НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Кваліфікаційна наукова
праця на правах рукопису

Давидов Вячеслав Вадимович

УДК 004.89+004.942

ДИСЕРТАЦІЯ

МОДЕЛІ ТА МЕТОДИ ПІДВИЩЕННЯ БЕЗПЕКИ БАЙТ-КОД ОРІЄНТОВАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ В УМОВАХ КІБЕРАТАК

Спеціальність 05.13.05 – Комп’ютерні системи та компоненти

Подається на здобуття наукового ступеня доктора технічних наук

Дисертація містить результати власних досліджень.
Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ В.В. Давидов

Науковий консультант:
Семенов Сергій Геннадійович,
доктор технічних наук,
професор

Харків – 2021

АНОТАЦІЯ

Давидов В.В. Моделі та методи підвищення безпеки байт-код орієнтованого програмного забезпечення в умовах кібератак. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора технічних наук за спеціальністю 05.13.05 «Комп'ютерні системи та компоненти». – Черкаський державний технологічний університет, Черкаси, 2021.

Дисертаційна робота присвячена вирішенню актуальної науково-технічної проблеми підвищення безпеки програмного забезпечення в умовах кібератак у комп'ютерних системах на основі розробки моделей та методів обфускації програмного коду та створення ліцензійних ключів для захисту авторських прав.

Проведено дослідження та порівняльний аналіз моделей та методів безпеки байт-код орієнтованого програмного забезпечення, який показав, що існуючі моделі і методи безпеки десктопного програмного забезпечення на системному рівні проектування в повній мірі не дозволяють усунути вплив засобів формування відповідних загроз безпеки. Тому існуючі моделі і методи захисту програмного забезпечення не відповідають вимогам якості, що регламентуються міжнародними стандартами та стандартами України щодо якості програмного забезпечення. Показано, що підвищення показників захищеності програмного забезпечення впливає на інші показники якості програмного забезпечення, а саме: переносимість, супроводжуваність, продуктивність.

Розроблено метод перевірки логіко-смислової подібності стандартних послідовних схем програм, що заснований на пошуку найбільш схожих з точки зору термальної історії шляхів в програмі. Цей метод використовує в якості опорної структури граф узгоджених маршрутів програм. Розмітка цього графу відповідає перетворенням термальних історій змінних на всіх

можливих маршрутів програми, а сам алгоритм, в свою чергу, зводиться до обчислення точних нижніх меж на множині підстановок. Ітеративна процедура, що здійснює це обчислення, і є процедурою побудови розмітки графа. Розроблений метод дозволив перейти до поліноміальних обчислювань пошуку лексемної історії шляхів на етапі розмітки графу для паралельних обчислень замість існуючих експоненціальних, що в свою чергу зменшило час перевірки коду на логіко-смислову подібність для визначення впливу розробленого методу обфускації коду на його коректність.

Розроблено GERT-модель процесу обфускації програмних модулів, в основі якої лежить математичний апарат гамма-розподілу на всіх етапах моделювання процесу обфускації. Це дозволило досягти уніфікації моделі в умовах модифікації GERT-мережі. Результати дослідження показали, що для розробленої математичної моделі при додаванні ще одного процесу обфускації дисперсія часу виконання збільшується на 12%, а при його видаленні з системи – зменшується до 13%. Математичне сподівання часу виконання змінюється в геометричній прогресії – так, при видаленні 1 вузла відбувається зменшення математичного сподівання на 9%, а при збільшенні на 1 вузол – збільшення математичного сподівання на 26%. Це показує незначність змін досліджуваних показників в умовах модифікації моделі і підтверджує гіпотезу про уніфікацію моделі в умовах використання математичного апарату гамма-розподілу як основного. Дані результати дають розробнику можливість спрогнозувати поведінку системи захисту програмних модулів з точки зору часу виконання. Це дозволяє зменшити час на прийняття рішення про доцільність використання процесу обфускації в умовах використання гнучких методологій.

Розроблено метод безпечного переходу в GERT-мережах, що використовуються в якості графа керуючої логіки програмного продукту. Дана логіка впроваджується в залежності від ідентифікаційного або серійного номера. Розроблено методологію масштабування розробленої

математичної моделі. Показана доцільність використання кожного типу масштабування з урахуванням критичності часу виконання перевірки безпеки програмного забезпечення на основі ліцензійних ідентифікаторів. Так, при вертикальному масштабуванні, в зв'язку з використанням паралельного процесу, час обробки даних практично не змінився. Це дає передумови використання даного типу масштабування при критичності часу виконання процесу, а також при необхідності істотного нарощування довжини ліцензійного ключа на слабких пристроях (наприклад, вбудованих пристроях та IoT). При горизонтальному масштабуванні, в зв'язку з використанням додаткових послідовних процесів обробки даних, час обробки значно збільшився (в 3,43 рази). Однак, введення додаткових послідовних дій збільшує час аналізу алгоритму поведінки. Це дає передумови використання даного типу масштабування при використанні прикладного програмного забезпечення, де час запуску програми не критичний.

Розроблено модель формування цифрового ідентифікатора програмного забезпечення, відмінною особливістю якої є використання індивідуальних даних про комп'ютерні системи кінцевого користувача для однозначної ідентифікації приналежності, на які ліцензійне програмне забезпечення встановлюється в процесі формування ліцензійного цифрового ідентифікатора. Це дає можливість підвищити безпеку програмного забезпечення шляхом захисту від неліцензійного копіювання. Запропоновано алгоритм функціонування системи і генерації ліцензійного ключа, адаптований до вхідних даних і можливих умов верифікації програмного забезпечення. Також, модель враховує можливість вбудовування довільного (заданого розробниками) коду в тіло ліцензійного ключа, який буде виконуватися при верифікації. Дані маніпуляції призвели до ускладнення аналізу і зламу ліцензійної складової програмного забезпечення. Це дозволило збільшити середній час зламу в 1,29 разів.

Розроблено відповідні алгоритми обфускації програмних модулів і генерації ліцензійних ключів та реалізовано систему безпеки програмного забезпечення, що базується на підсистемі роботи з ліцензійними ідентифікаторами та обфускацією програмного коду. Розроблено спосіб отримання метрик обфускованості для багатопроектного рішення на основі існуючих та удосконалених метрик якості коду, що дозволяє кількісно оцінити ступінь обфускованості коду. Дослідження розробленої системи дозволило підвищити безпеку байт-код орієнтованого програмного забезпечення в 1,4 разів. Проведено оцінку достовірності та ефективності запропонованих методів і моделей підвищення безпеки байт-код орієнтованого програмного забезпечення.

Ключові слова: кібератаки, захист авторських прав, водяні знаки, обфускація, GERT-мережі, безпека програмного забезпечення, метрики якості коду, цифрові ідентифікатори, логіко-смислова подібність, Standalone-застосунки.

SUMMARY

Davydov V. Models and methods of increasing the bytecode-oriented software security of during cyberattacks. – Qualifying scientific work on the rights of the manuscript.

The dissertation on competition of a scientific degree of the doctor of technical sciences on a specialty 05.13.05 "Computer systems and components". – Cherkasy State Technological University, Cherkasy, 2021.

The dissertation is devoted to solving the current scientific and technical problem of improving the security of software during cyberattacks in computer systems based on the development the models and methods of software code obfuscation and the license keys creation for copyright protection.

A study and comparative analysis of bytecode-oriented software models and methods security was conducted, which showed that the existing models and methods of desktop software security at the system level of design do not fully eliminate the impact of threats to the corresponding security threats. Therefore, the existing modules and methods of software protection do not meet the quality requirements regulated by international standards and standards of Ukraine on software quality. It is shown that increasing the security of software affects other indicators of software quality, namely: portability, maintainability, performance.

A method for checking the logical and semantic similarity of standard sequential program schemes has been developed, which is based on finding the most similar in terms of thermal history paths in the program. This method uses a graph of agreed program paths as a reference structure. The iterative procedure that performs this calculation is the procedure for constructing the graph markup. The developed method allowed to pass to polynomial calculations of thermal history ways search at a stage of the graph marking for parallel calculations instead of existing exponential. This allowed to reduce the time of checking the code for

logical and semantic similarity to determine the impact of the code obfuscation developed method on its correctness.

A GERT-model of the obfuscation process of software modules has been developed, which is based on the mathematical apparatus of the Gamma-distribution at all stages of modeling the obfuscation process. This allowed to achieve the model unification in terms of the GERT-network modification. The results of the study showed that for the developed mathematical model, when adding another obfuscation process, the variance of the execution time increases by 12%, and when it is removed from the system, it decreases to 13%. The mathematical expectation of execution time changes in a geometric progression – so, when removing 1 node there is a decrease in mathematical expectation by 9%, and when increasing by 1 node – an increase in mathematical expectation by 26%. This shows the insignificance of changes in the studied indicators in terms of model modification and confirms the hypothesis of model unification in terms of using the mathematical apparatus of the Gamma-distribution as the main one. These results allow the developer to predict the behavior of the software modules protection system in terms of execution time. This reduces the time to decide on the feasibility of using the obfuscation process in the use of flexible methodologies.

The method of safe transition in GERT-networks used as a graph of the control logic of the software product is developed. This logic is implemented depending on the identification or serial number. The methodology of scaling of the developed mathematical model by its horizontal and vertical scaling is developed. The expediency of using each type of scaling is shown taking into account the criticality of the execution time of the software security check based on the license identifiers. Thus, with vertical scaling, due to the use of a parallel data processing process, the data processing time has not changed. This gives the prerequisites for the use of this type of scaling at the critical time of the process, as well as the need to significantly increase the length of the license key on weak

devices (e.g., embedded devices and Internet of Things). When scaling horizontally, due to the use of additional sequential data processing processes, the processing time increased significantly (3.43 times). However, the introduction of additional sequential actions increases the analysis time of the behavior algorithm. This gives the prerequisites for the use of this type of scaling when using application software, where the start time of the program is not critical.

