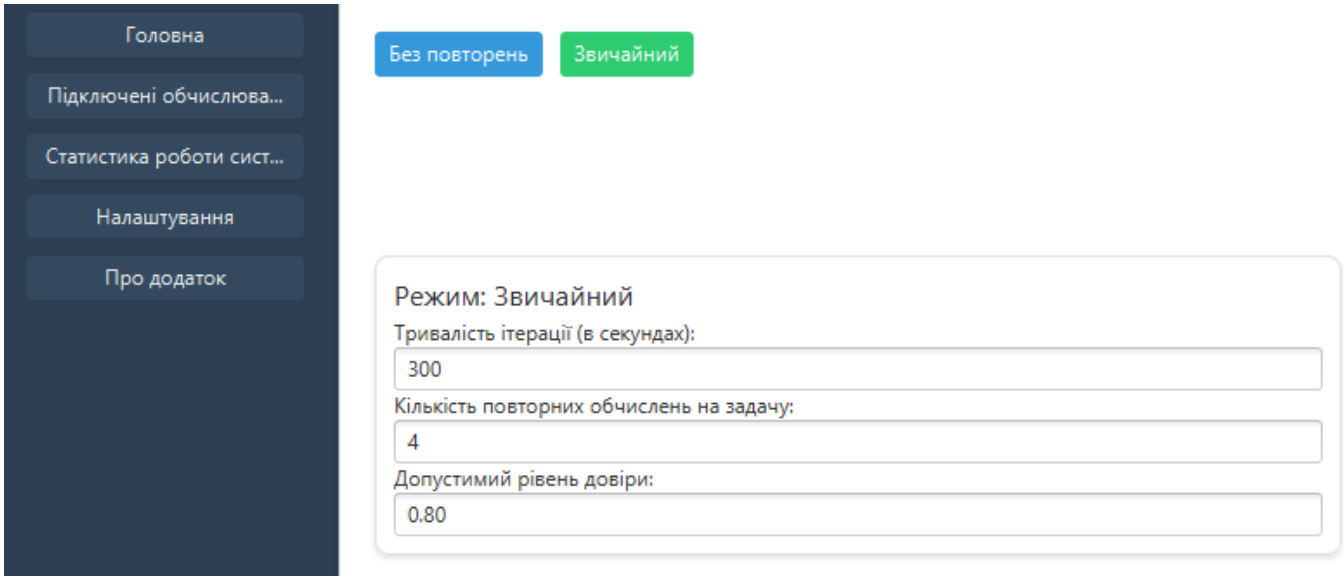


Рисунок 4.5 – Екран «Налаштування» програмної частини ЦС

У поточному вигляді цього екрану активним є режим «Звичайний», для якого доступні такі параметри конфігурації: тривалість ітерації, яка визначає часовий інтервал, протягом якого система виконує поставлені задачі на активних обчислювальних елементах перед переходом до наступної ітерації; кількість повторних обчислень на задачу, яка задає число обов’язкових перевірок однієї й тієї самої задачі, необхідних для підтвердження правильності її виконання; допустимий рівень довіри, що задається числовим коефіцієнтом у межах від 0 до 1, який визначає

мінімальний рейтинг ОЕ, необхідний для присвоєння йому статусу «Довірений обчислювальний елемент». Усі ці параметри є критичними для забезпечення надійності та якості виконання обчислень у середовищі, де можуть виникати конфлікти або похибки, пов’язані з можливою непередбачуваною поведінкою окреми ОЕ. Щодо застосунку для ОЕ, після успішної авторизації користувача система автоматично переводить його на екран «Головна» (рис. 4.6), який надає можливість керувати участю у процесі розподілених обчислень. У верхній частині інтерфейсу розміщено дві кнопки «Підключитись» та «Відключитись», які дозволяють відповідно активувати або припинити процес обчислень. Таким чином, навіть будучи авторизованим, користувач має можливість самостійно вирішувати, чи надавати свої ресурси системі в поточний момент.

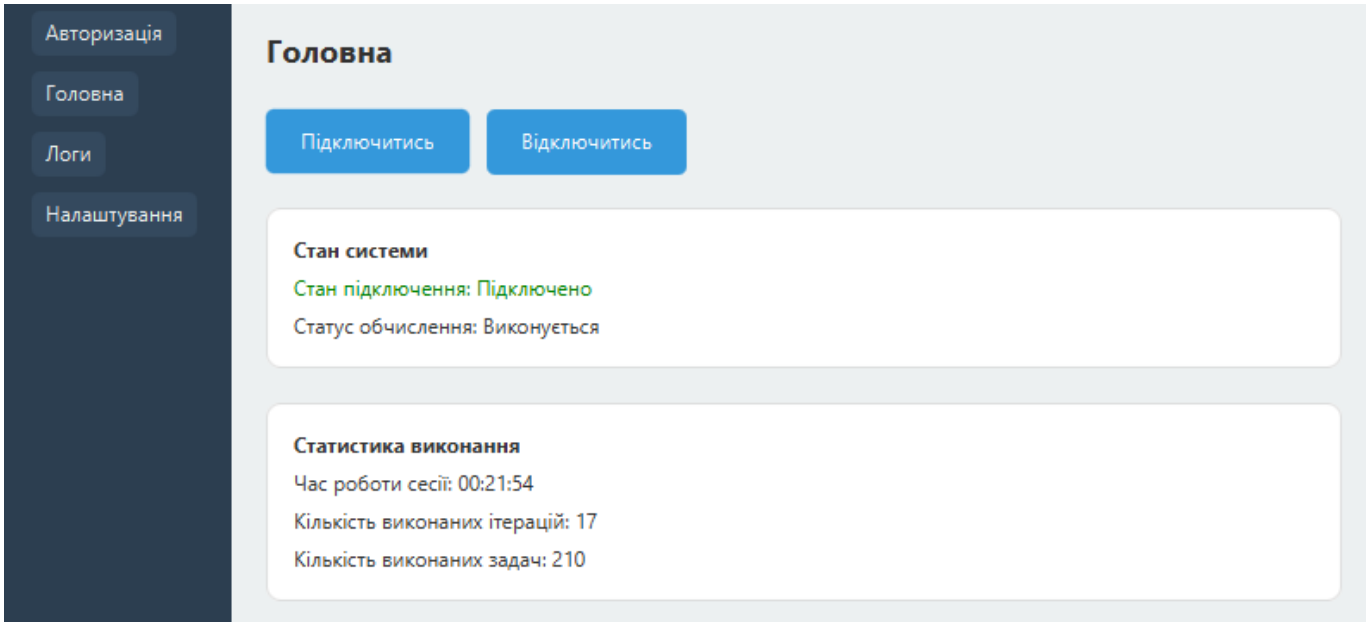


Рисунок 4.6 – Екран «Головна» програмної частини ОЕ

У нижній частині екрана також представлено статистику виконання, яка містить дані про тривалість поточної робочої сесії, кількість виконаних ітерацій та загальну кількість виконаних задач. Це дозволяє користувачеві оцінити свій внесок в обчислювальний процес системи.

Окрім цього, користувачеві доступні також екрани «Логи» та «Налаштування». Перший виконує довідкову функцію: він містить сервісні повідомлення, які періодично генеруються під час роботи застосунку. Зокрема, тут фіксується інформація про участь ОЕ в поточній робочій ітерації, зміну його ролі в системі, а

також повідомлення про помилки, пов’язані з передачею даних, виконанням задач та іншими технічними подіями. Щодо розділу «Налаштування», програмна частина ОЕ надає можливість користувачеві задати ключові параметри роботи клієнта, зокрема адресу сервера для підключення до розподіленої системи, інтервал оновлення статистики, бажаний рівень деталізації логів, а також мову інтерфейсу.

4.1.3 Мережевий протокол взаємодії елементів системи

Класичний підхід, який базується на моделі «один потік – одне з’єднання», не завжди є ефективним рішенням для побудови масштабованих грід-обчислювальних систем. Основним обмеженням є високе споживання пам’яті та інші ресурсні витрати, зумовлені специфікою використання класичних потоків. Кожен такий потік, використовує власний стек та контекст виконання, що ускладнює масштабування та обмежує кількість одночасних з’єднань, які може підтримувати система. В межах розробленої системи під масштабуванням приймається кількість ОЕ, що можуть бути підключені до одного серверного застосунку. І цей аспект є критичним, оскільки грід-обчислювальні системи базуються на ідеї залучення якомога більшої кількості ОЕ до процесу розподілених обчислень. Тому, для вирішення цієї задачі в проєкті було використано Project Loom – технологію, що розвивається в межах платформи Java, яка призначена для підтримки великої кількості конкурентних задач без суттєвого навантаження на ресурси. Основною перевагою Loom є використання віртуальних потоків, що не блокують фізичні потоки під час очікування операцій введення/виведення. Це дозволяє ефективно організовувати обробку великої кількості паралельних з’єднань. З огляду на характер функціонування системи, де переважають пасивні очікування між короткими фазами активної взаємодії, застосування віртуальних потоків є цілком виправданим. Типовий сценарій взаємодії між центральним сервером і ОЕ зводиться до двох основних подій: передавання завдання на обчислення та отримання результату виконання. Окрім цього, періодично здійснюються серверні запити, зокрема перевірка доступності ОЕ або ініціалізація зміни протоколу комунікації відповідно до поточної ролі ОЕ в системі. Приклад

типової взаємодії між учасниками системи представлено на діаграмі послідовності (рис. 4.7). Подібна логіка реалізована й на стороні ОЕ, де його програмна частина ініціалізує окремий потік для виконання обчислень. При цьому основний потік залишається відповідальним за прослуховування повідомлень, що надходять від центрального сервера, зокрема задач на виконання та сервісних запитів.

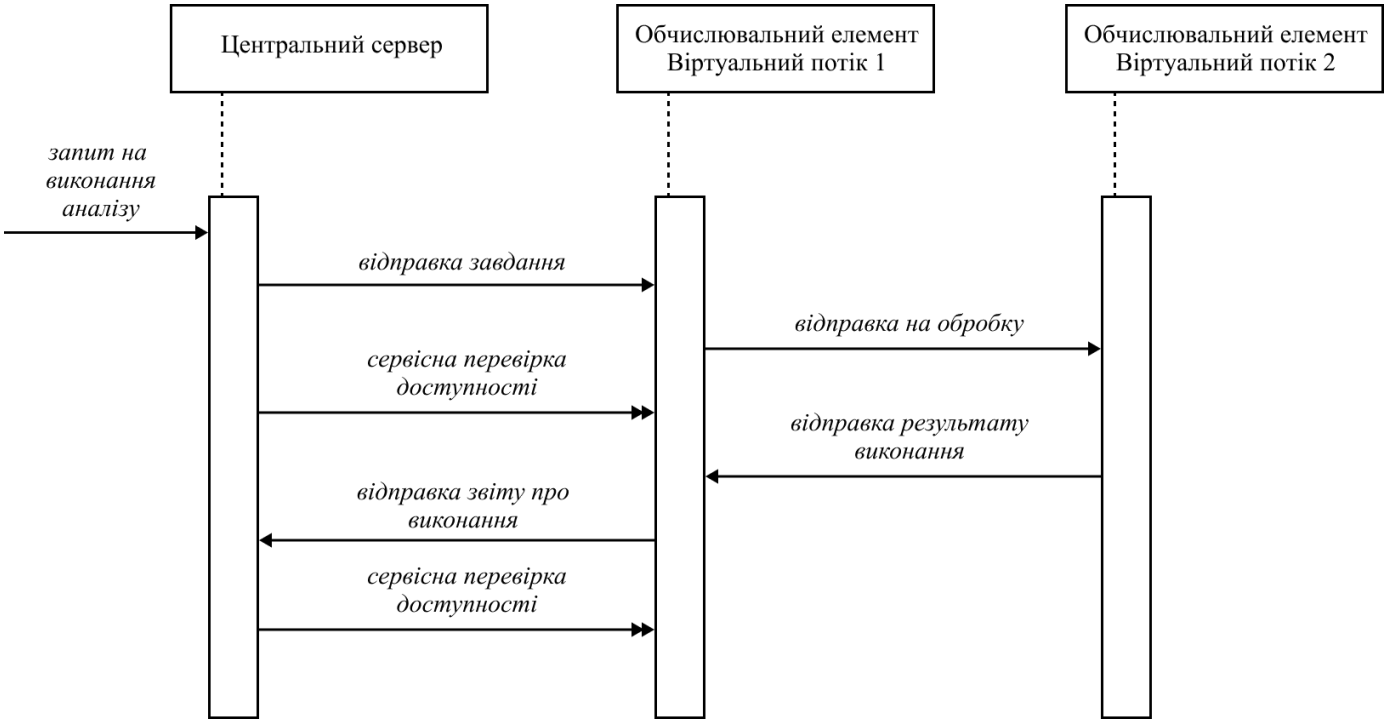


Рисунок 4.7 – Мережевий протокол в обчислювальній системі

Представлена діаграма послідовності ілюструє базовий сценарій взаємодії між серверним застосунком та одним ОЕ. Ця модель є типовою і застосовується до всіх активних ОЕ, що підключені до системи. Сценарій розпочинається зі встановлення мережевого з’єднання, після чого центральний сервер ініціює надсилання задачі до активного ОЕ. Після отримання задачі, ОЕ створює віртуальний потік, який відповідає виключно за виконання цієї задачі. Після завершення обчислень відповідний віртуальний потік автоматично завершує своє існування. У той же час головний віртуальний потік залишається активним і перебуває в режимі очікування, обробляючи сервісні повідомлення, що надходять від центрального сервера – зокрема, запити на перевірку активності або зміни в протоколі взаємодії. Завдяки такій організації забезпечується асинхронне виконання задач, що не блокує основну логіку взаємодії із центральним сервером.

Уся взаємодія з кожним ОЕ відбувається паралельно, завдяки чому програмна частина ЦС ініціалізує окремий віртуальний потік для кожного активного з'єднання. В процесі реалізації логіки розподілу задач був сформований розглянутий вище мережевий протокол, на основі якого були реалізовані модифіковані сценарії мережевої комунікації, що частково повторюють загальну модель, але містять відмінності для кожної ролі ОЕ. Такий підхід зумовлений необхідністю розмежуванням доступу до актуального набору задач на виконання, що в цілому виступає як захист обчислень від ненадійних ОЕ. В результаті довірені та звичайні ОЕ працюють із задачами, що потребують обчислень, тоді як інші ролі отримують задачі, які вже були розв'язані. Це дозволяє перевіряти стабільність їхньої роботи, без ризику шкоди для результатів, і поступово підвищувати рейтинг таких клієнтів. Окрім того, модифікація мережевого протоколу надала можливість реалізувати незалежну логіку зміни рейтингу, для різних ролей ОЕ. Це створює потенціал для розширення алгоритмів довіри в майбутньому – наприклад, через більш гнучкі формули.

Окрім обслуговування мережевих з'єднань із ОЕ, віртуальні потоки також знайшли активне застосування в реалізації низки фонових процесів, які забезпечують безперервну підтримку стабільної роботи системи та збереження її цілісності впродовж всього життєвого циклу. Одним із таких процесів є перевірка виконаних підзадач, яка реалізується як окремий віртуальний потік і дає змогу своєчасно аналізувати завершені обчислення та виявляти невідповідність результатів. На основі цих перевірок можливе ініціювання формування звітів про виконання поставленої задачі для користувача. За аналогічною логікою організовано механізм очікування нових ОЕ – він здебільшого перебуває в пасивному режимі, але готовий до активації у разі надходження нового з'єднання. Окрім цього, у вигляді окремих потоків реалізовано актуалізацію внутрішньої бази даних про кожен підключений ОЕ, що необхідно для підтримки коректної моделі довіри. Застосування віртуальних потоків також може бути ефективно задіяне в подальшому для організації підготовки до наступної робочої ітерації, що дає змогу формувати чергу задач ще до завершення поточної фази обчислень. Такий підхід дозволяє скоротити простої між ітераціями.

Таким чином, сформована система [121] має функціональні особливості, які дозволяють організувати процес розподіленого аналізу ІП, а саме: забезпечення інтерфейсу для комунікації та організації підключених ОЕ; формування черги активних задач із урахуванням повторних обчислень та динамічності середовища виконання; формування ітеративного процесу розподілу завдань між активними ОЕ в поточний момент часу; проведення аналізу виконаних обчислень; формування множини модифікованих ізольованих середовищ із залученням ОЕ для кожної ітерації. Лістинг програмних модулів представлено у додатку Г.

4.2 Постановка експериментів та функціонування системи

Для проведення експериментів представленої системи в якості елементів для аналізу використовувались розроблені штучні зразки моделей ІП. Ці зразки представляють собою програмний об’єкт, який відповідає моделі ІП. Штучні ІП подані у вигляді відповідних програмних об’єктів і містять усі необхідні складові. Окрім ІП, було сформовано ще декілька таких зразків, які структурно відповідають розробленим моделям, але при цьому не містять методів уникнення від виявлення чи зловмисного прояву. Вони будуть корисні для дослідження їх поведінки в різних модифікованих середовищах. В межах дослідження впливу визначених моделей ІП було також реалізовано базову пісочницю, яка генерує модифіковане ізольоване середовище згідно визначених методів виявлення емуляції. Розроблена пісочниця використовує базовий емулятор, який емулює частину інструкцій архітектури x86_64, пам’ять програми, а також процес впливу інструкцій на пам’ять. Разом із реалізацією емуляції виконання програмних переривань, було використано бібліотеку java.security, яка забезпечує перетворення інформації шляхом обчислення хеш-значення за алгоритмом SHA-256.