A model of digital software identifier creation has been developed. A distinctive feature of created identifier is the individual data on end-user computer systems usage for unambiguous identification of accessories for which licensed software is installed in the process of licensing digital identifier creation. This makes it possible to increase the security of the software by protecting against unlicensed copying. An algorithm for the operation of the system and the generation of a license key, adapted to the input data and possible conditions for software verification, is proposed. Also, the model takes into account the possibility of embedding an arbitrary (specified by the developers) code in the body of the license key, which will be executed during verification. This allowed to increase the average breaking time by 1.29 times.

Appropriate algorithms for software modules obfuscation and license keys generation have been developed and a software security system based on the subsystem of work with license identifiers and software code obfuscation has been implemented. A method for obtaining obfuscation metrics for a multi-project solution based on existing and improved code quality metrics has been developed, which allows to quantify the degree of code obfuscation. The study of the developed system allowed to increase the security of bytecode-oriented software by 1.4 times.

Keywords: cyberattacks, copyright protection, watermarks, obfuscation, GERT-networks, software security, code quality metrics, digital identifiers, logical and semantic similarity, Standalone applications.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

1. Semenov S., Davydov V., Lipchanska O., Lipchanskyi M. Development of unified mathematical model of programming modules obfuscation process based on graphic evaluation and review method // Eastern-European Journal of Enterprise Technologies. 2020. № 3(2(105)). P. 6–16. (**SCOPUS**).

2. Давидов В. В., Гавриленко С. Ю., Прохорова Т. М. Дослідження методів побудови синтаксичних аналізаторів // Системи обробки інформації. 2015. № 11(136). С. 125-128.

3. Давидов В. В., Семенов С. Г., Зиков І. С. Дослідження ризиків моніторингу технічного стану об’єктів авіації // Системи озброєння і військова техніка. 2015. № 4(44). С. 108-110.

4. Давидов В. В., Мовчан А. В., Сидоренко И. И. Разработка системы формирования цифрового идентификатора программного обеспечения для защиты авторских прав // Системи обробки інформації. 2016. № 3. С. 15-17.

5. Давыдов В. В., Гребенюк Д. С. Комплекс процедур генерации лицензионного ключа для защиты авторских прав интеллектуальной собственности на программное обеспечение // Системи управління, навігації та зв’язку. 2017. № 1(41). С. 11-15.

6. Давидов В. В., Пасько Д. А., Молчанов Г. І. Управління необмеженою кількістю хмарних сховищ // Сучасні інформаційні технології. 2018. Том 2. №3. С. 49–53.

7. Давидов В. В., Гавриленко С. Ю., Челак В. В. Разработка системы фиксации аномальных состояний компьютера // Вісник Національного технічного університету "ХПІ": Серія: Інформатика та моделювання. 2018. № 42 (1318). С. 109 – 121.

8. Давидов В. В., Бульба С. С., Кучук Г. А. Метод розподілу ресурсів між композитними застосунками // Системи управління, навігації та зв'язку. 2018. Том 4 (50). С. 99-104.

9. Семенов С. Г., Давидов В. В., Волошин Д. Г., Гребенюк Д. С. Метод захисту модуля програмного забезпечення на основі процедури обфускації // Телекомунікаційні та інформаційні технології. 2019. № 4 (65). С. 71–80.

10. Davydov V., Hrebeniuk D. Development of the methods for resource reallocation in cloud computing systems // Innovative Technologies and Scientific Solutions for Industries. 2020. № 3 (13), P. 25–33.

11. Давидов В. В., Можаєв М. А., Ліцзян Джан. Аналіз та порівняльні дослідження методів підвищення рівня безпеки програмного забезпечення // Сучасні інформаційні технології. 2020. Том 4. № 3. С. 124–132.

12. Давидов В. В., Гребенюк Д. С. Метод первинного виділення хмарних обчислювальних ресурсів на основі аналізу ієрархій // Системи управління, навігації та зв'язку. 2020. Том 3 (61). С. 80-85.

13. Davydov V., Hrebeniuk D. Development the resources load variation forecasting method within cloud computing systems // Advanced Information Systems. 2020. Vol. 4. No. 4. P. 128–135.

14. Золотухіна О. А., Волошин Д. Г., Давидов В. В., Бречко В. О. Розробка імітаційної моделі процесу розрахунку і коригування безпечної польотної траєкторії безпілотного літального апарату // Телекомунікаційні та інформаційні технології. 2020. № 4 (69). С. 87–94.

15. Semenov S., Davydov V., Hrebeniuk D. Research of the software security model and requirements // Advanced Information Systems. 2021. Vol. 5. № 1. P. 87–92.

16. Semenov S., Liqiang Zhang, Weiling Cao, Davydov V. Development of protecting a software product methematical model from unlicensed copying based on the GERT method // Information Processing Systems. 2021. № 1 (164). P. 73-82.

17. Liqiang Zhang, Weiling Cao, Davydov V., Brechko V. Analysis and comparative research of the main approaches to the mathematical formalization of the penetration testing process // Information Processing Systems. 2021. № 2 (64). C.73-77.

18. Kuchuk N., Cherneva G., Davydov V. Method of packet fragmentation in unstable data exchange in computer networks on transport // Mechanics Transport Communications: Academic journal. Vol. 19. Is. 1. 2021. P. X1-1 – X1-7, Article № 2064.

19. Кучук Н. Г., Давидов В. В. Моделі і методи захисту інформаційних структур для комп'ютерних систем на інтегрованих програмних платформах (монографія). Харків, 2021. 160 с.

20. Давидов В. В. Моделі та методи підвищення безпеки програмного забезпечення (монографія). Харків, 2021. 146 с.

21. Semenov S., Davydov V., Semenova A., Voloshyn D., Lymarenko V. Method of UAVs Quasi-Autonomous Positioning in the External Cyber Attacks Conditions // 10th International Conference on Dependable Systems, Services and Technologies (DESSERT). 2019. (**SCOPUS**).

22. S. Semenov S., Davydov V., Voloshyn D. Obfuscated Code Quality Measurement // Metrology and metrology assurance 2019: 29th International Scientific Symposium. September 6-9, 2019. Sozopol, Bulgaria. P. 44-47. (**SCOPUS**).

23. Semenov S., Davydov V., Voloshyn D. Data Protection Method of an Unmanned Aerial Vehicle based on Obfuscation Procedure // CEUR Workshop Proceedings, 2020, Volume 2654. P. 515-525. (**SCOPUS**).

24. Давыдов В. В. Анализ методов обнаружения злоумышленного кода в Android приложениях // Інформаційні технології, наука, техніка, технологія, освіта, здоров'я : міжнар. наук.-практ. конф., тези доповідей. Харків, 2015. С. 36.

25. Davydov V., Zmiivska V., Shypova T., Lysytsia D. Analysis of fractal noise indicators in measuring systems of technical objects // Metrology and metrology assurance 2018: 28th International Scientific Symposium, September 10-14, 2018 : Sozopol, Bulgaria. P. 44-47.

26. Kuchuk N., Shyman A., Hrebeniuk D., Davydov V. Mathematical model of the information system synthesis process // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління: міжнародна наук.-техн. конф., матеріали. – Баку: ВА ЗС АР; Харків: НТУ «ХПІ»; Київ: НАУ; Харків: ДП «ПДПРОНДІАВІАПРОМ»; Жиліна: Університет, 2021. С. 21.

27. Семенов С. Г., Давыдов В. В., Мовчан А. В. Система формирования цифрового идентификатора программного обеспечения для защиты авторских прав // Сучасні проблеми інформатики в управлінні, економіці, освіті та подоланні наслідків Чорнобильської катастрофи: міжнарод. наук. сем., тези доповідей. Київ, 2016. С. 110-116.

28. Семенов С. Г., Давидов В. В. Оценка рисков контроля и диагностики технического состояния объектов критического применения // Metrology and Metrology assurance 2016. 26th National Scientific Symposium with international participation. Sozopol, Bulgaria. 2016. P. 395-399.

29. Semenov S., Hrebeniuk D., Davydov V. Software copyright protection using identification key // Aviation in the XXI-st Century: the 7th World Congress, September 19-21, 2016: Kyiv, Ukraine. P. 1.10.20-1.10.23.

30. Давыдов В. В. Процедура генерации лицензионного ключа для защиты авторских прав // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління : міжнар. наук.-техн. конф., тези доповідей. Полтава, 2017. С. 52.

31. Давыдов В. В., Гребенюк Д. С. Особенности распределения ресурсов для многомашинных вычислительных комплексов // Проблеми

інформатизації: міжнар. наук.-техн. конф., тези доповідей. Полтава, 2017. С. 21.

32. Кучук Н. Г., Давидов В. В., Гребенюк Д. С. Аналіз методів розрахунку розмірності мінковського для фрактального трафіка мультисервісної мережі e-learning // Проблеми інформатизації: міжнародна наук.-техн. конф., тези доповідей. Черкаси, 2018. С. 34.

33. Давидов В. В., Волошин Д. Г. Семенов С. Г. Про завдання позиціонування безпілотних літальних апаратів в умовах кібератак // Актуальні питання протидії кіберзлочинності та торгівлі людьми: Всеукраїнська наук.-практич. конф., збірник матеріалів. Харків, 2018. С. 311.

34. Давидов В. В., Гребенюк Д. С. Метод прогнозування навантаження ресурсів хмарних обчислюваних систем // Проблеми інформатизації: міжнародна наук.-технічн. конф., тези доповідей. Черкаси. 2020. С. 73.

35. S Semenov S., Davydov V., Weilin C., Liqiang Z., Petrovskaya I. Enhanced Software vulnerability model // Інформаційні технології і безпека : міжнародна наук.-практ. конф., матеріали. Київ, 2020. С. 56-60.

36. Semenov S., Bartosh M., Davydov V., Turuta O. Improvement of the Task Scheduler Model Taking Into Account the Heterogeneity of the Entities // Fifth International Scientific and Technical Conference "Computer and Information Systems and Technologies". 2021. P. 42-43.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ГПВЧ – Генератор псевдовипадкових чисел
ДСТУ – Державні стандарти України
ІТ – Інформаційні технології
ЛКЗЗ – ліцензійний ключ захисту застосунків
ОС – Операційна система
ПЗ – Програмне забезпечення
СУБД – Системи управління базами даних
ЦП – Центральний процесор
ACL – Access Control List (списки контролю доступу)
CLR – Common Language Runtime
CRC – Cyclic Redundancy Check
DoS – Denial of Service
GERT – Graphical Evaluation and Review Technique
IEC – International Electrotechnical Commission
ISO – International Organization for Standardization
JVM – Java Virtual Machine
JWT – JSON Web Token
REST – Representational State Transfer
SSL – Secure Sockets Layer

ВСТУП

Актуальність. Повсюдне поширення комп'ютерних систем та технологій, збільшення попиту на інформаційні послуги, впровадження обчислювальних та інших засобів обробки даних у ключових сферах життя – це є основними ознаками сучасного суспільства. В той же час, в останні роки спостерігається збільшення кіберзагроз різної складності. При цьому, фахівці відзначають тенденції до зростання фінансових, іміджевих, соціальних та інших витрат у зв'язку з їх реалізацією.

Невід'ємною складовою існуючих комп'ютерних систем є програмне забезпечення. Проведені дослідження показали, що з точки зору безпеки, програмне забезпечення є однією з найбільш уразливих складових комп'ютерних систем. Пов'язано це з цілою низкою чинників об'єктивного і суб'єктивного характерів (висока вартість втрат і, як наслідок, підвищений інтерес зловмисників, недоліки сучасних засобів і платформ розробки програмного забезпечення, недостатня компетенція призначеного для користування персоналу та ін.).

Аналіз сучасних програмних продуктів показав тенденцію розвитку використання байт-код орієнтованого програмного забезпечення. Це пов'язано з можливістю створення платформо-незалежного коду, а також сучасних та ефективних механізмів роботи з пам'яттю (наприклад, через використання garbage collectors). Завдяки архітектурі платформо-незалежного коду (а саме зберігання проміжного коду, який можна декомпілювати), байт-код орієнтоване програмне забезпечення вразливе до кібератак, що пов'язані з порушенням конфіденційності та автентичності.

Проведені дослідження показали, що багато наукових праць присвячено проблематиці безпеки програмних засобів. Це роботи зарубіжних та вітчизняних науковців, серед яких варто відзначити наступних:

Рудницький В. [78, 79], Кузнецов О. [164, 165, 166, 167], Харченко В. [157, 159, 200, 230], Christodorescu M. [113, 114], Denning P. [126], Sheng Z. [225], Шибанов А. [95], Collberg C. [116, 117, 118], Fowler M. [132, 133], Летичевский А. [16, 58, 59 60, 61], Mohsen R. [178], Schrittwieser S. [212], Roy C. [208, 209, 210], Hua L. [144], Подимов В. [43, 72, 73, 74]. Однак, проблеми, пов'язані з безпекою байт-код орієнтованого програмного забезпечення в умовах кібератак на ліцензійні ідентифікатори та обфускацію в цих роботах не розкривались. Але це важливо в умовах підвищених вимог до конфіденційності та автентичності саме цього виду програмних продуктів.

Таким чином, на сьогоднішній день в теорії і практиці забезпечення безпеки програмних засобів загострилося **протиріччя** між підвищенням вимог до безпеки програмного забезпечення, збільшенням кіберзагроз на послуги автентифікації та конфіденційності та недостатньою якістю сучасних моделей та методів забезпечення безпеки байт-код орієнтованих програмних засобів, які не в змозі забезпечити необхідні якісні характеристики.

Подолати цю суперечність можна шляхом вирішення актуальної **науково-практичної проблеми** підвищення безпеки байт-код орієнтованого програмного коду в умовах кібератак на основі синтезу підсистеми забезпечення конфіденційності та автентичності програмних продуктів.

Зв'язок роботи з науковими програмами, планами, темами. Дисертаційна робота виконана у межах пріоритетних наукових напрямів, які охоплюють актуальні проблеми, відповідно до рішення Ради президентів академій наук України від 30 січня 2019 року «Про основні наукові напрями та найважливіші проблеми фундаментальних досліджень у галузі природничих, технічних, суспільних і гуманітарних наук Національної академії наук України на 2019-2023 роки», «Інформатика» за темами: «Розроблення і удосконалення методів верифікації та тестування баз знань», «Розроблення математичних методів та систем моделювання об'єктів та

процесів». Дисертаційну роботу виконано у межах зареєстрованих науково-дослідних робіт Національного технічного університету «Харківський політехнічний інститут»: «Дослідження методів управління та захисту даних в комп'ютеризованих інформаційно-вимірювальних та розподілених системах» (ДР №0119U002603) та «Створення завдань по мові програмування "С", тестування методологічної концепції, адаптації програми для інтеграції у систему вищої освіти України» (ДР №0119U103871).

Мета і задачі дослідження. Мета дисертаційної роботи – підвищення безпеки програмного забезпечення в умовах використання байт-код орієнтованих мов програмування. Мета дисертаційної роботи визначає необхідність розв'язання таких **основних задач**:

1. Порівняльний аналіз та дослідження моделей і методів забезпечення безпеки байт-код орієнтованого програмного забезпечення.
2. Розроблення показника безпеки програмних засобів.
3. Розроблення методу перевірки логіко-смислової подібності програм.
4. Розроблення GERT-моделі процесу обфускації програмних модулів з використанням парадигми математичного апарату гамма-розподілу в якості ключового на всіх етапах моделювання процесу обфускації.
5. Розроблення критерію якості обфускації коду для багатопроєктного рішення на основі дослідження та оцінки показників якості коду.
6. Розроблення методу обфускації програмних модулів для підвищення безпеки байт-код орієнтованого програмного забезпечення на основі розробленої GERT-моделі та критерію якості обфускації програмних модулів.
7. Розроблення моделі системи безпеки програмного забезпечення на основі математичного апарату моделювання GERT-мереж з

використанням операцій безпечного переходу та кодування ліцензійних ідентифікаторів.

8. Розроблення методу формування цифрового ідентифікатора програмного забезпечення для захисту авторських прав.

9. Розроблення імітаційної моделі систем формування вимог до безпеки та захисту програмного забезпечення.

10. Обґрунтування достовірності одержаних результатів наукових досліджень.

11. Розроблення практичних рекомендації щодо застосування розробленого методу підвищення безпеки байт-код орієнтованого програмного забезпечення.

12. Дослідження та впровадження розроблених моделей та методів підвищення безпеки байт-код орієнтованого програмного забезпечення в умовах кібератак.

Об’єктом дослідження є процес розробки та експлуатації програмного забезпечення.

Предметом дослідження є моделі і методи підвищення безпеки байт-код орієнтованого програмного забезпечення.

Методи дослідження. Для вирішення завдань математичного моделювання процесів обфускації використано теорію графічної оцінки та аналізу, теорію графів та теорію ймовірностей. Для створення програмної імітаційної моделі використано методи об’єктно-орієнтованого програмування та методи роботи з нереляційними базами даних. Для синтезу підсистем забезпечення конфіденційності та автентичності програмних продуктів було використано теорію статистичної обробки даних, теорію інформаційної безпеки та теорію технічного аналізу.

Наукова новизна одержаних результатів полягає у наступному:

– **Отримав подальший розвиток** метод перевірки логіко-смислової подібності програм складної логічної структури, що відрізняється від

відомих розпаралелюванням процесів, що обчислюються при порівнянні подібних елементів програмного коду, а також формуванням та використанням графу спільних обчислень в процесі пошуку подібних елементів коду. Це дозволило знизити складність процесу верифікації.

– **Вперше розроблена** GERT-модель процесу обфускації програмних модулів, що реалізує парадигми використання математичного апарату гамма-розподілу у якості ключового на всіх етапах моделювання процесу обфускації. Це дозволило досягти уніфікації моделі в умовах модифікації GERT-мережі.

– **Вдосконалено метод** обфускації програмних модулів, що відрізняється від відомих урахуванням варіативності типів лексем та ідентифікаторів. Це дозволило підвищити показник безпеки програмного забезпечення.

– **Вперше розроблено критерій** оцінки якості обфускації програмного забезпечення шляхом синтезу лінійної композиції часткових критеріїв метрик якості коду, що дозволило кількісно оцінити ступінь обфускованості програмних продуктів.

– **Вперше розроблено** модель безпечного переходу і кодування ліцензійних ідентифікаторів на основі математичного апарату GERT-мереж з парадигмою гамма-розподілу, що дозволило підвищити точність результатів моделювання.

– **Вдосконалено** метод формування цифрового ідентифікатора програмного забезпечення для захисту його авторських прав шляхом введення та удосконалення модулів менеджерів ліцензій, контроля цілісності та ідентифікації. Відмінною особливістю даної моделі є використання індивідуальних даних комп’ютерної системи кінцевого користувача для однозначної ідентифікації приналежності, що дозволило підвищити безпеку байт-код орієнтованих програмних застосунків в умовах кібератак.

Практичне значення одержаних результатів. Отримані в дисертаційній роботі результати дають змогу підвищити безпеку байт-код орієнтованих програмних застосунків, що в свою чергу дозволяє забезпечити достатній рівень захищеності програмного забезпечення в умовах кібератак.

Практична цінність роботи полягає у наступному:

– розроблено метод перевірки логіко-смислової подібності програм складної логічної структури для зменшення часу процесу верифікації;

– розроблено алгоритми обфускації програмного коду для підвищення захисту коду від реверс-інжинірингу;

– розроблено критерій оцінки якості обфускації програмного коду для підвищення точності оцінки безпеки програмного забезпечення;

– розроблено алгоритми формування персоналізованого ліцензійного ключа для захисту програмного забезпечення від неліцензійного копіювання;

– розроблено алгоритми захисту ліцензійних ключів від копіювання на основі прихованих переходів та кодування ліцензійного ключа.

Практичне значення отриманих результатів підтверджено відповідними актами впровадження.