В одному із експериментів, мета якого була зробити оцінку впливу використання програмних переривань на загальну швидкодію опрацювання інструкцій (виконання програм). Для цього використовувався визначений набір інструкцій, а також дві його модифіковані версії. Ці версії базувались на використанні

методів поліморфізму, а саме: вставка коду $i_{p_{ci}}$; перепризначення регістрів $i_{p_{rr}}$ по відношенню до оригінальної програми. В таблиці результатів експерименту та графіку порівняння, вони будуть представлені як оригінальний код (original code; OC), вставка коду (code insertion; CI) та перепризначення регістрів (register reassignment; RR).

Для оцінювання впливу на швидкодію обробки визначених програм для програмних переривань було сформовано три основні стратегії, які визначають момент їх виконання з метою формування поведінки виконання.

Визначені стратегії такі: покрокова для отримання хеш-значення після кожної виконаної інструкції; кожна третя для отримання хеш-значення після кожної третьої виконаної інструкції; тільки остання для отримання хеш-значення лише після виконання усіх інструкцій. В таблиці результатів експерименту та графіку порівняння, вони будуть представлені покроково (step by step; sbs), кожною третьою (every third; et), тільки останньою (last only; lo). В межах цього експерименту усі набори кодів виконувались згідно із зазначеними стратегіями виконання переривань, тому згідно із визначених позначок в таблиці і графіку порівняння будуть фігурувати пояснення такі як RR-sbs, що визначає як визначений набор інструкцій так і стратегію формування поведінки програми.

Таким чином, для експериментів кожен список інструкцій виконувався з усіма запропонованими стратегіями, що призвело до отримання наступних результатів, які подані в табл. 4.1.

Хоча усі набори інструкцій в кінці виконання показували однаковий результат виконання, але відхилення від оригінальної програми фіксується через представлену поведінку. Це відхилення може свідчити про використання методів поліморфізму.

Для оцінювання впливу застосування програмних переривань на швидкість виконання програми було використано п’ять подібних за обчислювальною складністю програм, які виконувались із використанням визначених стратегій. Результати порівняння зображено на рис. 4.8.

Таблиця 4.1

Поведінки програм із використанням різних стратегій переривань

Прог-рама	Значення стану
OC-sbs	cbd0b691087f677f341360a04c8b9e7c99850f6044c78ecd5bee1c7e4a6cadac
OC-et	0d1eeca3ab71b457d9738a736ad6a7e8428972de8238a012341d04ce36a4a757
OC-lo	db6d84b74ad1b0f174d6fbcb756c6f066959ccb92fe7cfa26815e47172c432f5
CI-sbs	e706331f4baf5567bf4a5ccc4f4d30888676a68d9cd4e6f5c513375f0cbf4730
CI-et	68cfca2635c8cc931643e47c575d54097c9d1e5e84a7735e852c2c713726627a
CI-lo	1ddcaf920c16528a1126d94126c65cb13e5c6ddac9168a85736bb190dde27f01
RR-sbs	f7ac5e97b94c9aa307d9aee5a0958190fc540847282cdffd5e57925f44c2eab5
RR-et	feebed55a581e48264b08c867eeaf53df363a85aca46a37b6448ae878c0e5517
RR-lo	659fcc19c497c56356b01cf210fa44c49f0d15c2f5b13776aa87428c5759ef8b

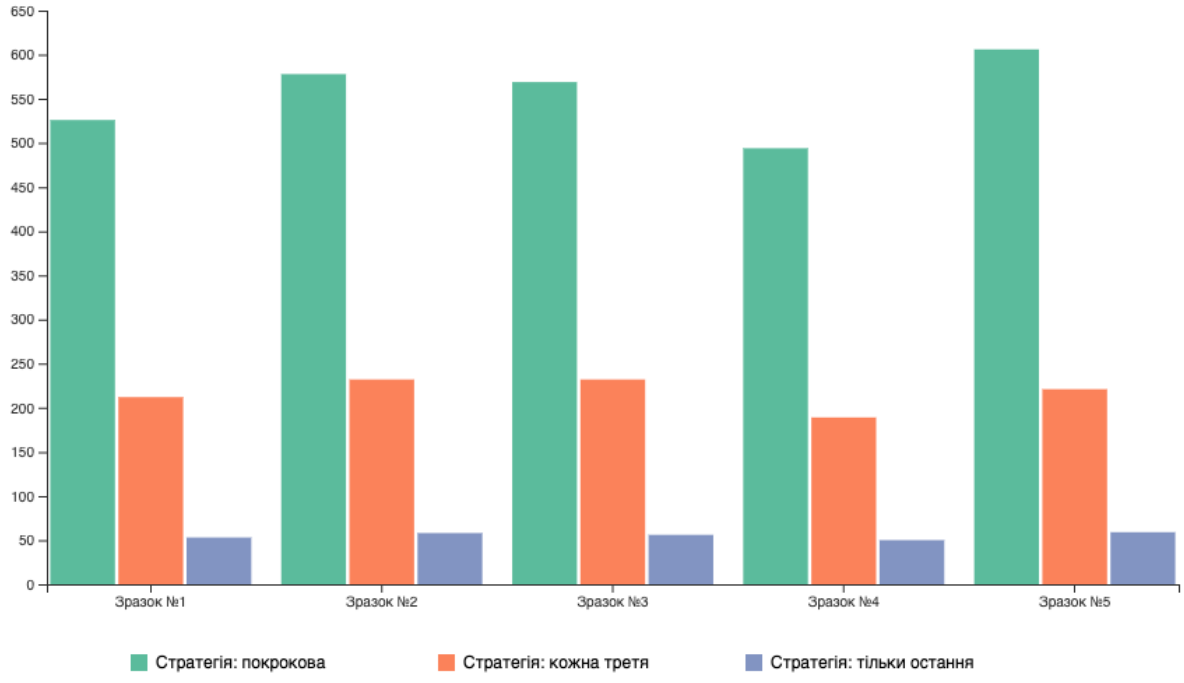


Рисунок 4.8 – Вплив застосування переривань на час виконання програми

З наведеного аналізу можна зробити висновок, що часте використання переривань суттєво впливає на швидкодію програми. Третя стратегія, яка передбачає лише одне зчитування, фактично відображає реальний час виконання набору інструкцій без зовнішнього втручання. Водночас не варто занадто зменшувати

інтервал виконання переривань, оскільки при достатньому рівні обфускації артефакти, пов’язані із їх використанням, можуть бути повністю приховані. Тому, вибір параметра частоти переривань має базуватись на компромісі між точністю виявлення аномалій та мінімальним впливом на продуктивність програми.

Також, в межах проведених експериментів здійснювалось окреме дослідження, метою якого було визначення мінімально необхідної кількості призначень для безпечного виконання розподіленого обчислення з урахуванням особливостей грід-систем. Зокрема, воно сфокусоване на розподіл задач між ОЕ з різним рівнем довіри, що є важливим чинником у системах із відкритою архітектурою. Для цього було використано п’ять ОЕ, кожен з яких мав попередньо встановлений статус: «довірений» або «звичайний».

В цьому експерименті, також, враховувалось ймовірне некоректне виконання обчислень, яке було змодельоване з рівнями помилки 10% та 25% відповідно. Оскільки тестування здійснювалось із залученням доступних комп’ютерних станцій, то для моделювання хибного або спотвореного виконання в програмна частина ОЕ було додано відповідну логіку, яка штучно змінювала результати обчислень згідно із заданим рівнем похибки. Це дозволило змоделювати поведінку недостовірних або потенційно скомпрометованих елементів у системі.

На графіку, який зображено на рис 4.9, наведено результати чотирьох експериментальних запусків, у межах яких було реалізовано два основні сценарії. У першому з них використовувався один довірений та чотири звичайних ОЕ, а в другому – два довірених та три звичайних. Обидва сценарії відтворювались при рівнях похибки 10% та 25%, що дозволило порівняти ефективність системи за різних умов довіри до учасників.

Графіки функцій на рис. 4.9 відображають залежність кількості робочих ітерацій, необхідних для завершення обчислень, від загальної кількості унікальних задач у розподіленому середовищі. Аналіз результатів показує що на ефективність виконання значно впливає як рівень довіри до ОЕ, так і ймовірність виникнення помилок під час обчислень. Зокрема, збільшення частки помилок призводить до суттєвого зростання кількості повторних обчислень, що безпосередньо впливає на

загальну кількість необхідних ітерацій. Це, своєю чергою, вказує на важливість впровадження механізмів виявлення недостовірних результатів і оптимізації перевірок. Окрім цього, порівняння сценаріїв із різною кількістю довірених ОЕ свідчить про те, що залучення більшої кількості довірених ОЕ дозволяє зменшити загальне навантаження на систему. Це досягається шляхом зниження кількості дублювань задач і скорочення загальної кількості ітерацій.

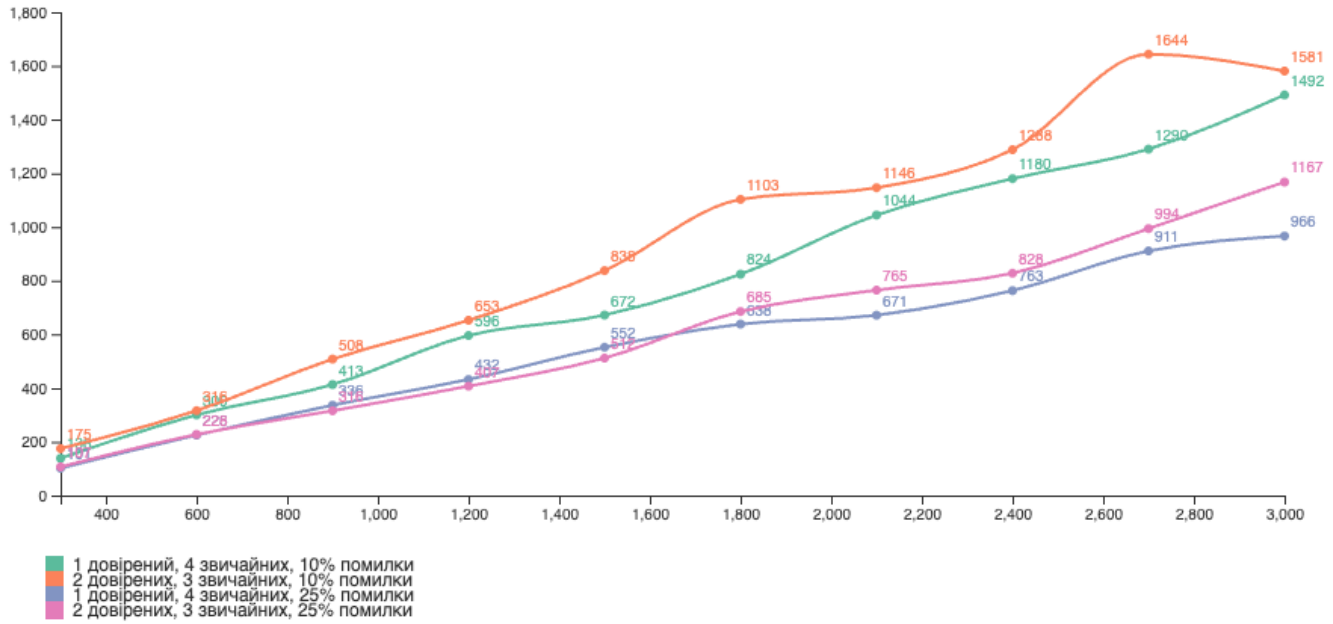


Рисунок 4.9 – Кількість робочих ітерацій у грід-системі залежно від довірених ОЕ та частки помилкових виконань завдань

По осі відображено кількість унікальних завдань, поставлених перед системою, по осі ординат – кількість робочих ітерацій, необхідних для їх вирішення системою за умови, що кожен ОЕ виконує лише одне завдання за одну ітерацію. Таким чином, графік підтверджує доцільність застосування рольової моделі та адаптивного механізму формування довіри як одного із ключових напрямків подальшого розвитку системи.

Наступним експериментом було дослідження ефективності використання запропонованого алгоритму розподілу завдань, і для його оцінки було реалізовано певну кількість додаткових алгоритмів планування, серед яких використовувались – Min-Min, FIFO та Min-Load. Такий підхід дозволив здійснити порівняльний аналіз продуктивності кожного з методів у єдиному тестовому середовищі. Усі алгоритми

були протестовані на ідентичних наборах даних, що складались з п’яти наборів задач, кожен з яких включав до 100 завдань для розподілу між п’ятьма ОЕ.

Результати експерименту наведені на рис. 4.10, де подано загальний час виконання всього набору завдань для кожного з алгоритмів. Аналіз отриманих даних свідчить про те, що запропонований підхід забезпечує нижчий час виконання в більшості тестових випадків. Зокрема, на окремих наборах задач покращення становило від 5% до 7% порівняно з найближчим алгоритмом. Це вказує на здатність методу ефективно враховувати як поточне навантаження на ОЕ, так і на складність задач при їх розподілі.

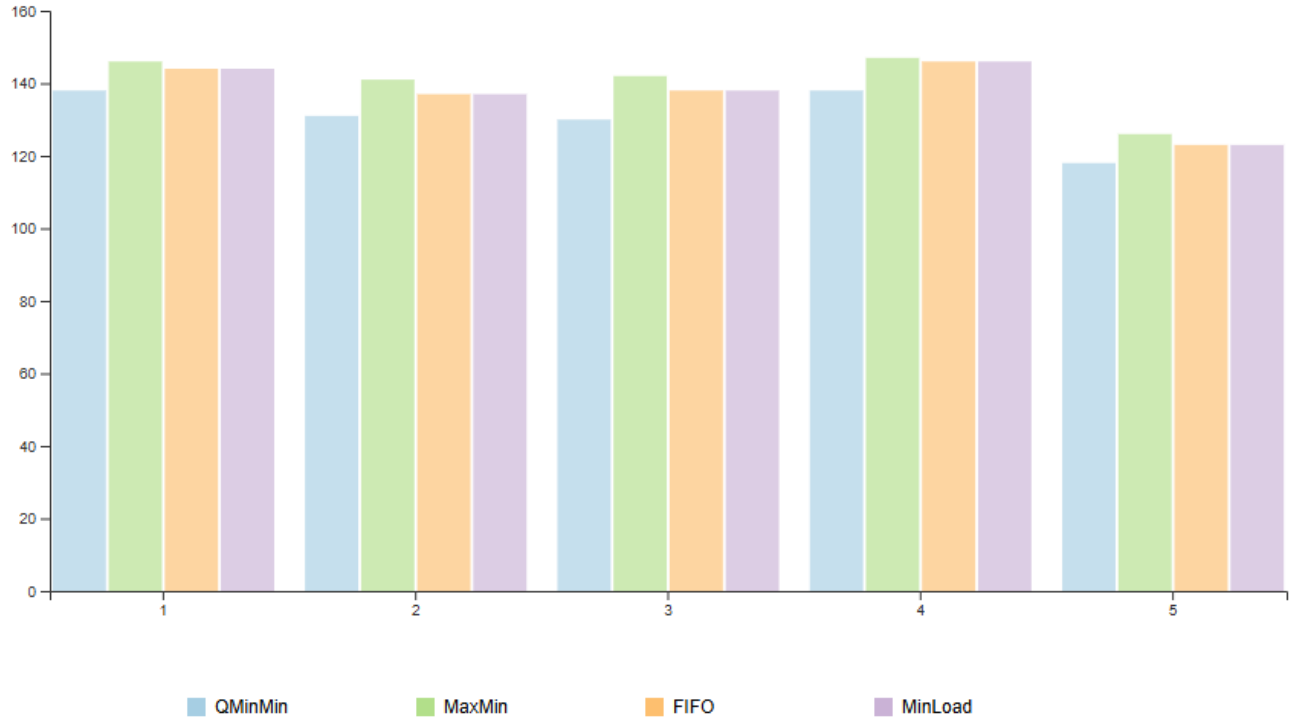


Рисунок 4.10 – Порівняльний аналіз алгоритмів планування обчислень в розподіленій системі обчислень

По осі абсцис розміщені проведені 5 експериментів, кожен з яких містив визначену вибірку задач, а по осі ординат загальну кількість робочих ітерацій які були необхідні для вирішення поставленого завдання із використанням різних алгоритмів планування.

Отримані результати підтверджують можливість застосування запропонованого алгоритму в грід-обчислювальних системах, що характеризуються

динамічним показником обчислювальної потужності для виконання задач з різною обчислювальною складністю. Порівняно з іншими підходами, розроблений метод забезпечує не лише приріст у продуктивності, але і враховує порядок виконання задач.

Наступний експеримент був спрямований на дослідження ефективності застосування модифікованої моделі довіри, що базується на використанні нечіткої логіки для визначення ролі ОЕ в системі. У рамках реалізації цього експерименту була використана відповідна бібліотека для збору відомостей про апаратні та програмні характеристики ОЕ. Зібрані дані стали основою для формування двох структурованих таблиць, у яких відображено ключові параметри, що впливали на коригування рейтингу кожного елемента. Залежно від результатів аналізу цих характеристик, система могла динамічно змінювати роль ОЕ – наприклад підвищувати його до рівня «довіреного» , або знизити у разі виявлення помилки в обчисленнях. Приклади відповідних таблиць наведено у табл. 4.2 та табл. 4.3.

Таблиця 4.2

Набір інформації про апаратне та програмне забезпечення

id	b_{os}	b_{cpua}	b_{cpum}	b_{cpus}	b_{cpun}	b_{ram}	b_{dir}	b_{mac}
17	win	x64	Intel core i5 11400	2600	6	16	C://usr1	9c:b5:3c: 9f:f1:4f
19	win	x64	Intel core i3 12100	3300	4	8	C:/admin	0b:c0:21: 87:58:57

Таблиця 4.3

Набір інформації про показники, які характеризують поведінку ОЕ

id	b_{taskc}	b_{taskp}	b_{tasks}	b_{taskr}	b_{prod}
11	2000	6100	22	33	3.05
14	1500	4400	19	32	2.93

В рамках експериментальної частини було проведення дослідження яке стосувалось декількох десятків наборів задач, що пов’язані із виявлення зловмисного

прояву в у наборах інструкцій. В рамках дослідження ефективності модифікованої моделі, проводилось 5 основних експериментів. Для підвищення достовірності результатів дослідження включало порівняння двох основних підходів: базовий сценарій із використанням звичайного підходу, який передбачає повторні перевірки усіх задач; альтернативний сценарій, в якому використовувалась модифікована версія, яка включала рольову складову.