Результати дисертації впроваджені і використовуються у діяльності компаній «Line Up», «Нікс Солюшенс ЛТД», Державного підприємства «Південний державний проектно-конструкторський та науково-дослідний інститут авіаційної промисловості», Державного підприємства «Харківський науково-дослідний інститут технологій машинобудування», а також використано у навчальному процесі Національного технічного університету «Харківський політехнічний інститут».

Особистий внесок здобувача. Усі наукові результати дисертаційної роботи автор отримав самостійно. Вони викладені як в роботах, які опубліковані без співавторів [20, 24, 30], так і у співавторстві. У друкованих працях, опублікованих у співавторстві, здобувачеві належать: [15, 28] – дослідження вимог безпеки до програмних засобів; [3, 7, 14, 19, 21, 26, 35] –

дослідження методів виявлення інформаційних та кібератак та впливів на системи обробки даних; [22] – розробка методу оцінки ступеню обфускованості коду; [23, 33] – розробка методу захисту програмного коду у вбудованих системах; [1, 16, 17] – дослідження методу графічної оцінки та аналізу для створення методів підвищення безпеки програмного забезпечення; [11, 25] – дослідження методів захисту програмного забезпечення; [9] – розробка методу обфускації програмного забезпечення; [2] – розробка способу побудови синтаксичного дерева коду; [4, 5, 27, 29] – розробка методу генерації ліцензійного ключа на основі даних комп'ютерної системи кінцевого користувача; [6, 8, 10, 12, 13, 18, 31, 32, 34, 36] – дослідження методів функціонування, масштабування та безпеки систем хмарних обчислень.

З робіт, що опубліковані у співавторстві, у дисертаційній роботі використовуються виключно результати, отримані особисто здобувачем.

Апробація результатів дисертації. Основні положення дисертаційної роботи доповідалися та обговорювалися на таких наукових конференціях та семінарах: XXIII Міжнародна науково-практична конференція «Інформаційні технології, наука, техніка, технологія, здоров'я» (Харків, 2015); XV Міжнародний науковий семінар «Сучасні проблеми інформатики в управлінні, економіці, освіті та подоланні наслідків Чорнобильської катастрофи» (Шацьк, 2016); 26th National Scientific Symposium with International Participation «Metrology and Metrology assurance» (Sozopol, Bulgaria, 2016); 7th World Congress «Aviation in the XXI-st Century» (Київ, 2016); VII Міжнародна науково-технічна конференція «Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління» (Полтава, 2017); V Міжнародна науково-технічна конференція «Проблеми інформатизації» (Полтава, 2017); 28th International Scientific Symposium «Metrology and Metrology Assurance» (MMA) (Sozopol, Bulgaria, 2018); VI Міжнародна науково-технічна конференція «Проблеми інформатизації»

(Черкаси, 2018); Всеукраїнська науково-практична конференція «Актуальні питання протидії кіберзлочинності та торгівлі людьми» (Харків, 2018); 10th International Conference on Dependable Systems, Services and Technologies (DESSERT) (Leeds, UK, 2019); 29th International Scientific Symposium «Metrology and Metrology Assurance» (MMA) (Sozopol, Bulgaria, 2019); Sun SITE Central Europe (CEUR) Workshop Proceedings (Kyiv, 2019); VIII Міжнародна науково-технічна конференція «Проблеми інформатизації» (Черкаси, 2020); XX Міжнародна науково-практична конференція «Інформаційні технології і безпека» (Київ, 2020); XI Міжнародна науково-технічна конференція «Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління» (Харків, 2021), Fifth International Scientific and Technical Conference «Computer and Information Systems and Technologies» (Kharkiv, 2021).

Публікації. Основні положення дисертації опубліковано в 36 наукових працях, у тому числі: 18 наукових статей (з них 1 входить до бази даних Scopus (другий квартіль); 1 – опубліковано у закордонному рецензованому виданні; 16 – у вітчизняних фахових наукових журналах), 16 тез доповідей (з них 3 входять до бази даних Scopus), а також 2 монографії (з них – 1 одноосібна).

Структура роботи та її обсяг. Дисертація складається зі вступу, шести розділів, загальних висновків, списку використаної літератури та додатків і містить 244 сторінок основного тексту, 49 рисунків, 14 таблиць, 243 джерел у списку літератури, 69 сторінок додатків. Загальний обсяг роботи 313 сторінок.

РОЗДІЛ 1. ПОРІВНЯЛЬНИЙ АНАЛІЗ ТА ДОСЛІДЖЕННЯ МОДЕЛЕЙ ТА МЕТОДІВ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ПОСТАНОВКА ЗАВДАННЯ ДОСЛІДЖЕННЯ

1.1. Дослідження моделей і вимог безпеки програмного забезпечення

У сучасному світі повсюдної комп'ютеризації, все більшу роль відіграє створення безпечних комп'ютерних систем. Це пов'язано з тим, що з розвитком інформаційних технологій збільшуються якісні та кількісні характеристики злочинних дій, пов'язаних з інформаційними технологіями та комп'ютерних системами, зокрема кібератаками. Збитки від кіберзлочинності досягають 100 млрд доларів на рік по всьому світу, і налічують мільйони доларів в рік по Україні. Таким чином, кіберзлочинність потрапила в п'ятірку найпоширеніших економічних злочинів в світі, і в Україні зокрема [11, 54]. Порівняння збитків від кіберзлочинності на цьому ж ресурсі за попередні роки [12, 54] показало тенденцію його збільшення.

Невід'ємною складовою існуючих комп'ютерних систем є програмне забезпечення.

Проведені дослідження [8, 18, 20, 29, 32, 99, 206, 207] показали, що, з точки зору безпеки, програмне забезпечення є однією з найбільш уразливих складових комп'ютерних систем. Пов'язано це з цілою низкою чинників об'єктивного та суб'єктивного характеру (висока вартість втрат і, як наслідок, підвищений інтерес зловмисників, недоліки сучасних засобів і платформ розробки програмного забезпечення, недостатня компетентність призначеного для користувача персоналу та ін.).

Однак, відповідно до міжнародного та українського законодавства вимоги до безпеки програмних засобів є одними з найбільш пріоритетних у

системі їх якісної оцінки. Наприклад, відповідно до закону України «Про авторські права» [38], програмне забезпечення є інтелектуальною власністю і вимагає захисту від піратства, плагіату, підробок різного роду, незаконного поширення та інших незаконних дій і втручань.

Таким чином, актуальною стає проблема підвищення безпеки програмного забезпечення. Метою дослідження є:

– дослідження моделі забезпечення безпеки програмного забезпечення з метою виявлення основних недоліків моделі, а також виявлення шляхів їх усунення;

– дослідження характеристик якості програмного забезпечення, що впливають на його захищеність, з метою виявлення можливості підвищення якості програмного забезпечення.

Узагальнена модель забезпечення безпеки програмних засобів приведена на рис. 1.1. До основних компонентів даної моделі можна віднести [86, 214]:

– нормативні документи (напр., ISO / ДСТУ стандарти), які регламентують процес забезпечення безпеки. Дані вимоги безпеки є жорсткими і обов’язковими. Це, в свою чергу, накладає обмеження, пов’язані з актуальністю даних документів. Так, багато нормативних документів, зокрема Закон «Про авторські права», мають останні поправки датовані 2016 роком. Таким чином, ці документи втрачають актуальність і вимагають коректування;

– власне, програмне забезпечення;

– методи та засоби забезпечення безпеки;

– загрози послуг безпеки – потенційно можлива подія, дія, процес або явище, яке може завдати шкоди програмного продукту і компанії, що випускає програмний продукт. Вони впливають на життєвий цикл програмного забезпечення;

– життєвий цикл програмного забезпечення - набір вимог, методологій розробки та тестування програмного забезпечення, що визначають процеси, види діяльності та завдання, які використовуються при поставці, розробленні, тестуванні, застосуванні, супроводі та припиненні застосування програмних продуктів.

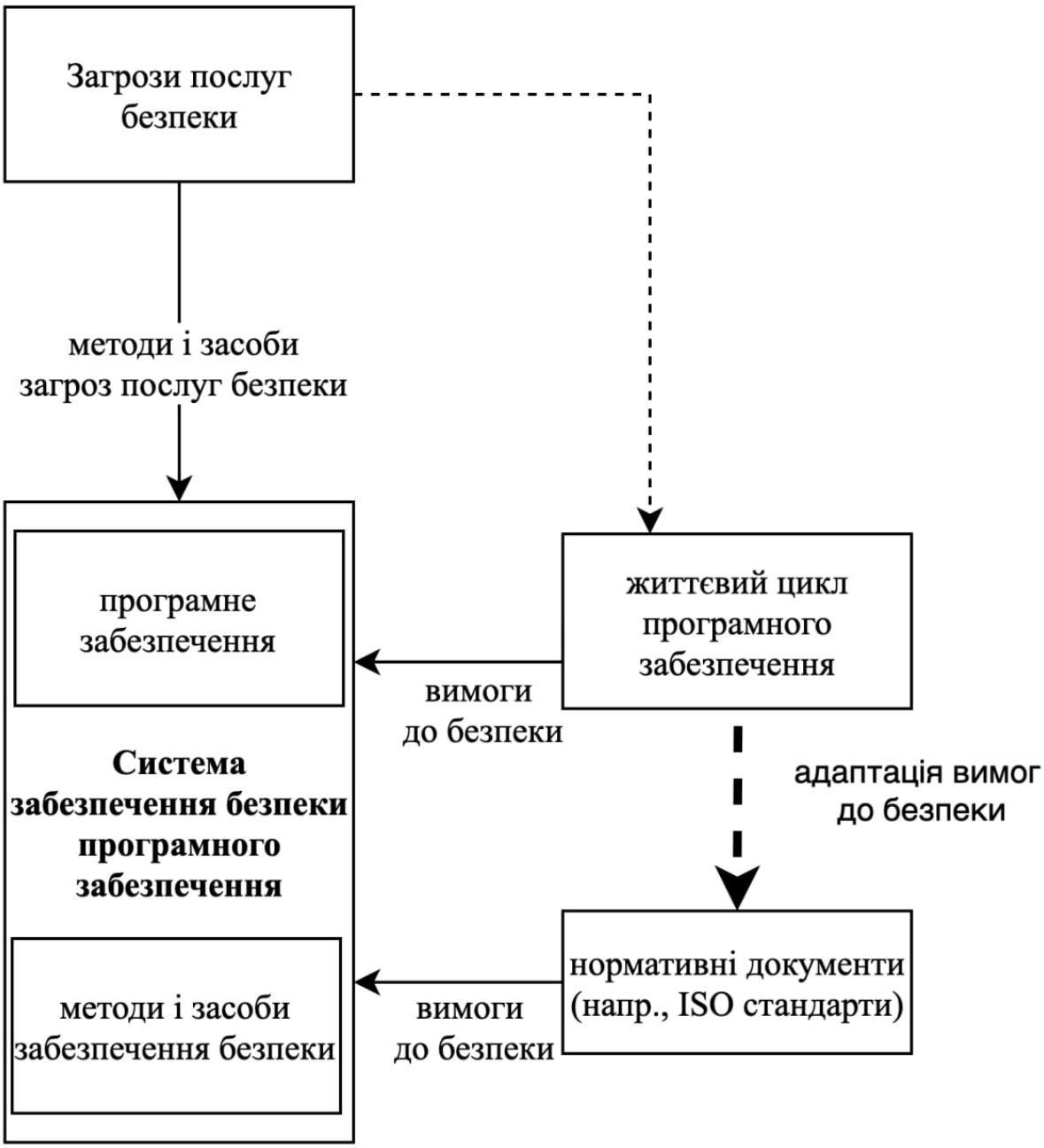


Рис. 1.1. Узагальнена модель забезпечення безпеки програмного забезпечення

Як видно з рис. 1.1, одним з важливих атрибутів представленої моделі є вимоги до безпеки. Вони регламентовані стандартами і життєвим циклом розробки. Розвиток технологій, методологій розробки вимагає коригування (адаптації) існуючих вимог до безпеки програмних засобів протягом усього життєвого циклу розробки програмного забезпечення. На жаль, у даний момент розробники стандартів і специфікацій в недостатній мірі приділяють увагу цьому питанню. Багато в чому це пов'язано з відсутністю можливості доступу до необхідної для цього конфіденційної інформації приватних компаній (кількість компаній, що випускають програмні продукти занадто велика, щоб всі їх рекомендації можна було врахувати; ряд вимог одних компаній можуть суперечити вимогам інших компаній, що може бути обумовлено розходженням фінансової підтримки і цільової аудиторії та ін.), і можливими динамічними змінами життєвого циклу програмного забезпечення.

На рис. 1.2 наведено опис життєвого циклу програмного продукту. Слід зазначити, що основними результатами виконання процесу аналізу вимог до програмних засобів є:

– визначення вимог до програмних елементів системи та їх інтерфейсів;

– аналіз вимог до програмних засобів на коректність та тестованість;

– аналіз впливу вимог до програмних засобів на середовище функціонування;

– виявлення сумісності і простежуваності між вимогами до програмних засобів та вимогами до системи;

– визначення пріоритетів реалізації вимог до програмних засобів;

– оцінка змін у вимогах до програмних засобів за вартістю, графіками робіт і технічними впливами.

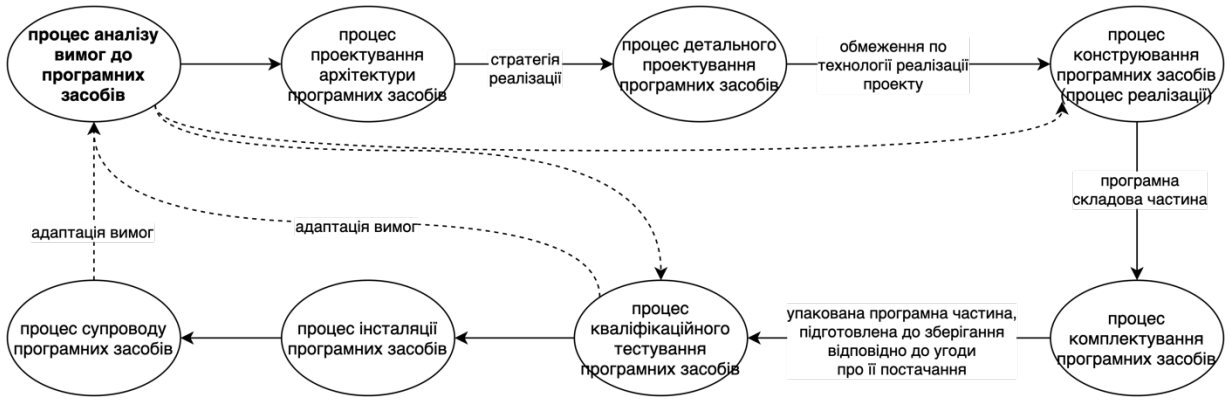


Рис. 1.2. Схематичний опис життєвого циклу програмного продукту

З рис. 1.2 видно, що результати аналізу вимог до програмних засобів істотно впливають на прийняття рішень, архітектуру проектування, розробки та тестування програмного забезпечення. При цьому, вимоги на кожному з етапів адаптуються з урахуванням виявленої специфіки програмного забезпечення, методологій розробки, засобів розробки (наприклад, фреймворки, шаблони проектування, середовище розробки) та інших факторів.

Слід також зазначити, що саме в процесах тестування і супроводу програмних засобів повинна відбуватися адаптація вимог.

Аналіз стандартів ISO 29148 [147], ISO 9126 [148] дозволив класифікувати вимоги до програмного продукту. Схематично, дані вимоги можна представити у вигляді рис. 1.3.

Так, виділяють 3 основні групи вимог:

– *системні вимоги* визначають зовнішні умови виконання системних функцій і обмежень на створення продукту, а також вимоги до опису програмно-апаратних підсистем. Системні вимоги накладають обмеження на архітектуру системи, засоби її візуального представлення і функціонування;

– *функціональні вимоги* - це перелік функцій або сервісів, які повинна надавати система, а також обмежень на дані і поведінку системи при її виконанні;

– *нефункціональні вимоги* визначають умови виконання функцій (наприклад, захист інформації в базі даних, автентифікація доступу до програмних засобів і т.д.) в середовищі, безпосередньо не пов'язані з функціями, а відображають потреби користувачів щодо їх виконання. Ці вимоги характеризують принципи взаємодії з середовищами або іншими системами, а також визначають показники часу роботи, захисту даних і досягнення якості з урахуванням рекомендацій використовуваного стандарту.

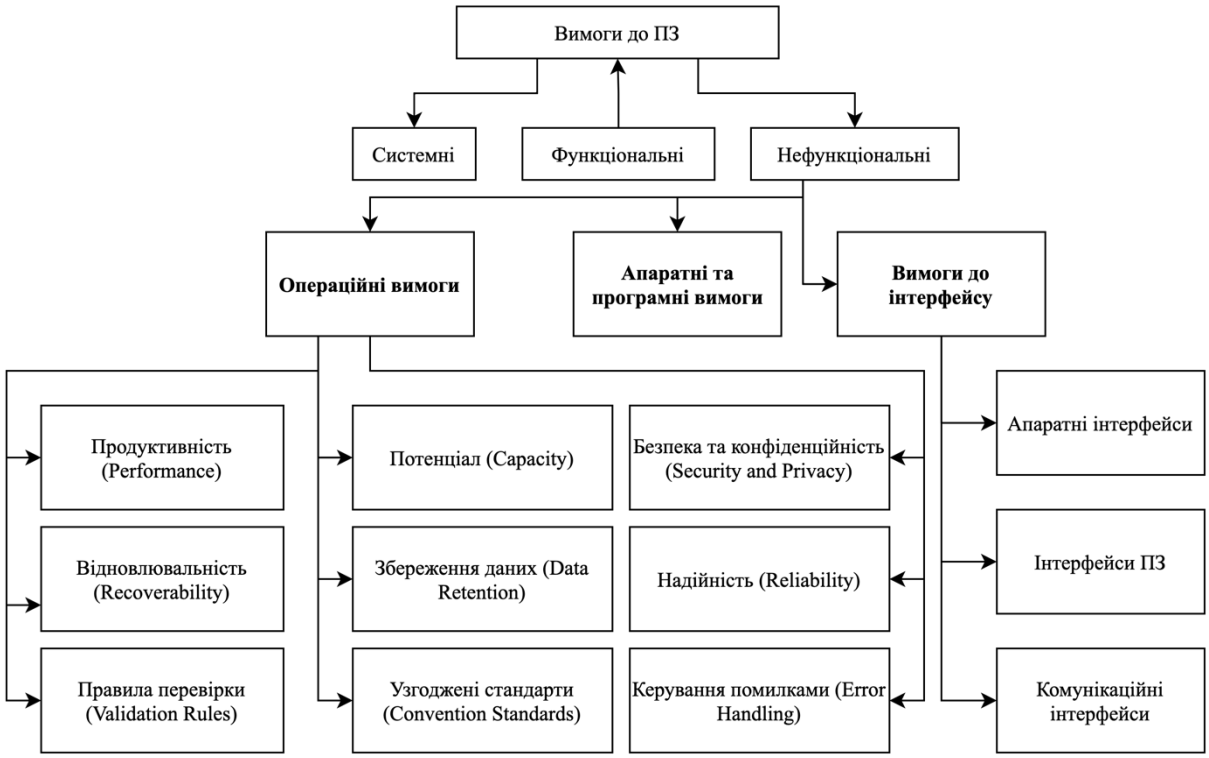


Рис. 1.3. Класифікація вимог до програмного забезпечення

Дослідження стандартів ISO 29148, ISO 9126 показало, що основної уваги заслуговують нефункціональні вимоги, зокрема, операційні. Ці вимоги повинні бути адаптивні:

– до різних програмних продуктів, які розробляються відповідно до життєвого циклу стандарту ISO 12207 [149]. На поточний момент вимоги визначені лише стандартами;

– до загроз послуг безпеки, які виникають в процесі життєвого циклу програмного забезпечення.