Під час експерименту було використано визначену кількість ОЕ, яким був назначений статус як новий учасник розподіленої системи. Виконуючи задачі, їх рейтинг поступово збільшувався, що дозволило деяким ОЕ отримати статус «довірений». Результати наведено в табл. 4.4, яка демонструє кількість повторних перевірок у кожному з підходів, також отримані розраховані показники економії обчислювальних ресурсів підключених ОЕ до системи.

Таблиця 4.4

Результати проведення експерименту із модифікованою моделюю довіри

	Експ.1	Експ.2	Експ.3	Експ.4	Експ.5
Кількість перевірок (без модифікації)	1200	990	910	1400	860
Кількість перевірок (із модифікацією)	1130	942	854	1315	823
Зменшення перевірок	70	48	56	85	37
Економія обчислювальних ресурсів	5,83%	4.85%	6.15%	6.07%	4.30%

Отримані результати демонструють, що застосування модифікованої моделі довіри дозволило, із визначеними параметрами, зменшити кількість повторних перевірок у середньому на 4-6% (рис. 4.11), при цьому не знижуючи надійності виявлення ІП.

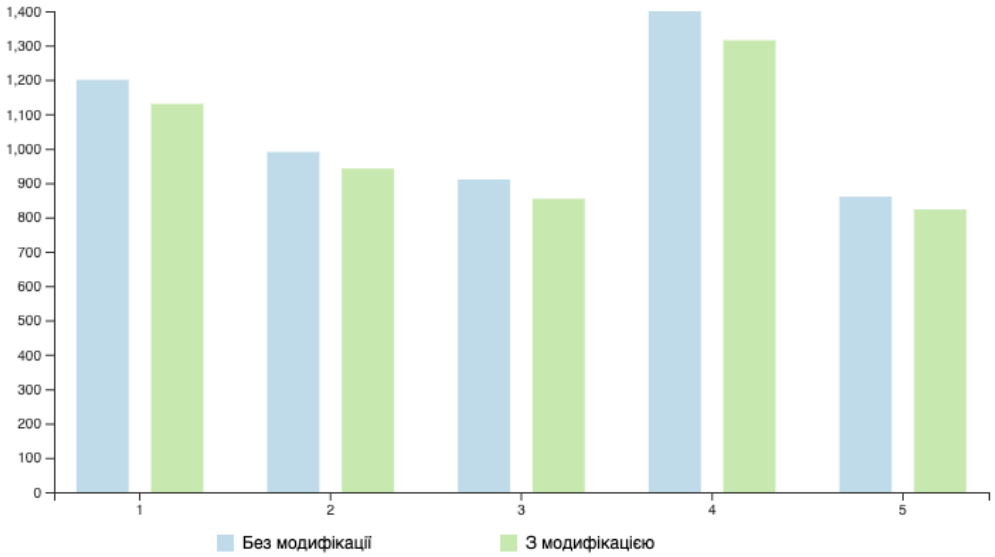


Рисунок 4.11 – Ефективність використання модифікації моделі довіри

По осі абсцис розміщені результати проведених експериментів, кожен з яких містив набір задач, які були поставлені перед системою на виконання. Щодо осі ординат, то на ній визначено кількість проведених перевірок обчислень для виконання задач в рамках одного експерименту.

Це свідчить про перспективність поєднання обчислювального аналізу з динамічним управлінням довірою в грід-обчислювальних системах, що дозволяє оптимізувати використання ресурсів системи без впливу на правильність виконання поставленої задач перед системою.

Наступний експеримент мав на меті поєднати усі представлені методи для організації розподіленої системи для виявлення зловмисного прояву в ІП. Усі розглянуті методи, допомагають приймають участь в організації в різних аспектах, які зручно подати у вигляді відповідної схеми (рис. 4.12).

На рис. 4.12 зображено компоненти ЦС та ОЕ беруть участь в використаних методах. $M1$ – метод синтезу засобів формування поведінки виконання ІП в ізольованих середовищах, $M2$ – метод організації функціонування грід-обчислювальних систем, $M3$ – метод оцінювання довіри. $M1$ використовує пісочницю і ізольоване середовище для формування поведінки і модуль аналізу виконаних завдань для оцінки наявності зловмисної поведінки в ІП. $M2$ – використовує 2 черги

виконання завдань та планувальник, а *М3* виконує оцінку в модулі довіри зважаючи на інформацію яка надходить з модуля аналізу виконаних завдань про кожного ОЕ.

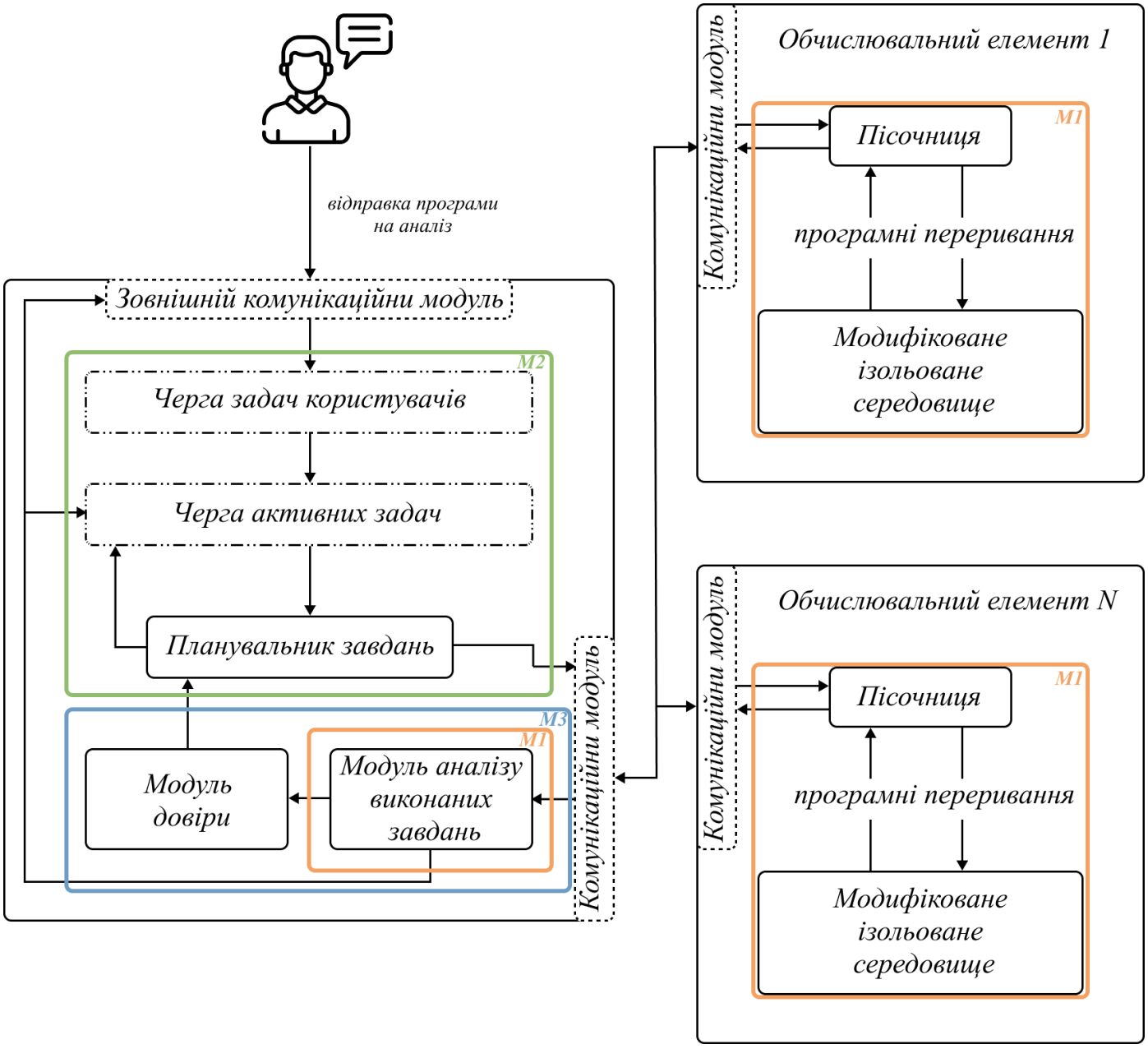


Рисунок 4.12 – Загальна схема застосування методів для організації

Для цього експерименту було згенеровано вісім моделей ІП для їх подальшого аналізу, а також множину параметрів для створення модифікованих середовищ. Основні результати поведінки представлені в табл. 4.5. Результат експерименту продемонструє вплив методів уникнення виявлення на поведінку ІП в різних модифікованих ізольованих середовищах.

Таблиця 4.5

Результати проведення експерименту виявлення ІП в розподіленій системі

	$M_1(h_{ca}, h_{hc}, h_{ia})$	$M_2(h_{ca}, h_{sc})$	$M_3(h_{sc}, h_{hc}, h_{id})$	$M_4(h_{ta}, h_{ht})$	$M_5(h_{ht}, h_{hc})$	$M_3(h_{ca}, h_{sc}, h_{ht})$	$M_7(h_{ta}, h_{ia})$	$M_8(h_{ta}, h_{id})$	
$I_1(i_{ed_{sc}}, i_{ed_{id}})$	$B^1_{I_1}$	$B^2_{I_1}$	$B^2_{I_1}$	$B^1_{I_1}$	$B^1_{I_1}$	$B^1_{I_1}$	$B^1_{I_1}$	$B^1_{I_1}$	виявлено
$I_2(i_{ed_{ht}}, i_{ed_{ia}})$	$B^1_{I_2}$	$B^2_{I_2}$	$B^2_{I_2}$	$B^1_{I_2}$	$B^1_{I_2}$	$B^1_{I_2}$	$B^1_{I_2}$	$B^2_{I_2}$	виявлено
$I_3(i_{ed_{ca}}, i_{ed_{ta}})$	$B^1_{I_3}$	$B^1_{I_3}$	$B^2_{I_3}$	$B^1_{I_3}$	$B^2_{I_3}$	$B^1_{I_3}$	$B^1_{I_3}$	$B^1_{I_3}$	виявлено
$I_4(i_{ed_{ta}})$	$B^1_{I_4}$	$B^1_{I_4}$	$B^1_{I_4}$	$B^1_{I_4}$	$B^1_{I_4}$	$B^1_{I_4}$	$B^1_{I_4}$	$B^1_{I_4}$	не виявлено
$I_5(i_{ed_{sc}})$	$B^1_{I_5}$	$B^2_{I_5}$	$B^2_{I_5}$	$B^1_{I_5}$	$B^1_{I_5}$	$B^2_{I_5}$	$B^1_{I_5}$	$B^1_{I_5}$	виявлено
$I_6(i_{ed_{it}})$	$B^1_{I_6}$	$B^1_{I_6}$	$B^1_{I_6}$	$B^1_{I_6}$	$B^1_{I_6}$	$B^1_{I_6}$	$B^1_{I_6}$	$B^1_{I_6}$	не виявлено
$I_7(\emptyset)$	$B^1_{I_7}$	$B^1_{I_7}$	$B^1_{I_7}$	$B^1_{I_7}$	$B^1_{I_7}$	$B^1_{I_7}$	$B^1_{I_7}$	$B^1_{I_7}$	не виявлено
$I_8(i_{ed_{hc}}, i_{ed_{ca}})$	$B^1_{I_8}$	$B^1_{I_8}$	$B^1_{I_8}$	$B^2_{I_8}$	$B^1_{I_8}$	$B^1_{I_8}$	$B^2_{I_8}$	$B^2_{I_8}$	виявлено

Моделі розроблених ІП представлені у першому стовпці, де для зручності в дужках позначені методи виявлення емуляції, що були застосовані в кожному зразку. Щодо модифікованих ізольованих середовищ, то їх параметри також вказані у відповідних дужках. Так як кожен рядок виступає за експерименти одного ІП в різних середовищах, то кожна комірка під середовищем визначає отриману поведінку. Щодо позначення поведінки, було вирішено представляти її як $B^i_{IS_j}$, де і — варіант поведінки виконання інфікованої програми, а j — номер розробленої моделі ЗПЗ. Ця нотація дозволяє для чистішого аналізу таблиці порівняно з представленням інформації як зашифрованих результатів. Тому, в таблиці включена та сама поведінка програми, яка залежить від параметрів, використаних для створення ізольованого середовища.

Окрім самої поведінки, запропонований метод орієнтується на нормалізовані часові показники виконання, то результати представлені в табл. 4.6. Варто врахувати, що кожна таблиця демонструє усереднене значення виконання, так як експеримент

проводився декілька разів, використовуючи ті самі параметри для модифікації середовища виконання.

Таблиця 4.6

Результати проведення експерименту виявлення ІП в розподіленій системі

	$M_1(h_{ca}, h_{hc}, h_{ia})$	$M_2(h_{ca}, h_{sc})$	$M_3(h_{sc}, h_{hc}, h_{id})$	$M_4(h_{ta}, h_{ht})$	$M_5(h_{ht}, h_{hc})$	$M_3(h_{ca}, h_{sc}, h_{ht})$	$M_7(h_{ta}, h_{ia})$	$M_8(h_{ta}, h_{id})$	
$I_1(i_{ed_{sc}}, i_{ed_{id}})$	806	797	906	819	875	848	870	898	не виявлено
$I_2(i_{ed_{ht}}, i_{ed_{ia}})$	689	735	796	730	786	750	682	698	не виявлено
$I_3(i_{ed_{ca}}, i_{ed_{ta}})$	754	714	821	775	806	699	768	801	не виявлено
$I_4(i_{ed_{ta}})$	1109	1165	1338	1146	1075	1148	1370	1398	виявлено
$I_5(i_{ed_{sc}})$	1023	975	1082	1007	962	996	1211	1104	не виявлено
$I_6(i_{ed_{it}})$	1211	1171	1290	1135	1183	1236	1234	1179	не виявлено
$I_7(\emptyset)$	701	727	772	808	754	734	721	775	не виявлено
$I_8(i_{ed_{hc}}, i_{ed_{ca}})$	991	1041	1062	1039	977	1065	1138	1146	не виявлено

Аналіз результатів формує наступні ключові висновки щодо наявності зловмисної поведінки в згенерованих моделях ІП. Зразки ІП 1, 2, 3, 5, 8 були виявлені на основі зібраної множини поведінок, і наявних в цій множині відмінних елементів, завдяки використанню модифікованих ізольованих середовищ. Ця зміна поведінки була пов’язана із використанням методів виявлення емуляції, які по різному реагували на поточне середовище виконання.

Екземпляр під номером 4 демонстрував однакову поведінку виконуючись на усіх підготовлених середовищах виконання, незважаючи на наявність параметрів, які мали б викликати зміну поведінки. Зокрема середовища M_4, M_7, M_8 містили необхідні параметри, які мали б викликати методи обфускації для уникнення виявлення. Одна поведіка залишилась незмінною, що свідчить про те, що ІП навмисно утримувалось від застосування зловмисного коду, що може свідчити про використання стратегії паузи. Про це, зокрема, свідчать певні часові показники виконання.

Екземпляр 6 був неправильно класифікований, як безпечний системою до виконання, оскільки сформована множина поведінок не показала відмінностей. Окрім цього часові показники відповідали очікуваним нормам. Однак, цей аналіз видався оманливим, оскільки екземпляр 6 таки виконав зловмисний код. Ця ситуація виникла через те, що система не передбачила усі досліджені модифікації в рамках згенерованої множини параметрів модифікації H. Екземпляр 6 підкреслює критичну необхідність дослідження нових методів виявлення і їх реалізацію в подібних системах виявлення зловмисного прояву.

Щодо екземпляру 7, який не містив жодної техніки виявлення емуляції, який продемонстрував однакову поведінку в усіх модифікованих середовищах. Ця послідовність у поведінці та отриманих часових метриках, що даний ІП не використовував жодних технік трансформації коду, а значить безпечний до використання, що відповідає його природі.

Для проведення наступного експерименту, який проводився в рамках розробленої системи, було створено 11 екземплярів інфікованих програм згідно визначеної моделі. Кожен екземпляр має унікальний набір методів уникнення виявлення та набір інструкцій. На основі таких екземплярів були згенеровані сімейства, кожне з яких мало певну кількість зразків. В рамках сімейства, методи виявлення емуляції та обфускації, які використовуються в зразках – однакові. Окрім цього, було згенеровано множину програмний об’єктів, що має схожу структуру з моделлю інфікованих програм, але при цьому не використовує жодних методів уникнення від виявлення. Такі об’єкти будемо приймати як неінфіковані програми. Для проведення експерименту, було сформовано 11 наборів моделей, які містили згенеровані моделі інфікованих програм та неінфікованих програм, при цьому кожен набір містив інфіковані програми тільки із одного сімейства.

Для оцінки ефективності роботи розробленої системи, були визначені наступні параметри: true positive (TP) – кількість випадків, коли система правильно ідентифікувала інфіковану програму; false positive (FP) – кількість помилкових спрацювань, коли неінфікована програма була класифікована як інфікована; true negative (TN) – кількість правильних класифікацій неінфікованих програм; false

negative (FN) – кількість інфікованих зразків, які система не змогла розпізнати. На основі цих показників були обчислені наступні характеристики: Accuracy (A) – загальна точність виявлення; Precision (P) – точність передбачень інфікованих програм; Recall (R) – повнота виявлення інфікованих програм; F1-Score ($F1$) – зважене гармонійне середнє між P та R. Результати проведеного експерименту представлено в табл. 4.7.

Таблиця 4.7

Результати експерименту виявлення ІП в розподіленій системі

	TP	FP	TN	FN	A	P	R	$F1$
Експеримент 1	78	1	28	1	98,15	98,73	98,73	98,73
Експеримент 2	93	0	40	1	99,25	100	98,94	99,47
Експеримент 3	56	1	34	4	94,74	98,25	93,33	95,73
Експеримент 1	83	0	30	2	98,26	100	97,65	98,81
Експеримент 5	88	1	29	1	98,32	98,88	98,88	98,88
Експеримент 6	79	0	39	1	99,16	100	98,75	99,37
Експеримент 7	78	0	20	2	98	100	97,5	98,73
Експеримент 8	59	1	39	1	98	98,33	98,33	98,33
Експеримент 9	98	0	27	1	99,21	100	98,99	99,49
Експеримент 10	75	1	19	1	97,92	98,68	98,68	98,68
Експеримент 11	90	0	25	2	98,29	100	97,83	98,9

Проведені експерименти в цілому позитивну динаміку, в більшості випадків загальна точність виявлення коливається від 98 до 99%, про що свідчать і метрики пов’язані кількістю правильно виявлених як інфікованих, так і неінфікованих моделей програм. Хибне визначення неінфікованих програм як інфікованих свідчить скоріше за все про неточну реалізацію деяких особливостей засобу, який використовується для формування поведінки програми. Визначення інфікованих програм, як таких, що не мають зловмисної поведінки говорить про те, що використані модифікації для ізольованих середовищ були підібрані не для усіх випадів правильно, як от в експерименті 3. Цей експеримент також визначає

важливість вивчення нових методів виявлення емуляції та їх імплементацію в засіб, що дасть можливість підвищити точність виявлення. Загалом результати показують позитивні тенденції виявлення зловмисного прояву в ІП, враховуючи їх досліджені техніки уникнення від виявлення.

З метою оцінювання розробленої грід-обчислювальної системи було проведено серію експериментів з іншими проектами, які реалізують подібні принципи побудови розподілених обчислювальних систем. До вибірки було включено такі системи: Alchemi(.NET) [1], XtremWeb-HEP [113], JPPF [51] та Proactive Programming [x]. Для оцінювання ефективності використовувалась обчислювальна задача, близька за характером до задачі, розв'язуваної досліджуваною системою, яка допускає на розбиття на незалежні підзадачі без необхідності очікування подальших етапів роботи. Результати експериментів наведено в табл. 4.8

Таблиця 4.8

Результати експерименту порівняння функціонування грід-обчислювальних систем

	Реалізована система	Alchemi (.NET)	XtremWeb HEP	JPPF	Proactive
Експеримент 1	1055 с.	1091 с.	1114 с.	1086 с.	1099 с.
Експеримент 2	1433 с.	1504 с.	1520 с.	1496 с.	1491 с.
Експеримент 3	3107с.	3167 с.	3322 с.	3222 с.	3204 с.
Експеримент 4	1711 с.	1778 с.	1814 с.	1781 с.	1812 с.

Під час проведення експерименту в усіх обраних системах не застосовувались механізми захисту обчислень, оскільки такі підходи або не передбачені архітектурою відповідних систем, або реалізовані з істотними відмінностями. Зокрема, Alchemi, JPPF та Proactive Programming не використовують механізмів реплікації чи голосування. У свою чергу, в системі XtremWeb-HEP реалізовано голосування, при якому результат вважається достовірним, якщо більшість ОЕ повертає однакове значення. Сам експеримент проводився із використанням наявного набору з чотирьох персональних комп'ютерів.

Таким чином, для сформованих наборів задач розроблена система продемонструвала кращі результати на 3-5% – порівняно із системами Alchemi(.NET), JPPF та Proactive Programming. Зазначені системи, як правило, підтримують декілька алгоритмів розподілу задач, серед яких були й такі, що базуються на принципах, подібних до алгоритму Min-min, який лежить в основі реалізованого підходу. Система XtremWeb-HEP продемонструвала дещо гірші результати, що пов'язано з використанням іншого підходу до обробки задач. У цій системі кожен ОЕ самостійно обирає задачу зі списку готових до виконання, що формується за принципом FIFO.

4.3 Висновки до четвертого розділу

Розроблена грід-обчислювальна система для виявлення зловмисної поведінки в інфікованих програмах включає реалізовані методи уникнення виявлення. Застосовуючи методи синтезу засобів виявлення зловмисного прояву, організації функціонування та оцінювання довіри до обчислювальних елементів, система продемонструвала ефективність на 5-7% при розподілі задач між підключеними обчислювальними елементами у проведених експериментах.

При цьому вона забезпечує пріоритет визначений користувачами системи. Розроблений алгоритм враховує динамічність топології системи, що критично для обчислень із використанням автономних ОЕ. Застосування рольового підходу для модифікованого модуля довіри показало покращення приблизно на 4-6%, що також є критичним фактором, адже такого плану обчислення вимагають залучення додаткових перевірок, для забезпечення коректності виконання поставлених задач.

При проведені експериментів порівняння ефективності виконання обчислювальних завдань із подібними системами показала ефективність в 3-5%, що виражено в швидкості виконання набору задач.

Основні наукові результати четвертого розділу опубліковані в [89-90, 125-129].

ВИСНОВКИ

У результаті виконання дисертаційного дослідження було розв’язано актуальну науково-прикладну задачу покращення ефективності виявлення інфікованих програм із техніками уникнення від виявлення за допомогою застосування розподіленої системи виявлення, що використовує модифіковані ізольовані середовища для аналізу процесу виконання програм, що розгортаються на автономних обчислювальних елементах системи. У роботі отримано наступні наукові та практичні результати.

1. Здійснено аналіз методів та особливостей функціонування грід-обчислювальних систем, встановлено їх переваги та недоліки, зокрема в частині залученні автономних гетерогенних обчислювальних елементів та забезпеченні захисту правильності проведених обчислень. В результаті, було встановлено, що для коректної організації необхідно залучати алгоритми організації обчислень стійкі до динамічності топології системи, а також застосовувати методи захисту обчислень які базуються на використанні моделей довіри.

2. Розроблені і представлені моделі інфікованих програм, які використовують методи виявлення емульованого середовища виконання та обфускації коду для уникнення виявлення. Окрім цього розроблені моделі включають і стратегію виконання на цільовій системі користувача. Сформовані моделі описують умови застосування чи приховування зловмисної поведінки в залежності від особливостей середовища виконання в рамках проаналізованих методів уникнення виявлення.

3. Розроблено модель централізованих грід-обчислювальних систем, в якій враховано вимоги до залучення автономних та гетерогенних обчислювальних елементів для забезпечення виконання задач із перевіркою на коректність в динамічному середовищі виконання. Вона дала змогу залучити під’єднані обчислювальні елементи для аналізу поведінки виконання інфікованих програм, забезпечуючи розподілений процес виявлення зловмисного прояву в інфікованих програмах.

4. Розроблено метод синтезу засобів формування шаблонів поведінки інфікованих програм, який характеризується залученням пісочниці для їх виконання

у наборі створюваних модифікованих ізольованих середовищ за допомогою виконання програмних переривань та базового емулятора із визначеним набором реалізованих низькорівневих інструкцій. Для формування шаблонів поведінки використовуються множини станів емульованих центральних процесорів. Синтезовані засоби формування шаблонів поведінки використовуються як пастки з метою виявлення зловмисної поведінки з урахуванням особливостей методів уникнення від виявлення, які реалізовані зловмисниками.

5. Удосконалено метод організації функціонування грід-обчислювальних систем, в основі якого застосовано жадібний алгоритм для оптимізації навантаження між автономними гетерогенними обчислювальними елементами. Також, використано додаткову чергу активних задач, що забезпечило пріоритетне виконання поставлених задач в розподілених системах із динамічно змінюваною топологією для розподіленого виявлення зловмисної поведінки в інфікованих програмах. Застосований метод дозволив покращити ефективність розподілу завдань на 5-7% загального часу виконання завдань у порівнянні із іншими алгоритмами.

6. Удосконалено метод оцінювання довіри до автономних обчислювальних елементів, в якому використано механізми призначення ролей із застосуванням елементів нечіткої логіки для оптимізації обчислювальних ресурсів шляхом скорочення кількості повторних обчислень у системах, що функціонують в динамічному середовищі. Застосування удосконаленого методу дає змогу оптимізувати використання обчислювальних ресурсів системи на 4-6%.

7. Розроблено централізовану грід-обчислювальну систему виявлення зловмисної поведінки в інфікованих програмах, проведено з нею експериментальні дослідження, які дозволили встановити покращення її характеристик, а саме: ефективність розподілу завдань та оптимальність використання обчислювальних ресурсів, та впроваджено її у виробництво. При проведені експериментів порівняння ефективності виконання обчислювальних завдань із подібними системами показала ефективність в 3-5%, що виражено в швидкості виконання набору задач.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Alchemi. .NET Grid Computing Framework. URL: https://github.com/stemarie/alchemi (дата звернення: 19.03.2025).

2. Alhaidari F., Shaib N. A., Alsafi M., Alharbi H., Alawami M., Aljindan R., Rahman A. U., Zagrouba R. ZeVigilante: Detecting zero-day malware using machine learning and sandboxing analysis techniques. *Computational Intelligence and Neuroscience*. 2022. P 1615528. DOI: https://doi.org/10.1155/2022/1615528

3. Alshuaibi E. A., Hamdi A. M., Hussain F. K. Volunteer computing for fog scalability: a systematic literature review. *Internet of Things*, 2024. Vol. 25, P. 101072. DOI: https://doi.org/10.1016/j.iot.2024.101072

4. Analyze Malware and Phishing in a safe environment. URL: https://any.run/features (дата звернення: 19.03.2025).

5. Anderson D. P. BOINC: a platform for volunteer computing. *Journal of Grid Computing*, 2020. Vol. 18, No. 1, P. 99-122. DOI: https://doi.org/10.1007/s10723-019-09497-9

6. Aslan Ö., Akin E. Malware detection method based on file and registry operations using machine learning. *Sakarya University Journal of Computer and Information Sciences*, 2022. Vol. 5, No. 2, P. 134-146. DOI: https://doi.org/10.35377/ saucis.05.02.1049798

7. AV-Atlas. Total amount of malware and pua. URL: https://portal.av-atlas.org/malware (дата звернення: 19.03.2025).

8. Baek S., Jeon J., Jeong B. and Jeong Y., 2021. Two-stage hybrid malware detection using deep learning. *Human-centric Computing and Information Sciences*, 2021. Vol. 11, No. 27, P. 22967. DOI: https://doi.org/10.22967/HCIS.2021.11.027

9. Bello S. A., Oyedele L. O., Akinade O. O., Bilal M., Delgado J. M. D., Akanbi L. A., Ajayi A. O., Owolabi H. A. Cloud computing in construction industry: use cases, benefits and challenges. *Automation in Construction*, 2021. Vol. 122. P. 103441-103458. DOI: https://doi.org/10.1016/j.autcon.2020.103441

10. Bensaoud A., Kalita J., Bensaoud M. A survey of malware detection using deep learning. *Machine Learning with Applications*, 2024. Vol. 16. P. 100546-100561. DOI: https://doi.org/10.1016/j.mlwa.2024.100546

11. Best Cloud-Based Antivirus Software. URL: https://www.wizcase.com/blog/best-cloud-based-antivirus/ (дата звернення: 19.03.2025).

12. Biondi F., Given-Wilson T., Legay A., Puodzius C., Quilbeuf J. Tutorial: An overview of malware detection and evasion techniques. *Leveraging Applications of Formal Methods, Verification and Validation. Modeling (ISoLA 2018)* : Proceedings of International Conference, 5-9 November, 2018. P. 565-586. DOI: https://doi.org/10.1007/978-3-030-03418-4_34

13. Bitdefender Total Security. URL: https://www.bitdefender.com/en-us/consumer/ total-security (дата звернення: 19.03.2025).

14. Brengel M., Rossow C. YARIX: Scalable YARA-based malware intelligence. *30th USENIX Security Symposium (USENIX Security 21)* : Proceedings of International Conference, 11-13 August, 2021. P. 3541-3558. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/brengel

15. Bussa S., Prabaharan G., Mishra N., Seetha J., Nair V. Singularity AntiVirus: A Novel Approach to Cybersecurity. *Advances in Computing, Communication and Networking (ICAC2N)* : Proceedings of International Conference, 16-17 December, 2024. P. 1636-1640. DOI: https://doi.org/10.1109/ICAC2N63387.2024.10895373

16. Chanajitt R., Pfahringer B., Gomes H. M. Combining static and dynamic analysis to improve machine learning-based malware classification. *8th International Conference on Data Science and Advanced Analytics (DSAA)* : Proceedings of International Conference, 06-09 October, 2021. P. 1-10. DOI: https://doi.org/10.1109/DSAA53316.2021.9564144

17. Chandrashekar C., Krishnadoss P., Kedalu Poornachary V., Ananthakrishnan B., Rangasamy K. HWACOA scheduler: hybrid weighted ant colony optimization algorithm for task scheduling in cloud computing. *Applied Sciences*, 2023. Vol. 13, No. 6, P. 3433-3455. DOI: https://doi.org/10.3390/app13063433

18. Chen S., Lin Z., Zhang Y. SelectiveTaint: Efficient data flow tracking with static binary rewriting. *30th USENIX Security Symposium (USENIX Security 21)* : Proceedings of International Conference, 11-13 August, 2021. P. 1665-1682. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/chen-sanchuan

19. Choudhury R., Luo Z., Nguyen K. A. Malware in motion. *Information Systems Security and Privacy (ICISSP 2022)* : Proceedings of International Conference, 9-11 February, 2022. P. 85. URL: https://research.brighton.ac.uk/en/publications/ malware-in-motion

20. CrowdStrike. 2025 CrowdStrike Global Threat Report. Sunnyvale: CrowdStrike Inc., 2025. 36 с. URL: https://www.crowdstrike.com (дата звернення: 19.03.2025).

21. Dalton A., Thomas D., Cheung P. DSCS: Towards an extensible secure decentralised distributed computation protocol. *2024 IEEE Cyber Security and Resilience (CSR)* : Proceedings of International Conference, 2-4 September, 2018. London, United Kingdom. P. 658-663. DOI: https://doi.org/ 10.1109/CSR61664.2024.10679478

22. Damera V. K., Nagesh A., Nagaratna M. Trust evaluation models for cloud computing. *Environment*, 2020. Vol. 9, No. 2, P. 1964-1971. URL: https://www.ijstr.org

23. Dang Q. V. Enhancing obfuscated malware detection with machine learning techniques. *Future Data and Security Engineering. Big Data, Security and Privacy, Smart City and Industry 4.0 Applications* : Proceedings of International Conference, November, 2022. P. 731-738. DOI: https://doi.org/10.1007/978-981-19-8069-5_54

24. Danieliene R., Bronin S., Milov O., Yevseiev S. Model basis of cybersecurity of socio-cyberphysical systems. *Baltic Journal of Modern Computing*. 2024. Vol. 12, No. 2, P. 125-149. DOI: https://doi.org/10.22364/bjmc.2024.12.2.01

25. Dener M., Ok G., Orman A. Malware detection using memory analysis data in big data environment. *Applied Sciences*, 2022. Vol. 12, No. 17, P. 8604-8624. DOI: https://doi.org/10.3390/app12178604

26. Dey S., Sarma W., Tiwari S. Deep learning applications for real-time cybersecurity threat analysis in distributed cloud systems. *World Journal of Advanced Research and Reviews*. 2023. Vol. 13, No. 3. P. 1044-1058. DOI: https://doi.org/10.30574/wjarr.2023.17.3.0288

27. Divya D., Goel B. M. Issues and challenges of load-balancing algorithms in cloud computing. *Computational Intelligence and Communication Technologies (CCICT)* :

Proceedings of International Conference, 8-9 July, 2022. P. 500-504. DOI: https://doi.org/10.1109/CCiCT56684.2022.00094

28. Dogonyaro, N. M., Victor, W. O., Shafii, A. M., Obada, S. L. Comparative Performance Analysis of Anti-virus Software. *Information and Communication Technology and Applications* : Proceedings of International Conference, 24–27 November, 2020. C. 430-443. Springer International Publishing. DOI: https://doi.org/10.1007/978-3-030-69143-1_33

29. Donta P. K., Murturi I., Casamayor Pujol V., Sedlak B., Dustdar S. Exploring the potential of distributed computing continuum systems. *Computers*, 2023. Vol. 12, No. 10, P. 198-226. DOI: https://doi.org/10.3390/computers12100198

30. Drakulić U., Mujčić E. A comparative performance analysis of various antivirus software. *Advanced Technologies, Systems, and Applications VIII (IAT)* : Proceedings of International Conference, 2023. P 423-430. Springer Nature Switzerland. DOI: https://doi.org/10.1007/978-3-031-43056-5_30

31. Ebad S. A., Darem A. A., Abawajy J. H. Measuring software obfuscation quality – a systematic literature review. *IEEE Access*. 2021. Vol. 9. P. 99024–99038. DOI: https://doi.org/10.1109/ACCESS.2021.3094517

32. ESET SMART SECURITY. URL: https://www.eset.com/fileadmin/ESET/US/ products/home/essp/v14/ESET-Smart-Security-Premium-Product-Overview.pdf (дата звернення: 19.03.2025).

33. Ezeonwu I. J., Musa S. M. Comparative analysis of machine learning classifiers for fileless malware detection. *Green Energy, Computing and Sustainable Technology (GECOST)* : Proceedings of International Conference, 05 April, 2024. P. 1-6. DOI: https://doi.org/10.1109/GECOST60902.2024.10474708

34. Faruki P., Bhan R., Jain V., Bhatia S., El Madhoun N., Pamula R. A survey and evaluation of Android-based malware evasion techniques and detection frameworks. *Information*, 2023. Vol. 14, No. 7. P. 374-411. DOI: https://doi.org/10.3390/info14070374

35. Features of Norton 360. URL: https://support.norton.com/sp/en/aa/home/current/solutions/v70057459 (дата звернення: 19.03.2025).