Наведені вище характеристики ПЗ є основними атрибутами його якості. У зв'язку з цим, в роботі були проаналізовані моделі якості, описані в стандартах і міжнародних документах:

– модель якості при використанні (ISO 25024) [150];

– модель якості продукту (ISO 25010) [151];

– модель якості даних (ISO 25012) [152].

Однак, серед перерахованих моделей, тільки модель якості продукту регламентує характеристики, пов'язані з безпекою програмного продукту. Так, на рис. 1.4 представлена класифікація характеристик якості ПЗ відповідно до стандарту ISO 25010. Дослідження стандарту дозволило виділити показники, що впливають на забезпечення безпеки. До них відноситься комплексна характеристика «захищеність», що включає в себе: конфіденційність, цілісність, непідробленість, справжність. Дані під-характеристики якості ПЗ є основоположними для наступних послуг безпеки: цілісність, автентифікація, конфіденційність, управління доступом.

У той же час, проведені дослідження [47, 75, 88] показали, що практична реалізація поставлених завдань підвищення захищеності може спричинити за собою погіршення інших показників якості ПЗ, наприклад, таких як:

– *переносимість*. Введення додаткових засобів захисту може спричинити за собою використання архітектурно-специфічних конструкцій, аналоги яких можна не знайти для інших;

– *продуктивність*. Механізми захисту програмного продукту, такі як обфускація і шифрування, позначаються на продуктивності і вимагають додаткових ресурсів процесора для виконання операцій;

– *супровідність*. Наприклад, виключення налагоджувальної інформації, необхідної для підвищення конфіденційності ПЗ, призводить до погіршення

аналізованості, зокрема, при отриманні стека викликів в разі виникнення помилок.

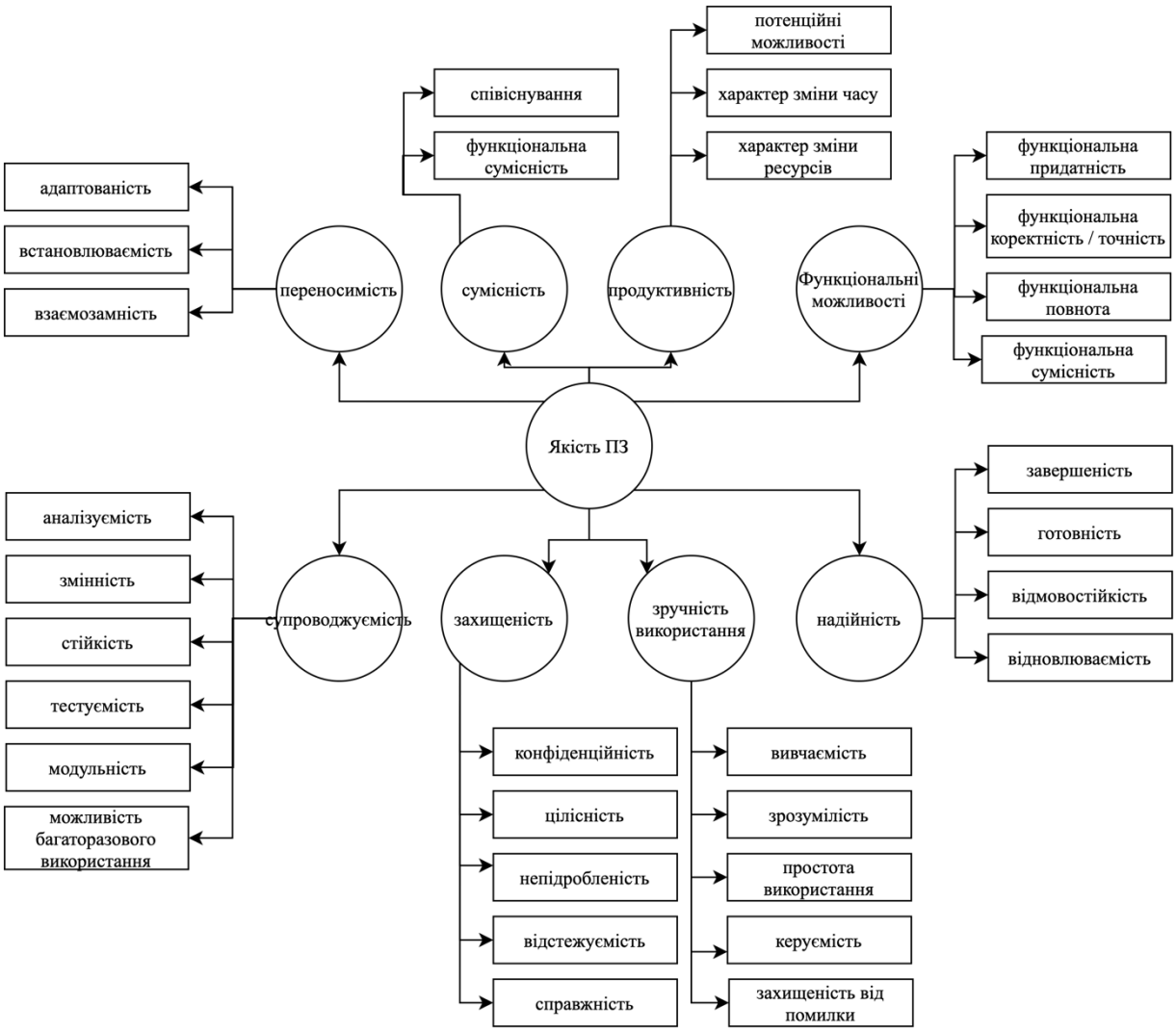


Рис. 1.4. Класифікація характеристик якості програмного забезпечення ISO/IEC 25010

Стандарт ISO 9126 визначає наступні метрики захищеності, що охоплюють досліджувані характеристики:

– здатність до відстеження доступів: X = A / B, де A – кількість «доступів користувача до системи і даних», зареєстрованих в базі даних передісторії доступів, B – кількість «доступів користувача до системи і даних», скоєних під час процесу оцінювання;

– контрольованість доступу: X = A / B, де A - кількість виявлених заборонених операцій різних типів, B - кількість заборонених операцій різних типів, зазначених у специфікації;

– запобігання спотворенню даних: X = 1 – A / N, де A - кількість виникнень основних випадків спотворення даних, N - кількість тестових даних, які намагалися викликати спотворення даних.

Також, в стандарті визначені метрики характеристик якості ПЗ, які мають негативний вплив на метрики захищеності:

– характеристика аналізованості: метрика здатності до аналізу (Чи може користувач точно встановити дію, яка призвела до відмови? Чи може фахівець із супроводу без роботи точно встановити дію, яка призвела до відмови?): X = A / B, де A - кількість даних, фактично зареєстрованих під час роботи, B - кількість даних, яку планувалося зареєструвати і достатня для того, щоб спостерігати за станом програмного забезпечення під час роботи;

– характеристика ефективності: метрика продуктивності (Скільки завдань можуть бути успішно виконані за даний період часу?): X = A / T, де A - кількість виконаних завдань, T - час спостереження;

– характеристика змінності: метрика складності модифікації (Чи може спеціаліст із супроводу легко змінити ПЗ, щоб вирішити проблему?): T = Sum (A / B) / N, де A - робочий час, витрачений на зміну, B - обсяг зміни програмного забезпечення, N - кількість змін;

– характеристика узгодженості і супроводжуваності: метрика ступеня узгодженості в супроводжуваності (Наскільки зручність супроводжуваності продукції відповідає застосуванням до положень, стандартів і угод?) X = 1 - A / B, де A - кількість пунктів узгодженості в супроводжуваності, які не були виконані під час випробувань, B - загальна кількість зазначених пунктів узгодженості в супроводжуваності.

На основі проведених досліджень сформовано матрицю впливу моделей і методів, що підвищують захищеність ПЗ на метрики інших характеристик якості ПЗ. Дані матриці представлені в табл. 1.1.

Таблиця 1.1. Вплив базових характеристик якості програмного забезпечення на характеристики його захищеності

	Здатність до відстеження доступів	Контрольованість доступу	Запобігання спотворенню даних
Здатність до аналізованості	+	—	—
Складність модифікації	+	+/-	+
Ступінь узгодженості і супроводжуваності	+	+/-	—
Продуктивність	+	+	+

Так, наприклад:

– для збільшення значення метрики здатності до відстеження доступів використовуються такі механізми, як обфускація, шифрування, списки контролю доступу (ACL) та ін. Дані механізми вносять додаткові нефункціональні (надлишкові) дії, що призводить до зниження продуктивності, здатності до аналізованості, а також збільшує складність модифікації і знижує ступінь супроводжуваності;

– для збільшення значення метрики контрольованості доступу використовуються механізми авторизації, цифрового підпису, списки контролю доступу (ACL). Додаткові перевірки призводять до зниження продуктивності. При можливості надання прав доступу уповноваженим представникам компанії, дані механізми не впливають на інші досліджувані метрики;

– для збільшення значення метрики запобігання спотворенню даних застосовуються методи цифрового підпису, контрольних сум, що збільшує складність модифікації, а також через використання надмірності коду впливає на продуктивність.

Таким чином, актуалізується необхідність вирішення завдання підвищення безпеки ПЗ з одночасним дотриманням вимог до основних показників його якості.

1.2. Дослідження загроз безпеки програмного забезпечення

1.2.1. Дослідження показника безпеки програмного забезпечення для різних типів застосунків

Проведені дослідження показали, що всі джерела загроз безпеці інформації можна розділити на три основні групи [6, 10, 18, 29, 35, 48, 49, 52, 91, 92, 135]: обумовлені діями суб'єкта (антропогенні джерела загроз); обумовлені технічними засобами (техногенні джерела небезпеки); обумовлені стихійними джерелами. При цьому щодо програмного забезпечення найбільш істотними є антропогенні джерела загроз.

Дані джерела загроз мають різну ступінь небезпеки, яку можна кількісно оцінити, провівши їх ранжування. Ці джерела впливають на показник безпеки програмного забезпечення $K_{без}$.

Як формування показника безпеки програмного забезпечення були досліджені існуючі функції:

– функція Лагранжа [17]:

$$W(X,\lambda) = S(X) + \sum L_i[F_i(X)], i = 1..m,$$

де $S(X)$ – цільова функція; L_i – неточний множник Лагранжа.

Дослідження даної функції показали, що вона має недолік, пов’язаний з тим, що аналітичні критерії обмежені розмірністю простору пошуку.

– – лінійна композиція приватних критеріїв, функція штрафу, додаткова функція:

$$W(X) = \sum \alpha_i \cdot F_i(X), i = 1..m,$$

де α_i – вагова функція. Однак, коефіцієнти α_i несуть суб'єктивний характер. Це вводить додаткові складності для об'єктів з неоднорідними фізичними критеріями.

З метою мінімізації недоліків досліджуваних функцій, була розроблена функція, на основі якої побудовано показник безпеки програмного забезпечення, перевагою якої є мінімізація впливу коефіцієнтів, що несуть суб'єктивний характер:

$$K_{без} = \sqrt[n]{\frac{1/\prod_{1}^{n} k_i + \sum_{1}^{n} k_i}{1/\sum_{1}^{n} k_i}}, \tag{1.1}$$

$$K_{без} \to \max, k_i \in [0..1],$$

де n – кількість характеристик, що описують безпеку програмного забезпечення. У нашому випадку використовується 3 характеристики;

k_1 – можливість виникнення джерела загрози. Визначає ступінь можливості використання вразливостей. Джерела загроз можуть використовувати вразливості для порушення безпеки інформації, отримання незаконної вигоди (нанесення шкоди власнику, користувачу інформації). Дана характеристика безпосередньо залежить від рівня захищеності програмного забезпечення [55, 63];

$$k_1 = f(k_c, k_i, k_a), k_1 \to 1;$$

k_2 – інвестиційна привабливість. Характеризується інвестиційною привабливістю програмного забезпечення та інвестиційною привабливістю корпорації, що володіє програмним продуктом (рис. 1.5). В рамках дослідження велику значимість має лише інвестиційна привабливість лише програмного забезпечення. Дана характеристика описується наступними показниками [22, 28, 36]:

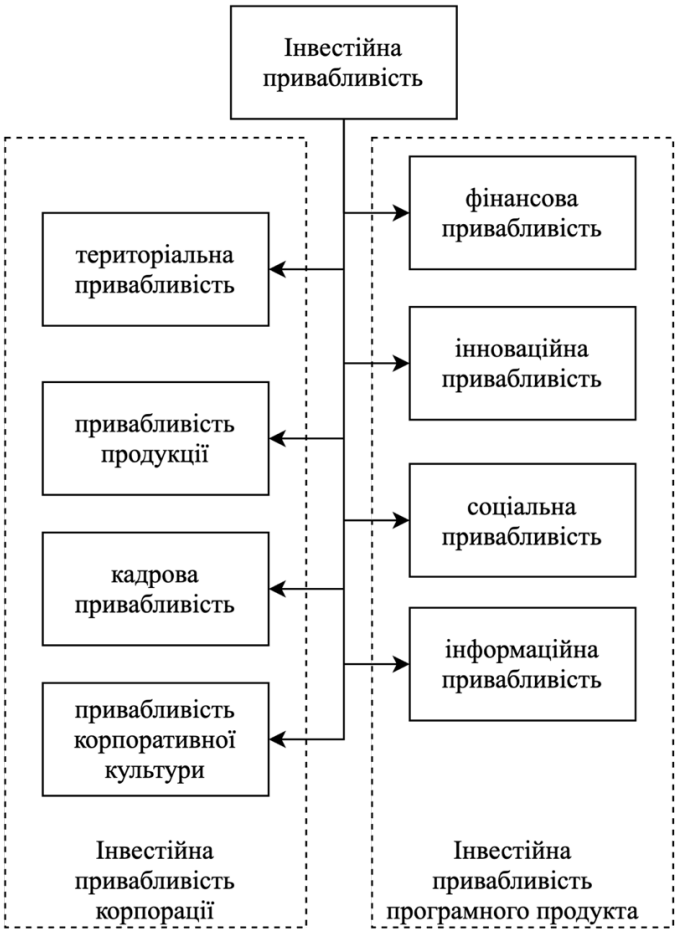


Рис. 1.5. Класифікація інвестиційної привабливості

– фінансова привабливість програмного продукту. Визначається оціночною ринковою вартістю програмного продукту;

– інноваційна привабливість. Визначається об’ємом наукових досліджень (наприклад, патентів), покладених в основі програмного продукту, а також перспективністю подальших досліджень;

– соціальна привабливість. Визначається спрямованістю програмного продукту: вузькоспеціалізований програмний продукт, продукт масового використання;

– інформаційна привабливість. Визначається об’ємом інформації, яку можна використовувати для подальших погроз споживачам даного програмного продукту.

$$k_2 = A = f\left(A_f, A_{it}, A_s, A_{in}\right).$$

k_3 – ступінь кваліфікації суб’єкта атаки. Дослідження літератури [45, 56, 77, 89] дозволило сформувати класифікацію суб’єкта атаки (рис. 1.6).

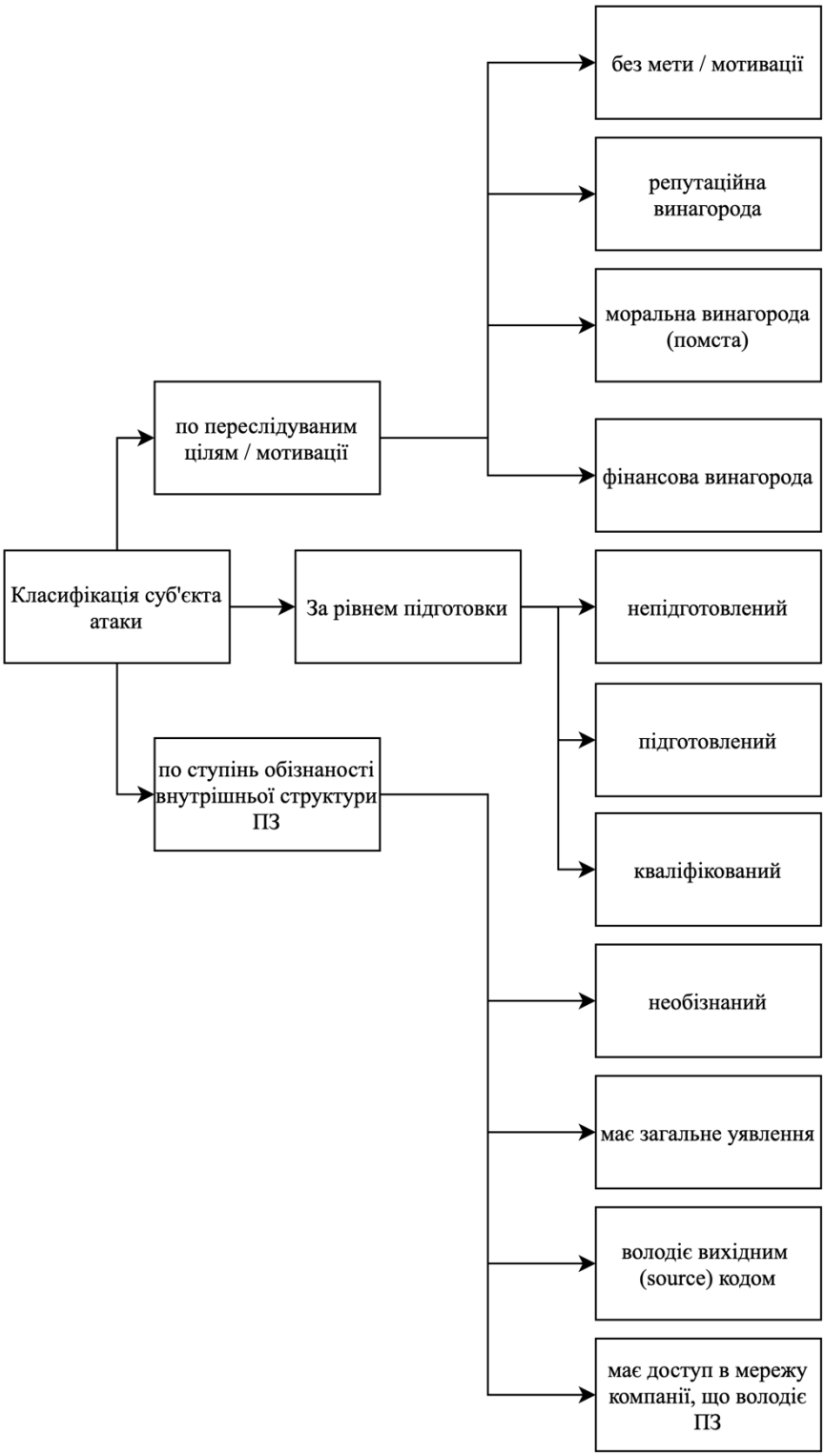


Рис. 1.6. Класифікація суб’єкта атаки

Так, суб’єкт атаки, зловмисник, характеризується:

– рівнем підготовки (починаючи зі студента, який вирішив вперше зламати програмний продукт, закінчуючи професійним реверс-інженером);

– ступенем обізнаності про внутрішню структуру ПЗ (вперше бачить, користувався як «кінцевий користувач», володіє кодами певної версії (або був одним з розробників), має доступ в корпоративну мережу або зв’язок з поточними розробниками);

– переслідуваними цілями (без корисливих цілей, отримання репутаційної або матеріальної винагороди. Окремим критерієм є моральна винагорода, наприклад, помста, що в ряді випадків має більш високу ступінь зацікавленості зловмисним впливом).

$$k_3 = Q = f\left(Q_p, Q_i, Q_m\right).$$

Експертним шляхом були обчислені значення коефіцієнтів для досліджуваних характеристик. Результати дослідження представлені в табл. 1.2 для двох основних категорій програмних продуктів [34, 125, 143, 159, 173, 217, 230]:

– Standalone-застосунки (desktop застосунки, «товсті» клієнти) – програмне забезпечення, яке повністю встановлюється на цільову комп’ютерну систему. До цього типу належать більшість встановлюваних в цільову комп’ютерну систему програмних засобів, наприклад, середовища розробки, обробки фотографій, проектування;

– Web-застосунки – тип програмного забезпечення, який характеризується тим, що в цільову комп’ютерну систему встановлюється тільки легкий клієнт, який взаємодіє з серверною частиною по протоколам обміну клієнт-серверних повідомлень, наприклад, за допомогою RestAPI, SOAP. При цьому використовуються сучасні засоби авторизації, такі як OAuth2 з використанням JWT-токенів [143, 157, 226].