36. Feng J., Liu L., Pei Q., Li K. Min-max cost optimization for efficient hierarchical federated learning in wireless edge networks. *IEEE Transactions on Parallel and Distributed Systems*, 2021. Vol. 33, No. 11, P. 2687-2700. DOI: https://doi.org/10.1109/TPDS.2021.3131654

37. Fu D., Liu Y. Complex task design based on crowdsourcing using Petri net. *Journal of Computers*, 2021. Vol. 32, No. 6, P. 1-14. DOI: https://doi.org/10.53106/199115992021123206001

38. Galloro N., Polino M., Carminati M., Continella A., Zanero S. A systematical and longitudinal study of evasive behaviors in Windows malware. *Computers & Security*, 2022. Vol. 113, P. 102550. DOI: https://doi.org/10.1016/j.cose.2021.102550

39. Gazzan M., Sheldon F. T. Opportunities for early detection and prediction of ransomware attacks against industrial control systems. *Future Internet*. 2023. Vol. 15, No. 4. P 144-161. DOI: https://doi.org/10.3390/fi15040144

40. Gorment N. Z., Selamat A., Krejcar O. Obfuscated malware detection: impacts on detection methods. *Recent Challenges in Intelligent Information and Database Systems (ACIIDS 2023)* : Proceedings of International Conference, July, 2023. P. 55-66. DOI: https://doi.org/10.1007/978-3-031-42430-4_5

41. Gržinić T., González E. B. Methods for automatic malware analysis and classification: a survey. *International Journal of Information and Computer Security*, 2022. Vol. 17, No. 1-2, P. 179-203. DOI: https://doi.org/10.1504/ IJICS.2022.121297

42. Hai T. H., Van Thieu V., Duong T. T., Nguyen H. H., Huh E. N. A proposed new endpoint detection and response with image-based malware detection system. *IEEE Access*, 2023. No. 11, P. 122859-122875. DOI: https://doi.org/10.1109/ACCESS.2023.3329112

43. Hakobyan R., Jamgharyan T. Polymorphic Malware Analysis Model. *CSIT Conference 2023* : Proceedings of International Conference, 25-30 September, 2023. P. 85-88. DOI: https://doi.org/10.51408/csit2023_17

44. Hassan W. U., Bates A., Marino D. Tactical provenance analysis for endpoint detection and response systems. *2020 IEEE Symposium on Security and Privacy* :

Proceedings of International Conference, San Francisco, CA, USA, 18-21 May, 2020. P. 1172-1189. DOI: https://doi.org/10.1109/SP40000.2020.00096

45. Heimdal Cyber-Security & Threat Intelligence Report 2022-2023. URL: https://heimdalsecurity.com/blog/cybersecurity-threat-report/ (дата звернення: 19.03.2025).

46. Hudson N., Khamfroush H., Baughman M., Lucani D. E., Chard K., Foster I. QoS-aware edge AI placement and scheduling with multiple implementations in FaaS-based edge computing. *Future Generation Computer Systems*, 2024. Vol. 157. P. 250-263. DOI: https://doi.org/10.1016/j.future.2024.03.035

47. Hybrid Analysis. Releases & Updates. URL: https://www.hybrid-analysis.com/ release-notes (дата звернення: 19.03.2025).

48. IBM. 10 industries that use distributed computing. URL: https://www.ibm.com /think/insights/distributed-computing-use-cases (дата звернення: 19.03.2025).

49. Imamverdiyev Y., Baghirov E. Evasion techniques in malware detection: challenges and countermeasures. *Problems of Information Technology*. 2024. Vol. 15, No. 2. P. 9-15. DOI: http://doi.org/10.25045/jpit.v15.i2.02

50. JOESandbox Ultimate Automated Deep Malware Analysis targeting Windows, Android or Mac. URL: https://www.joesecurity.org/joe-sandbox-ultimate (дата звернення: 19.03.2025).

51. JPPF. The open source grid computing solution URL: https://github.com/jppf-grid/JPPF (дата звернення: 19.03.2025).

52. Kang W., Son B., Heo K. TRACER: signature-based static analysis for detecting recurring vulnerabilities. *ACM SIGSAC Conference on Computer and Communications Security* : Proceedings of International Conference, 7 November, 2022. P. 1695-1708. DOI: https://doi.org/10.1145/3548606.3560664

53. Kashtalian A., Lysenko S., Savenko O., Nicheporuk A., Sochor T., Avsiyevych V. Multi-computer malware detection systems with metamorphic functionality. *Radioelectronic and Computer Systems*, Vol. 2024, No. 1. P. 152-175. DOI: https://doi.org/10.32620/reks.2024.1.13

54. Kawakoya Y., Shioji E., Iwamura M., Miyoshi J. API Chaser: Taint-assisted sandbox for evasive malware analysis. *Journal of Information Processing*. 2019. Vol. 27. P. 297-314. DOI: https://doi.org/10.2197/ipsjjip.27.297

55. Kerns Q., Payne B., Abegaz T. Double-extortion ransomware: A technical analysis of Maze ransomware. *Future Technologies Conference (FTC)* : Proceedings of International Conference, 2021. Vol. 3. P. 1-6. DOI: https://doi.org/10.1007/978-3-030-89912-7_7

56. Kharchenko V., Ponochovnyi Y., Ivanchenko O., Fesenko H., Illiashenko O. Combining Markov and semi-Markov modelling for assessing availability and cybersecurity of cloud and IoT systems. *Cryptography*, 2022. Vol. 6, No. 3, P. 44-76. DOI: https://doi.org/10.3390/cryptography6030044

57. Khoroshko V., Khokhlachova Y., Vyshnevska N. Choice of indicators for forecasting cyber protection of computer systems. *Ukrainian Scientific Journal of Information Security*, Vol. 29, No. 1, P.41-47. DOI: https://doi.org/10.18372/2225-5036.29.17551

58. Kim H. M., Lee K. H. IoT malware detection using edge computing and deep learning for cybersecurity in smart factories. *Applied Sciences*, Vol. 12, No. 15, P. 7679-7703. DOI: https://doi.org/10.3390/app12157679

59. Kim M., Cho H., Yi J. H. Large-scale analysis on anti-analysis techniques in real-world malware. *IEEE Access*. 2022. Vol. 10. P. 75802 – 75815. DOI: https://doi.org/10.1109/ACCESS.2022.3190978

60. Kocot B., Czarnul P., Proficz J. Energy-aware scheduling for high-performance computing systems: a survey. *Energies*, 2023. Vol. 16, No. 2, P. 890-917. DOI: https://doi.org/10.3390/en16020890

61. Körber M., Kalysch A., Massonne W., Benenson Z. Usability of antivirus tools in a threat detection scenario. *ICT Systems Security and Privacy Protection* : Proceedings of International Conference, 3 June, 2022. P. 306-322. DOI: https://doi.org/10.1007/978-3-031-06975-8_18

62. Krips K., Snetkov N., Vakarjuk J., Willemson J. Trust assumptions in voting systems. *Computer Security. ESORICS 2023 International Workshops* : Proceedings of International Conference, 2024. DOI: https://doi.org/10.1007/978-3-031-54129-2_18

63. Leon R. S., Kiperberg M., Leon Zabag A. A., Zaidenberg N. J. Hypervisor-assisted dynamic malware analysis. *Cybersecurity*, 2021. Vol. 4. P. 1-14. DOI: https://doi.org/10.1186/s42400-021-00083-9

64. Letychevskyi O., Peschanenko V. Applying algebraic virtual machine to cybersecurity tasks. *9th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT)* : Proceedings of International Conference, 28-30 May, 2022. Hammamet, Tunisia. P. 161-169. DOI: https://doi.org/10.1109/SETIT54465.2022.9875895

65. Liu B. A survey on trust modeling from a Bayesian perspective. *Wireless Personal Communications*, 2020. Vol. 112, P. 1205-1227. DOI: https://doi.org/10.1007/s11277-020-07097-5

66. Lysenko S., Savenko O., Bobrovnikova K., Kryshchuk A. Self-adaptive system for the corporate area network resilience in the presence of botnet cyberattacks. *Communications in Computer and Information Science (CCIS)* : Proceedings of International Conference, 29 May, 2018. Vol. 860. P. 385-401. DOI: https://doi.org/10.1007/978-3-319-92459-5_31

67. Mahmoud R. V., Anagnostopoulos M., Pastrana S., Pedersen J. M. Redefining malware sandboxing: Enhancing analysis through Sysmon and ELK integration. *IEEE Access*. 2024. Vol. 12. P. 68624-68636. DOI: https://doi.org/10.1109/ACCESS.2024.3400167

68. Markowsky G., Savenko O., Lysenko S., Nicheporuk A. The technique for metamorphic viruses' detection based on its obfuscation features analysis. *Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer (ICTERI 2018)* : Proceedings of International Conference, 14-17 May, 2018. Vol. 2104. P. 680-687. URL: https://ceur-ws.org/Vol-2104/paper_231.pdf

69. Markowsky G., Savenko O., Sachenko A. Distributed malware detection system based on decentralized architecture in local area networks. *Advances in Intelligent*

Systems and Computing III (CSIT 2018) : Proceedings of International Conference, 2018. Vol. 871. P. 582-598. DOI: https://doi.org/10.1007/978-3-030-01069-0_42

70. Mitra S., Torri S. A., Mittal S. Survey of malware analysis through control flow graph using machine learning. *22nd International Conference on Trust, Security and Privacy in Computing and Communications* : Proceedings of International Conference, 1-3 November, 2023. P. 1554-1561. DOI: https://doi.org/10.1109/ TrustCom60117.2023.00212

71. Moskalenko V., Kharchenko V., Moskalenko A., Kuzikov B. Resilience and resilient systems of artificial intelligence: taxonomy, models and methods. *Algorithms*, 2023. Vol. 16, No. 3, P. 165-208. DOI: https://doi.org/ 10.3390/a16030165

72. Mukhin V., Kornaga Y., Bondarenko V., Zavgorodnii V., Herasymenko O., Sholokhov O. Mathematical model for heterogeneous databases parameters estimation in distributed systems with dynamic structure. *2020 IEEE Advanced Trends in Information Theory (ATIT)* : Proceedings of International Conference, 25-27 November, 2020. Kyiv, Ukraine. P. 158-161. DOI: https://doi.org/10.1109/ATIT50783.2020.9349331

73. Mukhin V., Kornaga Y., Zavgorodnii V., Bazaka Y., Zavgorodnya A., Mukhin O. Method of data processing system synthesis for heterogeneous distributed databases based on network-centric control. *2023 IEEE Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)* : Proceedings of International Conference, 07-09 September, 2023. Dortmund, Germany. P. 607-612. DOI: https://doi.org/10.1109/IDAACS58523.2023.10348667

74. Mukhin V., Zavgorodnii V., Nikitin. V., Kornaga. Y., Fartushnyi I., Stepanov A. Method of Determining the Required Number of Database Nodes in a Distributed Data Processing System. *3rd International Conference on Advanced Trends in Information Theory (ATIT)* : Proceedings of International Conference, 15-17 December, 2021. Kyiv, Ukraine. P. 88-92. DOI: https://doi.org/ 10.1109/ATIT54053.2021.9678569

75. Olaimat M. N., Maarof M. A., Al-Rimy B. A. S. Ransomware anti-analysis and evasion techniques: A survey and research directions. *International Cyber Resilience Conference (CRC)* : Proceedings of International Conference, 2021. P. 1-6. DOI: https://doi.org/10.1109/CRC50527.2021.9392529

76. Owoh N., Adejoh J., Hosseinzadeh S., Ashawa M., Osamor J., Qureshi A. Malware detection based on API call sequence analysis: a gated recurrent unit–generative adversarial network model approach. *Future Internet*, 2024. Vol. 16, No. 10, P. 369-394. DOI: https://doi.org/10.3390/fi16100369

77. Oz H., Aris A., Levi A., Uluagac A. S. A survey on ransomware: evolution, taxonomy, and defense solutions. *ACM Computing Surveys*. 2022. Vol. 54, No 11s. P. 1-37. DOI: https://doi.org/10.1145/3514229

78. Panda Security Antivirus: Innovation in digital protection. URL: https://www.pandasecurity.com/en/ (дата звернення: 19.03.2025).

79. Pandit A. V., Mondal D. Real-time malware detection on IoT devices using behavior-based analysis and neural networks. *Research Journal of Computer Systems and Engineering*, 2023. Vol. 4, No. 2, P. 117-129. DOI: https://doi.org/ 10.52710/rjcse.82

80. Peng Z., Pirozmand P., Motevalli M., Esmaeili A. Genetic algorithm-based task scheduling in cloud computing using MapReduce framework. *Mathematical Problems in Engineering*, 2022. Vol. 2022, No. 1. DOI: https://doi.org/10.1155/2022/4290382

81. Pérez-Sánchez A., Palacios R. Evaluation of local security event management system vs. standard antivirus software. *Applied Sciences*. 2022. Vol. 12, No 3. P. 1076. DOI: https://doi.org/10.3390/app12031076

82. Pomorova O., Savenko O., Lysenko S., Kryshchuk A., Nicheporuk A. A technique for detection of bots which are using polymorphic code. *Communications in Computer and Information Science* : Proceedings of International Conference, 23-27 June, 2014. Vol. 431. P. 265-276. DOI: https://doi.org/10.1007/978-3-319-07941-7_27

83. Pomorova O., Savenko O., Lysenko S., Nicheporuk A. Metamorphic Viruses Detection Technique based on the Modified Emulators. *Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer (ICTERI 2016)* : Proceedings of International Conference, 21-24 June, 2016. Vol. 1614. P. 375-383. URL: https://ceur-ws.org/Vol-1614/paper_106.pdf

84. ProActive Programming. URL: https://github.com/ow2-proactive/programming (дата звернення: 19.03.2025).

85. Qiu T., Chi J., Zhou X., Ning Z., Atiquzzaman M., Wu D. O. Edge computing in industrial internet of things: architecture, advances and challenges. *IEEE Communications Surveys & Tutorials*, 2020. Vol. 22, No. 4, P. 2462-2488. DOI: https://doi.org/10.1109/COMST.2020.3009103

86. Raeisi-Varzaneh M., Dakkak O., Fazea Y., Kaosar M. G. Advanced cost-aware Max–Min workflow tasks allocation and scheduling in cloud computing systems. *Cluster Computing*, 2024. Vol. 27, No. 9, P. 13407-13419. DOI: https://doi.org/10.1007/s10586-024-04594-1

87. Rao S.M., Jain A. Advances in Malware Analysis and Detection in Cloud Computing Environments: A Review. *International Journal of Safety & Security Engineering*. 2024. Vol. 14, No. 1. P. 225. DOI: https://doi.org/10.18280 /ijsse.140122

88. Rehida P., Markowsky G., Sachenko A., Savenko O. State-based Sandbox Tool for Distributed Malware Detection with Avoid Techniques. *2023 13th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, Athens, Greece, 13–15 October 2023. P. 1-6. DOI: https://doi.org/ https://doi.org/10.1109/DESSERT61349.2023.10416467

89. Rehida P., Savenko O., Kashtalian A., Sachenko A. Malware Detection Tool Based on Emulator State Analysis. *2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Dortmund, Germany, 7–9 September 2023. P. 135-140. DOI: https://doi.org/10.1109/IDAACS58523.2023.10348678

90. Rehida P., Savenko O., Sachenko A., Drozd A., Vizhevski P. A trust model that ensures the correctness of computing in grid computing system. 2024 5th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS), Khmelnytskyi, Ukraine, March 28, 2024. P. 388-401. URL: https://ceur-ws.org/Vol-3675/paper28.pdf

91. Rehida P., Sochor T., Martynyuk V., Tarasova O., Orlenko V. A distributed malware detection model based on sandbox technology. *2023 4th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS)*,

Khmelnytskyi, Ukraine, March 22–24, 2023. P. 475-485. URL: https://ceur-ws.org/Vol-3373/paper32.pdf

92. Rohith C., Kaur. G. Rohith, C. and Kaur, G., 2021, April. A comprehensive study on malware detection and prevention techniques used by anti-virus. *Intelligent Engineering and Management (ICIEM)* : Proceedings of International Conference, 28-30 April, 2021. C. 429-434. DOI: https://doi.org/10.1109/ICIEM51511.2021.9445322

93. Sahay S. K., Sharma A., Rathore H. Evolution of malware and its detection techniques. *Information and Communication Technology for Sustainable Development* : Proceedings of International Conference, 2020. P. 139-150. DOI: https://doi.org/10.1007/978-981-13-7166-0_14

94. Savenko B., Kashtalian A., Lysenko S., Savenko O. Malware detection by distributed systems with partial centralization. *12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)* : Proceedings of International Conference, 07-09 September, 2023. Dortmund, Germany. P. 265-270. DOI: https://doi.org/10.1109/ IDAACS58523.2023.10348773

95. Savenko O., Lysenko S., Nicheporuk A., Savenko B. Approach for the unknown metamorphic virus detection. *9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)* : Proceedings of International Conference, 21-23 September, 2017. Vol. 1. P. 71-76. DOI: https://doi.org/10.1109/IDAACS.2017.8095052

96. Savenko O., Lysenko S., Nicheporuk A., Savenko B. Metamorphic Viruses' Detection Technique Based on the Equivalent Functional Block Search. *Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer (ICTERI 2016)* : Proceedings of International Conference, 15-18 May, 2017. Vol. 1844. P. 555-568. URL: https://ceur-ws.org/Vol-1844/10000555.pdf

97. Savenko O., Nicheporuk A., Hurman I., Lysenko S. Dynamic signature-based malware detection technique based on API call tracing. *Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer* : Proceedings of

International Conference, 12-15 June, 2019. Vol. 2393. P. 633-643. URL: https://ceur-ws.org/Vol-2393/paper_278.pdf

98. Savenko O., Sachenko A., Lysenko S., Markowsky G., Vasylkiv N. Botnet detection approach based on the distributed systems. *International Journal of Computing*, 2019. Vol. 19, No. 2, P. 190-198. DOI: https://doi.org/10.47839/ijc.19.2.1761

99. Sayadi H., Gao Y., Mohammadi Makrani H., Lin J., Costa P. C., Rafatirad S., Homayoun H. Towards accurate run-time hardware-assisted stealthy malware detection: a lightweight, yet effective time series CNN-based approach. *Cryptography*, 2021. Vol. 5, No. 4, P. 28-52. DOI: https://doi.org/10.3390/ cryptography5040028

100. Serpanos D., Michalopoulos P., Xenos G., Ieronymakis V. Sisyfos: A modular and extendable open malware analysis platform. *Applied Sciences*. 2021. Vol. 11, No. 7. P. 2980. DOI: https://doi.org/10.3390/app11072980

101. Shafin S. S., Karmakar G., Mareels I. Obfuscated memory malware detection in resource-constrained IoT devices for smart city applications. *Sensors*. 2023. Vol. 23, No. 11. P. 5348-5365. DOI: https://doi.org/10.3390/s23115348

102. Shukla D. K., Kumar D., Kushwaha D. S. An efficient tasks scheduling algorithm for batch processing heterogeneous cloud environment. *International Journal of Advanced Intelligence Paradigms*, 2022. Vol. 23, No. 1-2, P. 203-216. DOI: https://doi.org/10.1504/IJAIP.2022.125242

103. Sihwail R., Omar K., Arifin K. A. Z. An effective memory analysis for malware detection and classification. *Computers, Materials & Continua*. 2021. Vol. 67, No. 2. P. 2301-2320. DOI: https://doi.org/10.32604/cmc.2021.014510

104. Singh A., Chatterjee K. Trust management in online computing environment: a complete review. *Journal of Ambient Intelligence and Humanized Computing*, 2024. Vol. 15, No. 1, P. 491-545. DOI: https://doi.org/10.1007/s12652-023-04676-9

105. SonicWall. 2024 SonicWall Mid-Year Cyber Threat Report. Milpitas: SonicWall Inc., 2024. 18 с. URL: https://www.sonicwall.com (дата звернення: 19.03.2025).

106. Sophos 2024 Threat Report: Cybercrime on Main Street. Ransomware remains the biggest existential cyber threat to small business, but others are growing. URL:

https://assets.sophos.com/X24WTUEQ/at/wwf5phjtj9bjvmpqqsbfxc/sophos-2024-threat-report.pdf (дата звернення: 19.03.2025).

107. Steltner B., Papa M. A., Eggenstein H. B., Prix R., Bensch M., Allen B., Machenschalk B. Deep Einstein@Home all-sky search for continuous gravitational waves in LIGO O3 public data. *The Astrophysical Journal*, 2023. Vol. 952, No. 1, P. 55-64. DOI: https://doi.org/10.3847/1538-4357/acdad4

108. Sun Q., Abuhamad M., Abdukhamidov E., Chan-Tin E., Abuhmed T. MLxPack: Investigating the effects of packers on ML-based malware detection systems using static and dynamic traits. *1st Workshop on Cybersecurity and Social Sciences* : Proceedings of International Conference, 2022. P 11-18. DOI: https://doi.org/10.1145/3494108.3522768

109. Symantec Content Analysis: Multilayered Thread Protection. URL: https://www.security.com/product-insights/symantec-content-analysis-multi-layered-threat-protection (дата звернення: 19.03.2025).

110. Uchenna C.C., Jamil N., Ismail R., Yan L. K., Mohamed M. A. Malware threat analysis techniques and approaches for IoT applications: a review. *Bulletin of Electrical Engineering and Informatics*. 2021. Vol. 10, No. 3. P. 1558-1571. DOI: https://doi.org/10.11591/eei.v10i3.2423

111. Understanding your protection features. URL: https://download.mcafee.com/pro ducts/webhelp/4/1033/GUID-8A7F4357-8BEA-431B-8610-E157C499FFF5.html (дата звернення: 19.03.2025).

112. Vyšniūnas T., Čeponis D., Goranin N., Čenys A. Risk-based system-call sequence grouping method for malware intrusion detection. *Electronics*, 2024. Vol. 13, No. 1, P. 206-222. DOI: https://doi.org/10.3390/electronics13010206

113. Wood B., Watling B., Winn Z., Messiha D., Mahmoud Q. H., Azim A. Remote method delegation: a platform for grid computing. *Journal of Grid Computing*, 2020. Vol. 18, No. 4, P. 711-725. DOI: https://doi.org/10.1007/s10723-020-09525-z

114. Xie N., Qin Z., Di X. GA-StackingMD: Android malware detection method based on genetic algorithm optimized stacking. *Applied Sciences*. 2023. Vol. 13, No. 4, P. 2629-2644. DOI: https://doi.org/10.3390/app13042629

115. XtremWeb-HEP URL: https://github.com/iExecBlockchainComputing/xtremweb-hep (дата звернення: 19.03.2025).

116. Yin L., Zhou J., Sun J. A stochastic algorithm for scheduling bag-of-tasks applications on hybrid clouds under task duration variations. *Journal of Systems and Software*, 2022. Vol. 184, P. 111123. DOI: https://doi.org/10.1016/j.jss.2021.111123

117. Yousuf M. I., Anwer I., Riasat A., Zia K. T., Kim S. Windows malware detection based on static analysis with multiple features. *PeerJ Computer Science*, 2023. Vol. 9. DOI: https://doi.org/10.7717/peerj-cs.1319

118. Yviquel H., Pereira M., Francesquini E., Valarini G., Leite G., Rosso P., Ceccato R., Cusihualpa C., Dias V., Rigo S., Souza A. The OpenMP cluster programming model. *51st International Conference on Parallel Processing* : Proceedings of International Conference, 29 August, 2022. P. 1-11. DOI: https://doi.org/ 10.1145/3547276.3548444

119. Zainodin M. E., Zakaria Z., Hassan R., Abdullah Z. Entropy-based method for malicious file detection. *JOIV: International Journal on Informatics Visualization*, 2022. Vol. 6, No. 4. P. 856-861. DOI: https://doi.org/10.30630/joiv.6.4.1265

120. Zaw H. T., Aung T., Maw A. H., Mon M. T. Comparative analysis of YARA and VirusTotal integrations with Wazuh for enhanced malware detection. *5th International Conference on Advanced Information Technologies (ICAIT)* : Proceedings of International Conference, 07 November, 2024. P 1-6. DOI: https://doi.org/10.1109/ICAIT65209.2024.10754931

121. Zeyu H., Geming X., Zhaohang W., Sen Y. Survey on edge computing security *International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)* : Proceedings of International Conference, 12-14 June, 2020. Fuzhou, China. P. 96-105. DOI: https://doi.org/10.1109/ICBAIE49996.2020.00027

122. Zhang S., Wu J., Zhang M., Yang W. Dynamic malware analysis based on API sequence semantic fusion. *Applied Sciences*. 2023. Vol. 13, No. 11. P. 6526. DOI: https://doi.org/10.3390/app13116526

123. Марченко Р.М., Коваленко А. А., Знайдюк В. Г. Аналіз методів виявлення аномального трафіку в мережах IoT. *Системи управління, навігації та зв'язку*, 2024. Том 1, № 75, С. 133-136. DOI: https://doi.org/10.26906/SUNZ.2024.1.133

124. Регіда П. Г. Засіб розподіленого виявлення зловмисного програмного забезпечення із використанням технології емулювання. *XX ювілейна міжнародна науково-практична конференція «Математичне та програмне забезпечення інтелектуальних систем» (МПЗІС-2022)*, Дніпро, Україна, листопад 23-25, 2022. С. 170-171. URL: http://mpzis.dnu.dp.ua/wp-content/uploads/2022/11/Program_MPZIS_2022.pdf

125. Регіда П. Г., Савенко О. С., Нічепорук А. О., Дрозд А. І. Авторське свідоцтво №134190. Україна. Комп'ютерна програма «Програмне забезпечення організації розподіленої системи виконання програмного коду в ізольованих середовищах» («РСВКІС»). Дата реєстрації: 10 березня 2025 р.

126. Регіда П., Бармак. О., Каштальян А., Манзюк Е. Концепція застосування розподілених систем для аналізу поліморфних вірусів. *Вісник Хмельницького національного університету. Серія: Технічні науки*, 2024. Том. 331, № 1, С. 38-43. DOI: https://doi.org/10.31891/2307-5732-2024-331-4

127. Регіда П., Капустян М. Лигун О. Динамічне емулювання як засіб виявлення поліморфних вірусів із маскувальними техніками. *The 13th International Scientific Conference «ITSec»*, м. Львів, Україна Травень 9-11, 2024. С. 179-180.

128. Регіда П.Г. Метод організації розподіленої системи виявлення інфікованих програм в ізольованих середовищах. *Вісник Хмельницького національного університету. Технічні науки.* 2025. Т. 347. № 1. С. 554-560. DOI: https://doi.org/10.31891/2307-5732-2025-347-76

129. Регіда П.Г. Поведінкова модель довіри в грід обчислювальній системі на основі нечіткої логіки. *Вимірювальна та обчислювальна техніка в технологічних процесах.* 2025. №1 (2025). С. 287-293. DOI: https://doi.org/10.31891/2219-9365-2025-81-35

130. Регіда П.Г., Савенко О.С. Метод виявлення зловмисної активності в інфікованих програмах. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2024. №1. С. 178-186. DOI: https://doi.org/10.32782/IT/2024-4-21

131. Юдін О., Сидоренко В., Гнатюк С., Верховець О. Модель розрахунку кількісного критерію оцінювання захищеності інформаційно-телекомунікаційних систем критичної інфраструктури держави. *Сучасні інформаційні системи*. 2021. Т. 5, № 4, С. 109-115. DOI: https://doi.org/10.20998/2522-9052.2021.4.15

ДОДАТКИ

ДОДАТОК А.
СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Наукові праці, в яких опубліковані основні наукові результати дисертації

1. Регіда П.Г., Бармак О.В., Каштальян А.С., Манзюк Е.А. Концепція застосування розподілених систем для аналізу поліморфних вірусів. *Вісник Хмельницького національного університету. Технічні науки.* 2024. Т. 331. №1. С. 38-43. DOI: https://doi.org/10.31891/2307-5732-2024-331-4

2. Регіда П.Г., Савенко О.С. Метод виявлення зловмисної активності в інфікованих програмах. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2024. №1. С. 178-186. DOI: https://doi.org/10.32782/IT/2024-4-21

3. Регіда П.Г. Метод організації розподіленої системи виявлення інфікованих програм в ізольованих середовищах. *Вісник Хмельницького національного університету. Технічні науки.* 2025. Т. 347. № 1. С. 554-560. DOI: https://doi.org/10.31891/2307-5732-2025-347-76

4. Регіда П.Г. Поведінкова модель довіри в грід обчислювальній системі на основі нечіткої логіки. *Вимірювальна та обчислювальна техніка в технологічних процесах.* 2025. №1 (2025). С. 287-293. DOI: https://doi.org/10.31891/2219-9365-2025-81-35

Праці, які засвідчують апробацію матеріалів дисертації

5. Регіда П. Г. Засіб розподіленого виявлення зловмисного програмного забезпечення із використанням технології емулювання. *XX ювілейна міжнародна науково-практична конференція «Математичне та програмне забезпечення інтелектуальних систем» (МПЗІС-2022)*, Дніпро, Україна, листопад 23-25, 2022. С. 170-171.

6. Rehida P., Sochor T., Martynyuk V., Tarasova O., Orlenko V. A distributed malware detection model based on sandbox technology. *2023 4th International Workshop*

on Intelligent Information Technologies & Systems of Information Security (IntelITSIS), Khmelnytskyi, Ukraine, March 22–24, 2023. P. 475-485. *(Scopus)* URL: https://ceur-ws.org/Vol-3373/paper32.pdf

7. Rehida P., Savenko O., Kashtalian A., Sachenko A. Malware Detection Tool Based on Emulator State Analysis. *2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Dortmund, Germany, 7–9 September 2023. P. 135-140. *(Scopus)* DOI: https://doi.org/10.1109/IDAACS58523.2023.10348678

8. Rehida P., Markowsky G., Sachenko A., Savenko O. State-based Sandbox Tool for Distributed Malware Detection with Avoid Techniques. *2023 13th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, Athens, Greece, 13–15 October 2023. P. 1-6. *(Scopus)* DOI: https://doi.org/10.1109/DESSERT61349.2023.10416467

9. Rehida P., Savenko O., Sachenko A., Drozd A., Vizhevski P. A trust model that ensures the correctness of computing in grid computing system. 2024 5th International Workshop on Intelligent Information Technologies & Systems of Information Security (IntelITSIS), Khmelnytskyi, Ukraine, March 28, 2024. P. 388-401. *(Scopus)* URL: https://ceur-ws.org/Vol-3675/paper28.pdf

10. Регіда П., Капустян М. Лигун О. Динамічне емулювання як засіб виявлення поліморфних вірусів із маскувальними техніками. *The 13th International Scientific Conference «ITSec»*, м. Львів, Україна Травень 9-11, 2024. С. 179-180.

Публікації, які додатково відображають наукові результати дисертації

11. Регіда П. Г., Савенко О. С., Нічепорук А. О., Дрозд А. І. Авторське свідоцтво №134190. Україна. Комп’ютерна програма «Програмне забезпечення організації розподіленої системи виконання програмного коду в ізольованих середовищах» («РСВКІС»). Дата реєстрації: 10 березня 2025 р.

ДОДАТОК Б.
АКТИ ВПРОВАДЖЕНЬ

«Затверджую»
Генеральний директор ПП «Нолт Технолоджис»
Олександр МЕЛЬНИЧЕНКО
21 квітня 2025 р.

АКТ
про впровадження результатів дисертаційної роботи
старшого викладача кафедри комп'ютерної інженерії та інформаційних систем
Хмельницького національного університету Регіди Павла Геннадійовича
«Методи та засоби організації розподілених систем виявлення інфікованих виконуваних програм, стійких до емуляції в середовищі виконання»

Результати дисертаційної роботи аспіранта кафедри комп'ютерної інженерії та інформаційних систем Хмельницького національного університету Регіди П.Г. впроваджені на ПП «Нолт Технолоджис».

При розробці централізованої грід-обчислювальної системи виявлення інфікованих виконуваних програм із використанням автономних обчислювальних елементів були отримані на ПП «Нолт Технолоджис» такі результати, які одержані Регідою П.Г. особисто:

1) модель централізованої грід-обчислювальної системи, що залучає обчислювальні ресурси автономних та гетерогенних елементів, для забезпечення виконання задач із перевіркою на коректність в динамічному середовищі виконання, що дозволило використати акумульовані ресурси для дослідження та аналізу поведінки виконання інфікованих програм в модифікованих ізольованих середовищах виконання, тим самим реалізуючи метод розподіленого виявлення зловмисного прояву в інфікованих програмах

2) метод планування обчислень для організації процесу виконання задач в грід-обчислювальній системі, який використовує додаткову чергу виконання і реалізацію жадібного алгоритму, що імплементується в централізовану модель, мета якої забезпечити розподілений підхід до виявлення інфікованих програм.

Виконані дослідження включають методи та централізовану розподілену систему покращення ефективності функціонування розподілених обчислювальних систем для виявлення зловмисного коду в інфікованих програмах за рахунок синтезу засобів аналізу їх виконання в модифікованих ізольованих середовищах.

Отримані результати покращують ефективність планування розподілу задач між автономними та обчислювальними елементами за рахунок реалізованого методу у порівнянні із його аналогами приблизно на 5-7%.

Цей акт не є підставою для фінансових розрахунків.

Генеральний директор Олександр МЕЛЬНИЧЕНКО

«Затверджую»
Директор ТОВ «ІТТ»
Сімогук В.С.
«14» квітня 2025 р.

АКТ

про впровадження результатів дисертаційної роботи
аспіранта кафедри комп'ютерної інженерії та інформаційних систем
Хмельницького національного університету Регіди Павла Геннадійовича
«Методи та засоби організації розподілених систем виявлення інфікованих виконуваних програм, стійких до емуляції в середовищі виконання»

Результати дисертаційної роботи Регіди П. Г. аспіранта кафедри комп'ютерної інженерії та інформаційних систем Хмельницького національного університету впроваджені на ТОВ «ІТТ».

При розробленні грід-обчислювальної системи виявлення інфікованих програм, що використовують методи уникнення виявлення були отримані на ТОВ «ІТТ» такі результати, які одержані Регідою П.Г. особисто:

1) модель централізованих грід-обчислювальних систем, в якій враховано вимоги до залучення автономних та гетерогенних обчислювальних елементів для забезпечення виконання задач із перевіркою на коректність в динамічному середовищі виконання, і яка дає змогу залучити під'єднані обчислювальні елементи для аналізу поведінки виконання інфікованих програм, забезпечуючи розподілений процес виявлення зловмисного прояву в інфікованих програмах;

2) метод синтезу засобів формування шаблонів поведінки інфікованих програм, який на відміну від відомих відрізняється залученням пісочниці для їх виконання у наборі створюваних модифікованих ізольованих середовищах за допомогою виконання програмних переривань та базового емулятора із визначеним набором реалізованих низькорівневих інструкцій, що дає змогу отримувати з них шаблони поведінки на множинах станів емульованих центральних процесорів з метою виявлення зловмисної поведінки з урахуванням особливостей методів уникнення від виявлення, які реалізовані зловмисниками;

3) метод оцінювання довіри автономних обчислювальних елементів, який на відміну від відомих використовує механізми призначення ролей із використанням елементів нечіткої логіки, що дає змогу оптимізувати використання обчислювальних ресурсів шляхом скорочення кількості повторних обчислень у системах, що функціонують в динамічному середовищі.

Отриманні результати дозволили покращити ефективність використання обчислювальних ресурсів в грід-обчислювальних системах, що залучають автономні обчислювальні елементи для виявлення зловмисної поведінки в інфікованих програмах на 4-6%.

Цей акт не є підставою для фінансових розрахунків.

Заступник директора Іванов О.В.

Системний адміністратор Веремеєнко В.А.

«Затверджую»
Проректор з науково-педагогічної роботи
_____Віктор ЛОПАТОВСЬКИЙ
«16» квітня 2025 р.

АКТ

про впровадження в навчальний процес Хмельницького національного університету результатів дисертаційної роботи аспіранта кафедри комп'ютерної інженерії та інформаційних систем Регіди Павла Геннадійович «Методи та засоби організації розподілених систем виявлення інфікованих виконуваних програм, стійких до емуляції в середовищі виконання»

Ми, комісія в складі: декана факультету інформаційних технологій, професора кафедри комп'ютерної інженерії та інформаційних систем, д.т.н., професора Говорущенко Т. О. (голова комісії), доцента кафедри комп'ютерної інженерії та інформаційних систем, к.т.н., доцента Нічепорука А. О., доцента кафедри комп'ютерної інженерії та інформаційних систем, к.т.н., доцента Капустян М. В. склала акт про те, що результати дисертаційної роботи Регіди П. Г. впроваджені та використовуються в освітньому процесі Хмельницького національного університету при викладанні дисциплін на кафедрі комп'ютерної інженерії та інформаційних систем для спеціальності 123 Комп'ютерна інженерія, зокрема в курсах «Теорія і проєктування комп'ютерних та кіберфізичних систем і мереж», «Безпека та захист комп'ютерних систем», «Комп'ютерні мережі, системне адміністрування та кібербезпека»

При викладанні цих дисциплін викладачами кафедр використовувалися наступні матеріали досліджень, отримані Регідою П.Г. особисто

1) модель централізованих грід-обчислювальних систем, в якій враховано вимоги до залучення автономних та гетерогенних обчислювальних елементів для забезпечення виконання задач із перевіркою на коректність в динамічному середовищі виконання, і яка дає змогу залучити під'єднані обчислювальні елементи для аналізу поведінки виконання інфікованих програм, забезпечуючи розподілений процес виявлення зловмисного прояву в інфікованих програмах;

2)метод синтезу засобів формування шаблонів поведінки інфікованих програм, який на відміну від відомих відрізняється залученням пісочниці для їх виконання у наборі створюваних модифікованих ізольованих середовищах за допомогою виконання програмних переривань та базового емулятора із визначеним набором реалізованих низькорівневих інструкцій, що дає змогу отримувати з них шаблони поведінки на множинах станів емульованих

центральних процесорів з метою виявлення зловмисної поведінки з урахуванням особливостей методів уникнення від виявлення, які реалізовані зловмисниками;

3)метод організації функціонування грід-обчислювальних систем, який на відміну від відомих залучає жадібний алгоритм для оптимізації навантаження між автономними гетерогенними обчислювальними елементами та використовує додаткову чергу активних задач, що дає змогу забезпечити збалансоване виконання поставлених задач в розподілених системах із динамічно змінюваною топологією для розподіленого виявлення зловмисної поведінки в інфікованих програмах;

Отримані матеріали досліджень дозволили розробити лабораторні практикуми з використанням архітектури централізованої грід-обчислювальних систем, методу синтезу засобів формування шаблону поведінки інфікованих програм.

Говорущенко Т. О.
Нічепорук А. О.
Капустян М. В.

ДОДАТОК В.

ЛІСТИНГ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

```
private static final ConcurrentMap<String, CEConnection> activeClients = new
ConcurrentHashMap<>();
private boolean running = true;

public static ConcurrentMap<String, CEConnection> getActiveClients() {
    return activeClients;
}

public void Start() {
    try (ServerSocket serverSocket = new ServerSocket(PORT)) {
        System.out.println("Listening on port " + PORT);

        while (running) {
            Socket clientSocket = serverSocket.accept();
            Thread.startVirtualThread(() -> handleComputingElement(clientSocket));
            Thread.startVirtualThread(() -> {
                activeClients.values().forEach(client -> {
                    if (client.isActive()) {
                        try {
                            runCalc(client);
                        } catch (IOException e) {
                            System.err.println("Error in runCalc for client: " + client.getClientId());
                        }
                    }
                });
                activeClients.values().forEach(client -> {
                    if (client.isActive()) {
                        try {
                            evaluateCompPower(client);
                        } catch (IOException e) {
                            System.err.println("Error in runCalc for client: " + client.getClientId());
                        }
                    }
                });
            });
        }
    } catch (IOException e) {
        System.err.println("Server error: " + e.getMessage());
    }
}

public void Stop() {
    running = false;
    activeClients.values().forEach(client -> {
        try {
            client.close();
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
    activeClients.clear();
    System.err.println("Server stopped");
}

private static void runCalc(CEConnection client) throws IOException {
    ObjectMapper mapper = new ObjectMapper();
    File jsonFile = new File("instructions.json");
    try {
        String jsonContent = mapper.writeValueAsString(mapper.readTree(jsonFile));
        client.sendMessage(jsonContent);
        String responseString = client.receiveMessage();
        if (responseString != null) {
            JsonNode response = mapper.readTree(responseString);
            String state = response.get("state").asText();
            JsonNode hardwareInfo = response.get("hardwareInfo");
            System.out.println("Received state: " + state);
            System.out.println("Received hardware info: " + hardwareInfo.toString());
        } else {
            System.err.println("Client disconnected without sending a response.");
        }
    } catch (IOException e) {
        System.err.println("Error in runCalc: " + e.getMessage());
    }
}

private static void evaluateCompPower() {
    ObjectMapper mapper = new ObjectMapper();
    File jsonFile = new File("test.json");
    activeClients.values().forEach(client -> {
        if (client.isActive()) {
            try {
                String jsonContent = mapper.writeValueAsString(mapper.readTree(jsonFile));
                client.sendMessage(jsonContent);
                String response = client.receiveMessage();
                if (response != null) {
                    try {
                        int computationTime = Integer.parseInt(response);
                        System.out.println("Computation time from client " + client.getClientId() + ": " +
computationTime + " ms");
                    } catch (NumberFormatException e) {
                        System.err.println("Invalid computation time received from client: " + response);
                    }
                } else {
                    System.err.println("Client " + client.getClientId() + " disconnected without sending
computation time.");
                }
            } catch (IOException e) {
                System.err.println("Error in evaluateCompPower: " + e.getMessage());
```

```
            }
        }
    });


public class CEConnection {
    private final Socket socket;
    private final PrintWriter writer;
    private final BufferedReader reader;
    private final AtomicBoolean active;

    private final String clientId;

    public CEConnection(Socket socket) throws IOException {
        this.socket = socket;
        this.writer = new PrintWriter(socket.getOutputStream(), true);
        this.reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        this.active = new AtomicBoolean(true);
        this.clientId = "Client-" + socket.getInetAddress() + ":" + socket.getPort();
    }

    public String getClientId() {
        return clientId;
    }

    public boolean isActive() {
        return active.get();
    }

    public void sendMessage(String message) {
        if (isActive() && !socket.isClosed()) {
            writer.println(message);
        } else {
            System.err.println("Cannot send message. Client is inactive or socket is closed.");
        }
    }

    public String receiveMessage() {
        try {
            String message = reader.readLine();
            if (message == null) {
                active.set(false);
            }
            return message;
        } catch (IOException e) {
            active.set(false);
            return null;
        }
    }

    public boolean performHeartbeat () {
        try {
            writer.println("HEARTHBEAT");
```

```
            String response = reader.readLine();
            return "ALIVE".equalsIgnoreCase(response);
        } catch (IOException e) {
            active.set(false);
            return false;
        }
    }

    public void close() throws IOException {
        active.set(false);
        socket.close();
    }

public class DatabaseService {
    private static final SessionFactory sessionFactory = new Configuration()
            .configure("hibernate.cfg.xml")
            .addAnnotatedClass(HardwareInfo.class)
            .buildSessionFactory();

    public static void saveHardwareInfo(HardwareInfo hardwareInfo) {
        try (Session session = sessionFactory.openSession()) {
            session.beginTransaction();

            session.persist(hardwareInfo);

            session.getTransaction().commit();
            System.out.println("Hardware info saved to PostgreSQL!");
        } catch (Exception e) {
            System.err.println("Error saving hardware info: " + e.getMessage());
        }
    }
}
Map<String, Object> hardwareInfo = collectHardwareInfo(hardware);

HardwareInfo hardwareEntity = new HardwareInfo();
Map<String, Object> cpu = (Map<String, Object>) hardwareInfo.get("cpu");
Map<String, Object> memory = (Map<String, Object>) hardwareInfo.get("memory");

hardwareEntity.setCpuName((String) cpu.get("name"));
hardwareEntity.setArchitecture((String) cpu.get("architecture"));
hardwareEntity.setCoreCount((int) cpu.get("coreCount"));
hardwareEntity.setThreadCount((int) cpu.get("threadCount"));
hardwareEntity.setMemoryTotal((String) memory.get("total"));
hardwareEntity.setMemoryAvailable((String) memory.get("available"));
DatabaseService.saveHardwareInfo(hardwareEntity);

public class TaskCalc {
    private final String id;
    private final double complexity;

    public TaskCalc(String id, double complexity) {
        this.id = id;
```

```
        this.complexity = complexity;
    }

    public String getId() {
        return id;
    }

    public double getComplexity() {
        return complexity;
    }

    @Override
    public String toString() {
        return "Task{" +
                "id='" + id + '\'' +
                ", complexity=" + complexity +
                '}';
    }
}
Map<TaskCalc, ComputationalElement> taskAssignment = new HashMap<>();

public void distributeTasks() {
    List<TaskCalc> tasks = new ArrayList<>();
    modifiedTasks.drainTo(tasks);
    if (tasks.isEmpty()) {
        System.out.println("No tasks to schedule.");
        return;
    }
    Map<CEConnection, Double> readyTime = new HashMap<>();
    for (CEConnection client : activeClients.values()) {
        readyTime.put(client, 0.0);
    }
    while (!tasks.isEmpty()) {
        TaskCalc  bestTask   = null;
        CEConnection bestClient = null;
        double    bestFinish = Double.POSITIVE_INFINITY;
        for (TaskCalc task : tasks) {
            for (CEConnection client : activeClients.values()) {
                double perf    = estimatePerformance(client);
                double start   = readyTime.get(client);
                double runtime = task.getComplexity() / perf;
                double finish  = start + runtime;
                if (finish < bestFinish) {
                    bestFinish   = finish;
                    bestTask     = task;
                    bestClient   = client;
                }
            }
        }

        if (bestTask == null) break;
        String payload = toJson(bestTask);
```

```
            bestClient.sendMessage(payload);
            readyTime.put(bestClient, bestFinish);
                bestTask.getId(), bestClient.getClientId(), bestFinish);
            tasks.remove(bestTask);
        }
    }
    private double estimatePerformance(CEConnection client) {
        HardwareInfo hw = client.getHardwareInfo();
        return hw.getCoreCount() * 1000.0;
    }
    private String toJson(TaskCalc task) {
        try {
            return new ObjectMapper().writeValueAsString(task);
        } catch (JsonProcessingException e) {
            throw new RuntimeException(e);
        }
    }

    public class SecureTaskManager {
        private final ExecutorService executorService;
        private final BlockingQueue<Task> taskQueue;
        private final Map<String, TaskStatus> taskStatusMap;
        private final Map<String, String> clientTokens;
        private final SecretKey secretKey;
        private final AuditLogger auditLogger;
        private final MonitoringService monitoringService;

        public SecureTaskManager(int threadPoolSize, int maxRetries, String secret) throws
    NoSuchAlgorithmException {
            this.executorService = Executors.newFixedThreadPool(threadPoolSize);
            this.taskQueue = new LinkedBlockingQueue<>();
            this.taskStatusMap = new ConcurrentHashMap<>();
            this.clientTokens = new ConcurrentHashMap<>();
            this.secretKey = generateKey(secret);
            this.auditLogger = new AuditLogger();
            this.monitoringService = new MonitoringService();
        }


        private SecretKey generateKey(String secret) {
            return new SecretKeySpec(secret.getBytes(), "HmacSHA256");
        }


        public boolean authenticateClient(String clientId, String token) {
            String expectedToken = clientTokens.get(clientId);
            return expectedToken != null && expectedToken.equals(token);
        }


        public void registerClient(String clientId, String token) {
            clientTokens.put(clientId, token);
```

```
    }


    public boolean verifyIntegrity(String data, String receivedHmac) throws NoSuchAlgorithmException,
InvalidKeyException {
        Mac mac = Mac.getInstance("HmacSHA256");
        mac.init(secretKey);
        byte[] computedHmac = mac.doFinal(data.getBytes());
        String computedHmacString = Base64.getEncoder().encodeToString(computedHmac);
        return computedHmacString.equals(receivedHmac);
    }

    public void addTask(Task task) {
        taskQueue.offer(task);
        taskStatusMap.put(task.getId(), TaskStatus.NEW);
        auditLogger.log("Task added: " + task.getId());
    }

    public void start() {
        while (!Thread.currentThread().isInterrupted()) {
            try {
                Task task = taskQueue.take();
                processTask(task);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.err.println("Task processing interrupted.");
            }
        }
    }

    private void processTask(Task task) {
        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            try {
                taskStatusMap.put(task.getId(), TaskStatus.IN_PROGRESS);
                monitoringService.recordTaskStart(task.getId());
                task.execute();
                taskStatusMap.put(task.getId(), TaskStatus.COMPLETED);
                auditLogger.log("Task completed: " + task.getId());
            } catch (Exception e) {
                System.err.println("Task " + task.getId() + " failed: " + e.getMessage());
                auditLogger.log("Task failed: " + task.getId() + " - " + e.getMessage());
                handleFailure(task);
            }
        }, executorService);

        future.orTimeout(task.getTimeoutMillis(), TimeUnit.MILLISECONDS)
                .exceptionally(ex -> {
                    System.err.println("Task " + task.getId() + " timed out.");
                    auditLogger.log("Task timed out: " + task.getId());
                    handleFailure(task);
                    return null;
                });
```

```
    }

    private void handleFailure(Task task) {
        int retries = task.getRetryCount();
        if (retries < 3) {
            System.out.println("Retrying task " + task.getId() + " (attempt " + (retries + 1) + ")...");
            auditLogger.log("Retrying task: " + task.getId() + " (attempt " + (retries + 1) + ")");
            task.incrementRetryCount();
            taskQueue.offer(task);
        } else {
            System.err.println("Task " + task.getId() + " failed after max retries.");
            auditLogger.log("Task failed after max retries: " + task.getId());
            taskStatusMap.put(task.getId(), TaskStatus.FAILED);
        }
    }

    public Map<String, TaskStatus> getTaskStatuses() {
        return Collections.unmodifiableMap(taskStatusMap);
    }

    public void shutdown() {
        executorService.shutdown();
        auditLogger.log("Task Manager shut down.");
        monitoringService.recordShutdown();
        System.out.println("Task Manager shut down.");
    }

    public enum TaskStatus {
        NEW,
        IN_PROGRESS,
        COMPLETED,
        FAILED
    }
}

public class ClientAuthenticator {
    private final Map<String, String> clientCredentials = new ConcurrentHashMap<>();
    private final Map<String, String> activeTokens = new ConcurrentHashMap<>();

    public void registerClient(String clientId, String password) {
        String hashedPassword = hashPassword(password);
        clientCredentials.put(clientId, hashedPassword);
    }

    public String authenticateClient(String clientId, String password) {
        String storedHash = clientCredentials.get(clientId);
        if (storedHash != null && verifyPassword(password, storedHash)) {
            String token = generateToken(clientId);
            activeTokens.put(clientId, token);
            return token;
        }
        return null;
```

```
    }

    public boolean verifyToken(String clientId, String token) {
        return token.equals(activeTokens.get(clientId));
    }

    private String hashPassword(String password) {

        return BCrypt.hashpw(password, BCrypt.gensalt());
    }

    private boolean verifyPassword(String password, String hash) {
        return BCrypt.checkpw(password, hash);
    }

    private String generateToken(String clientId) {

        return UUID.randomUUID().toString();
    }
}

ClientAuthenticator authenticator = new ClientAuthenticator();
            authenticator.registerClient("client1", "password123");


            String clientId = getClientIdFromRequest();
            String password = getPasswordFromRequest();
            String token = authenticator.authenticateClient(clientId, password);

            if (token == null) {
                System.out.println("Authentication failed for client: " + clientId);
                clientSocket.close();
                return;
            }

            System.out.println("Client authenticated: " + clientId);


public class Client {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "client_id", unique = true, nullable = false)
    private String clientId;

    @Column(name = "token", nullable = false)
    private String token;

    public Client() {}

    public Client(String clientId, String token) {
        this.clientId = clientId;
```

```
        this.token = token;
    }

    public Long getId() {
        return id;
    }

    public String getClientId() {
        return clientId;
    }

    public void setClientId(String clientId) {
        this.clientId = clientId;
    }

    public String getToken() {
        return token;
    }

    public void setToken(String token) {
        this.token = token;
    }
}

public void saveClient(Client client) {
    try (Session session = sessionFactory.openSession()) {
        session.beginTransaction();
        session.save(client);
        session.getTransaction().commit();
        System.out.println("Client saved: " + client.getClientId());
    } catch (Exception e) {
        System.err.println("Error saving client: " + e.getMessage());
    }
}


public static void main(String[] args) {
    try (Socket socket = new Socket("localhost", address);
         PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);
         BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

        ObjectMapper mapper = new ObjectMapper();

        while (true) {
            String serverMessage = reader.readLine();
            if (serverMessage == null) {
                System.err.println("Server disconnected.");
                break;
            }

            JsonNode jsonNode = mapper.readTree(serverMessage);
            if (jsonNode.has("instruction")) {
                long startTime = System.currentTimeMillis();
```

```
            System.out.println("Processing instruction: " + jsonNode);
            new InstructionExecutor.executeInstructions();
            long endTime = System.currentTimeMillis();


            JsonNode response = HardwareIndetification.collectHardwareInfo();
            writer.println(mapper.writeValueAsString(response));
          } else {
            long startTime = System.currentTimeMillis();
            new InstructionExecutor.executeInstructions();
            long computationTime = System.currentTimeMillis() - startTime;
            writer.println(computationTime);
          }
        }
    } catch (IOException | InterruptedException e) {
        System.err.println("Error in client: " + e.getMessage());
    }
}
```

```
        SystemInfo systemInfo = new SystemInfo();
HardwareAbstractionLayer hardware = systemInfo.getHardware();
Map<String, Object> hardwareInfo = collectHardwareInfo(hardware);
ObjectMapper objectMapper = new ObjectMapper();
String json = objectMapper.writerWithDefaultPrettyPrinter().writeValueAsString(hardwareInfo);
private static Map<String, Object> collectHardwareInfo(HardwareAbstractionLayer hardware) {
    Map<String, Object> hardwareInfo = new HashMap<>();
    CentralProcessor processor = hardware.getProcessor();
    Map<String, Object> cpuInfo = new HashMap<>();
    cpuInfo.put("name", processor.getProcessorIdentifier().getName());
    cpuInfo.put("architecture", System.getProperty("os.arch"));
    cpuInfo.put("coreCount", processor.getPhysicalProcessorCount());
    cpuInfo.put("threadCount", processor.getLogicalProcessorCount());
    hardwareInfo.put("cpu", cpuInfo);
    GlobalMemory memory = hardware.getMemory();
    Map<String, Object> memoryInfo = new HashMap<>();
    memoryInfo.put("total", memory.getTotal() / (1024 * 1024 * 1024) + " GB");
    memoryInfo.put("available", memory.getAvailable() / (1024 * 1024 * 1024) + " GB");
    hardwareInfo.put("memory", memoryInfo);
    List<Map<String, String>> storageInfo = new ArrayList<>();
    for (HWDiskStore disk : hardware.getDiskStores()) {
        Map<String, String> diskDetails = new HashMap<>();
        diskDetails.put("model", disk.getModel());
        diskDetails.put("serial", disk.getSerial());
        diskDetails.put("size", disk.getSize() / (1024 * 1024 * 1024) + " GB");
        storageInfo.add(diskDetails);
    }
    hardwareInfo.put("storage", storageInfo);
    List<Map<String, String>> networkInfo = new ArrayList<>();
    for (NetworkIF net : hardware.getNetworkIFs()) {
        Map<String, String> netDetails = new HashMap<>();
        netDetails.put("name", net.getName());
        netDetails.put("macAddress", net.getMacaddr());
```

```
            netDetails.put("ipAddress", net.getIPv4addr().length > 0 ? net.getIPv4addr()[0] : "N/A");
            networkInfo.add(netDetails);
        }
        hardwareInfo.put("network", networkInfo);
        ComputerSystem computerSystem = hardware.getComputerSystem();
        Map<String, String> motherboardInfo = new HashMap<>();
        motherboardInfo.put("manufacturer", computerSystem.getManufacturer());
        motherboardInfo.put("model", computerSystem.getModel());
        motherboardInfo.put("serial", computerSystem.getSerialNumber());
        hardwareInfo.put("motherboard", motherboardInfo);
        return hardwareInfo;

    public class Emulator {

        private long rax, rbx, rcx, rdx;
        private long rsi, rdi;
        private long rbp, rsp;
        private long r8, r9, r10, r11, r12, r13, r14, r15;
        private long rip;
        private long rflags;

        public Emulator() {
            this.rax = this.rbx = this.rcx = this.rdx = 0;
            this.rsi = this.rdi = 0;
            this.rbp = this.rsp = 0;
            this.r8 = this.r9 = this.r10 = this.r11 = 0;
            this.r12 = this.r13 = this.r14 = this.r15 = 0;
            this.rip = 0;
            this.rflags = 0;
        }

        public long getRAX() {
            return rax;
        }

        public void setRAX(long rax) {
            this.rax = rax;
        }

        public long getRBX() {
            return rbx;
        }

        public void setRBX(long rbx) {
            this.rbx = rbx;
        }

        public long getRCX() {
            return rcx;
        }

        public void setRCX(long rcx) {
            this.rcx = rcx;
```

```
    }

    public long getRDX() {
        return rdx;
    }
}

private Emulator emulator;
private Map<String, BiConsumer<String, String>> instructionMap;

public InstructionExecutor(Emulator emulator) {
    this.emulator = emulator;
    this.instructionMap = new HashMap<>();

    private void registerInstructions {
        instructionMap.put("add", this::add);
        instructionMap.put("sub", this::sub);
        instructionMap.put("mul", this::mul);
        instructionMap.put("imul", this::imul);
        instructionMap.put("div", this::div);
        instructionMap.put("idiv", this::idiv);
        instructionMap.put("inc", this::inc);
        instructionMap.put("dec", this::dec);
        instructionMap.put("adc", this::adc);
        instructionMap.put("sbb", this::sbb);
    }
}

public void executeInstructions(String filePath) {
    try {
        ObjectMapper objectMapper = new ObjectMapper();
        JsonNode root = objectMapper.readTree(new File(filePath));

        Iterator<JsonNode> instructions = root.elements();
        while (instructions.hasNext()) {
            JsonNode instructionNode = instructions.next();
            String instruction = instructionNode.get("instruction").asText();
            String operator1 = instructionNode.has("operator1") ? instructionNode.get("operator1").asText() :
null;
            String operator2 = instructionNode.has("operator2") &&
!instructionNode.get("operator2").isNull()
                    ? instructionNode.get("operator2").asText()
                    : null;
            executeInstruction(instruction, operator1, operator2);
        }
    } catch (IOException e) {
        System.err.println("Error reading JSON file: " + e.getMessage());
    }
}

private void executeInstruction(String instruction, String operator1, String operator2) {
    BiConsumer<String, String> executor = instructionMap.get(instruction);
```

```
    if (executor != null) {
        executor.accept(operator1, operator2);
    } else {
        System.out.println("Unknown instruction: " + instruction);
    }
}

private void add(String dest, String value) {
    long destValue = getRegisterValue(dest);
    long operand = parseOperand(value);
    long result = destValue + operand;
    setRegisterValue(dest, result);
    System.out.println("ADD executed: " + dest + " = " + result);
}

private void sub(String dest, String value) {
    long destValue = getRegisterValue(dest);
    long operand = parseOperand(value);
    long result = destValue - operand;
    setRegisterValue(dest, result);
    System.out.println("SUB executed: " + dest + " = " + result);
}

private void mul(String dest, String value) {
    long destValue = getRegisterValue(dest);
    long operand = parseOperand(value);
    long result = destValue * operand;
    setRegisterValue(dest, result);
    System.out.println("MUL executed: " + dest + " = " + result);
}

private void imul(String dest, String value) {
    long destValue = getRegisterValue(dest);
    long operand = parseOperand(value);
    long result = destValue * operand;
    setRegisterValue(dest, result);
    System.out.println("IMUL executed: " + dest + " = " + result);
}

private void div(String dest, String value) {
    long destValue = getRegisterValue(dest);
    long operand = parseOperand(value);
    if (operand == 0) {
        throw new ArithmeticException("Division by zero");
    }
    long result = destValue / operand;
    setRegisterValue(dest, result);
    System.out.println("DIV executed: " + dest + " = " + result);
}

private void idiv(String dest, String value) {
    long destValue = getRegisterValue(dest);
```

```
    long operand = parseOperand(value);
    if (operand == 0) {
        throw new ArithmeticException("Division by zero");
    }
    long result = destValue / operand;
    setRegisterValue(dest, result);
    System.out.println("IDIV executed: " + dest + " = " + result);
}

private void inc(String dest, String unused) {
    long destValue = getRegisterValue(dest);
    long result = destValue + 1;
    setRegisterValue(dest, result);
    System.out.println("INC executed: " + dest + " = " + result);
}

private void dec(String dest, String unused) {
    long destValue = getRegisterValue(dest);
    long result = destValue - 1;
    setRegisterValue(dest, result);
    System.out.println("DEC executed: " + dest + " = " + result);
}

private void adc(String dest, String value) {
    long destValue = getRegisterValue(dest);
    long operand = parseOperand(value);
    long carry = getCarryFlag();
    long result = destValue + operand + carry;
    setRegisterValue(dest, result);
    System.out.println("ADC executed: " + dest + " = " + result);
}

private void sbb(String dest, String value) {
    long destValue = getRegisterValue(dest);
    long operand = parseOperand(value);
    long borrow = getCarryFlag();
    long result = destValue - operand - borrow;
    setRegisterValue(dest, result);
    System.out.println("SBB executed: " + dest + " = " + result);
}

private String hash(String input) {
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] hashBytes = digest.digest(input.getBytes());

        StringBuilder hexString = new StringBuilder();
        for (byte b : hashBytes) {
            String hex = Integer.toHexString(0xff & b);
            if (hex.length() == 1) {
                hexString.append('0');
            }
```

```
            hexString.append(hex);
        }
        return hexString.toString();
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException("SHA-256 not supported", e);
    }
}

public String calculateHash(List<Map.Entry<Long, Long>> memoryPairs) {
    long registerSum = 0;
    for (List<Long> registerHistory : registerValuesOverTime) {
        for (long value : registerHistory) {
            registerSum += value;
        }
    }

    StringBuilder textRepresentation = new StringBuilder();
    textRepresentation.append(registerSum);

    for (Map.Entry<Long, Long> pair : memoryPairs) {
        long id = pair.getKey();
        long val = pair.getValue();
        textRepresentation.append(id + val);
    }

    return hash(textRepresentation.toString());
}

private void and(String dest, String value) {
    long destValue = getRegisterValue(dest);
    long operand = parseOperand(value);
    long result = destValue & operand;
    setRegisterValue(dest, result);
    System.out.println("AND executed: " + dest + " = " + result);
}

private void or(String dest, String value) {
    long destValue = getRegisterValue(dest);
    long operand = parseOperand(value);
    long result = destValue | operand;
    setRegisterValue(dest, result);
    System.out.println("OR executed: " + dest + " = " + result);
}

private void xor(String dest, String value) {
    long destValue = getRegisterValue(dest);
    long operand = parseOperand(value);
    long result = destValue ^ operand;
    setRegisterValue(dest, result);
    System.out.println("XOR executed: " + dest + " = " + result);
}
```

```
private void not(String dest, String unused) {
    long destValue = getRegisterValue(dest);
    long result = ~destValue;
    setRegisterValue(dest, result);
    System.out.println("NOT executed: " + dest + " = " + result);
}

private void shl(String dest, String value) {
    long destValue = getRegisterValue(dest);
    int shift = (int) parseOperand(value);
    long result = destValue << shift;
    setRegisterValue(dest, result);
    System.out.println("SHL executed: " + dest + " = " + result);
}

private void shr(String dest, String value) {
    long destValue = getRegisterValue(dest);
    int shift = (int) parseOperand(value);
    long result = destValue >>> shift;
    setRegisterValue(dest, result);
    System.out.println("SHR executed: " + dest + " = " + result);
}

/private void sal(String dest, String value) {
    shl(dest, value);
    System.out.println("SAL executed (same as SHL).");
}

private void sar(String dest, String value) {
    long destValue = getRegisterValue(dest);
    int shift = (int) parseOperand(value);
    long result = destValue >> shift;
    setRegisterValue(dest, result);
    System.out.println("SAR executed: " + dest + " = " + result);
}

private void rol(String dest, String value) {
    long destValue = getRegisterValue(dest);
    int shift = (int) parseOperand(value) % Long.SIZE;
    long result = (destValue << shift) | (destValue >>> (Long.SIZE - shift));
    setRegisterValue(dest, result);
    System.out.println("ROL executed: " + dest + " = " + result);
}

private void ror(String dest, String value) {
    long destValue = getRegisterValue(dest);
    int shift = (int) parseOperand(value) % Long.SIZE;
    long result = (destValue >>> shift) | (destValue << (Long.SIZE - shift));
    setRegisterValue(dest, result);
    System.out.println("ROR executed: " + dest + " = " + result);
}
```

ДОДАТОК Г.

ПРИКЛАДИ ПЕРЕТВОРЕННЯ КОДУ МЕТОДАМИ ОБФУСКАЦІЇ (ЧАСТКОВІ КОДИ)

Вплив методів обфускації (метод використання додаткового коду, метод перепризначення регістру)

Оригінальний код	Метод використання додаткового коду			Метод перепризначення регістру		
section data. operand1 dd 4 operand2 dd 3 constant dd 5 result dd 0 section .text global _start _start: mov eax, [operand1] mov ebx, [operand2] imul eax, ebx add eax, [constant] mov [result] ,eax ;exiting the program mov eax,1 xor ebx, ebx int 0x80	section data. operand1 dd 4 operand2 dd 3 constant dd 5 result dd 0 section .text global _start _start: mov eax, [operand1] mov ebx, [operand2] mov ecx, [result] cmp eax, ecx jg greater_than je equal	greater_than: mov eax, [operand1] mov ebx, [operand2] imul eax, ebx add eax, [constant] mov [result] ,eax equal: add eax,ebx mov [result], eax		section data. operand1 dd 4 operand2 dd 3 const dd 5 result dd 0 section .text global _start _start: mov ecx, [operand2] mov edx, [operand1] imul edx, ecx add edx, [const] mov [result], edx	;exiting the program mov eax,1 xor ebx, ebx int 0x80	

Вплив методів обфускації (метод заміни інструкцій, метод перемішування виконання підпрограм)

Оригінальний код	Метод заміни інструкцій			Метод перемішування виконання підпрограм		

section data.
operand1 dd 4
operand2 dd 3
constant dd 5
result dd 0
 	
section .text
global _start
 	
_start:
mov eax,
[operand1]
mov ebx,
[operand2]
imul eax, ebx
add eax,
[constant]
mov [result]
,eax
 	
;exiting the
program
mov eax,1
xor ebx, ebx
int 0x80
	section data.
	operand1 dd 4
	operand2 dd 3
	constant dd 5
	const dd 1
	result dd 0
	
	section .text
	global _start
	
	_start:
	mov eax,
	[operand1]
	mov ebx,
	[operand1]
	add eax, ebx
	add eax, ebx
	mov ecx,
	[constant]
		imul ecx,
		[const]
		add eax, ecx
		mov [result],
		eax
		
		;exiting the
		program
		mov eax,1
		xor ebx, ebx
		int 0x80
				section data.
				operand1 dd 4
				operand2 dd 3
				constant dd 4
				result dd 0
				
				section .text
				global _start
				
				Multiplication:
				mov eax,
				[operand1]
				mov ebx,
				[operand2]
				imul eax, ebx
				add eax,
				[result]
				mov [result],
				eax
				ret
				
				Addition:
				mov eax,
				[constant]
				add eax,
				[result]
					mov [result],
					eax
					ret
					
					Increment:
					mov eax, 1
					add eax,
					[result]
					mov [result],
					eax
					ret
					
					_start:
					call Increment
					call Addition
					call
					Multiplication
					
					;exiting the
					program
					mov eax,1
					xor ebx, ebx
					int 0x80	

Вплив методів обфускації (метод транспонування коду, метод перемішування виконання підпрограм)

Оригінальний код	Метод транспонування коду			Метод перемішування виконання підпрограм		

section data.
operand1 dd 4
operand2 dd 3
constant dd 5
result dd 0
 	
section .text
global _start
 	
_start:
mov eax,
[operand1]
mov ebx,
[operand2]
imul eax, ebx
add eax,
[constant]
mov [result]
,eax
 	
;exiting the
program
mov eax,1
xor ebx, ebx
int 0x80
	section data.
	operand1 dd 4
	operand2 dd 3
	constant dd 4
	result dd 0
	
	section .text
	global _start
	
	Increment:
	mov eax, 1
	add eax,
	[result]
	mov [result],
	eax
	ret
	
	Multiplication:
	mov eax,
	[operand1]
	mov ebx,
	[operand2]
	imul eax, ebx
	add eax,
	[result]
		mov [result],
		eax
		ret
		
		Addition:
		mov eax,
		[constant]
		add eax,
		[result]
		mov [result],
		eax
		ret
		
		_start:
		call
		Multiplication
		call Addition
		call Increment
		
		;exiting the
		program
		mov eax,1
		xor ebx, ebx
		int 0x80
				section data.
				operand1 dd 4
				operand2 dd 3
				constant dd 4
				result dd 0
				
				section .text
				global _start
				
				Multiplication_1:
				mov eax,
				[operand1]
				mov ebx,
				[operand2]
				imul eax, ebx
				ret
				
				Multiplication_2:
				add eax, [result]
				mov [result], eax
				ret
					Addition_1:
					mov eax,
					[constant]
					add eax,
					[result]
					ret
					
					Addition_2:
					mov
					[result],
					eax
					ret
					
					Increment:
					mov eax, 1
					add eax,
					[result]
					mov
					[result],
					eax
					ret
						_start:
						call
						Multiplication_1
						call
						Multiplication_2
						call Addition_1
						call Addition_2
						call Increment
						
						;exiting the
						program
						mov eax,1
						xor ebx, ebx
						int 0x80