Оскільки коефіцієнти k_2, k_3 несуть суб'єктивний характер, було прийнято встановити їм «базове» значення рівне 1. Відносні відхилення значень сформовано на основі наступних фактів:

– для standalone-застосунків засобів і методів здійснення злочинного вторгнення більше, ніж у відповідних web-застосунках;

– можливостей отримання інформації про standalone-застосунки більше, ніж для відповідних web-застосунків;

– інформаційна привабливість web-застосунків вище, тому що є узагальнена база клієнтів;

– соціальна привабливість web-застосунків вище, тому що розрахована на велику аудиторію і варіативність платформ;

– застосунки типу Standalone мають велику ймовірність виникнення загрози. Це пов'язано з тим, що тіло застосунку (бінарний код) цієї категорії повністю доступне зловмисникові і ймовірність зламу визначається часом, витраченим зловмисником на реверс-інжиніринг;

– основний код Web-застосунків прихований від злочинного впливу. Таким чином, ймовірність зламу застосунків даного типу визначається наявністю вразливостей, більшою мірою певних OWASP-стандартів. Однак, на сайті owasp.org визначені всі ці можливі загрози і механізми усунення їх. У доповнення до цього, існують комерційні тестування на проникнення [197], виправлення помилок яких дозволить знизити ймовірність злочинного вторгнення практично до 0;

– фатальність загроз для Web-застосунків набагато більше, так як дискредитація серверної частини дозволить отримати конфіденційну інформацію про всіх клієнтів, а також до можливості дискредитації інших застосунків даного виробника. Прикладом може служити досвід таких великих компаній як eBay, Adobe, LinkedIn, Instagram [2, 229]. Дискредитація standalone-застосунків обмежена конкретним користувачем і конкретною лінійкою програмного продукту (в тому числі, конкретною її версією).

Нехтування захищеністю standalone-застосунків призводить до загрози неліцензійного тиражування даного програмного продукту, що призводить до втрати прибутку компанії. Іншим фактором дискредитації даного типу програмного забезпечення є можливість реверс-інжинірингу з метою отримання алгоритму роботи важливих компонентів, що може призводить до відтворення роботи програми, створення конкурентних аналогів і втрати ринку продажів компанії.

Незважаючи на свої переваги, Web-застосунки мають недоліки [31, 57, 111, 120, 121, 160, 213, 216, 219]:

– необхідність постійного підключення до Інтернету та серверів компанії. Це особливо важливо, коли цільова комп'ютерна система належить закритій компанії з обмеженим виходом в Інтернет, зокрема, до НЕ довірених (не сертифікованих) ресурсів;

– можливість атаки Man-In-The-Middle з метою аналізу трафіку (сніффінг);

– можливість атаки Man-In-The-Middle з метою підміни адреси компанії-виробника програмного забезпечення для видачі неправдивих даних.

Таблиця 1.2. Коефіцієнти безпеки загроз для Standalone і Web-застосунків

Тип ПЗ	k_1	k_2	k_3	$K_{без}$
Standalone-застосунки	0.4	1	0.59	2.01
Web-застосунки	0.71	0.66	1	2.20

За представленими у табл. 1.2 даними можна зробити висновок про доцільність пріоритетності забезпечення безпеки саме Standalone-застосунків, тому що значення коефіцієнтів безпеки там вище.

Таким чином:

– розроблено показник безпеки загроз програмних продуктів, що відрізняється від відомих мінімізацією суб’єктивності вагових коефіцієнтів кожного показника;

– виділено коефіцієнти розробленого показника, які мають найбільший вплив на захищеність програмного забезпечення.

1.2.2 Дослідження загроз безпеки на різних рівнях архітектури програмного забезпечення

Аналіз літератури [29, 133] дозволив виділити 5 шарів застосунків, які притаманні всім сучасним застосункам (як Standalone-застосункам, так і Web-застосункам) (рис. 1.7):

– шар, призначений для користувача (взаємодія з користувачем, графічний інтерфейс користувача, зовнішні пристрої);

– репрезентативний шар (SingleSignOn, керування сесіями, створення контенту, його упорядкування та доставка контенту (напр., JSP));

– бізнес-шар (бізнес-логіка, транзакції, дані, сервіси, EJB);

– шар взаємодії (інтеграції) (взаємозв’язок із зовнішніми системами, JMS, JDBC);

– ресурсний шар (ресурси, дані і зовнішні системи, напр. СУБД);

– операційний шар (шар взаємодії з операційною системою).

У зв’язку з тим, що деякі шари мають схожі функціональні особливості, в рамках дослідження було прийнято розмежування шарів на більш високорівневі характеристики – рівні:

– прикладний рівень (включає в себе користувальницький і репрезентативний шари);

– системний рівень (включає в себе бізнес та інтеграційний шари);

– ресурсний рівень (включає в себе ресурсний шар);

– операційний рівень (включає в себе операційний шар).

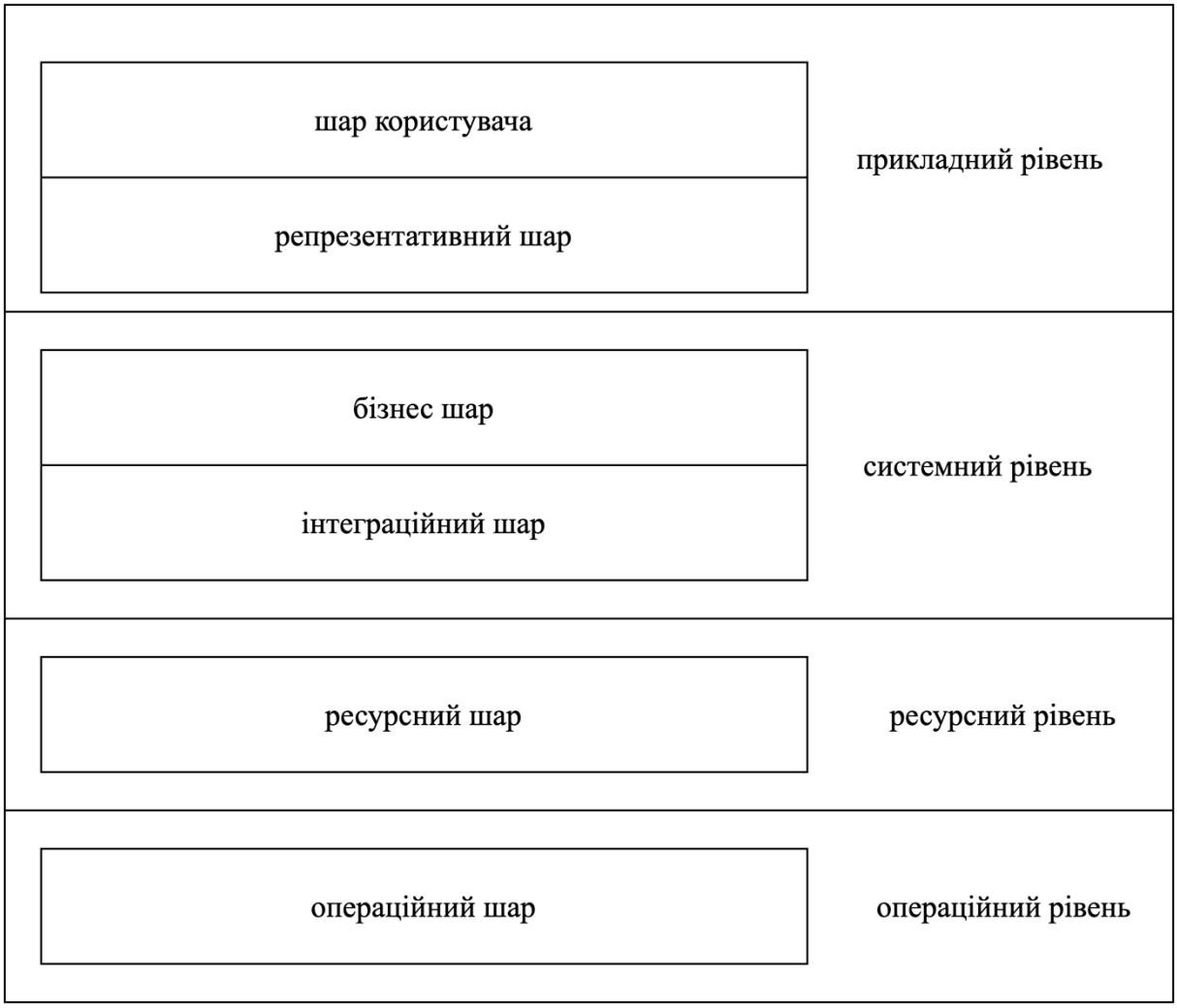


Рис. 1.7. Рівні і шари сучасного застосунку

Аналіз репрезентативного рівня проектування дозволив виділити наступні можливі атаки, що представляють загрозу безпеці [19, 23, 24, 27, 157, 230]:

- CORS (Cross-origin resource sharing) атаки. Суть даної атаки полягає в можливості здійснення Ajax-запиту з сайту А на сайт Б (який може бути зловмисним). Метод захисту від даного виду атаки - надання сервером заголовка Access-Control-Allow-Origin, який буде містити список довірчих доменів. Даний заголовок підтримується всіма сучасними браузерами.
- CSRF (Cross-Site Request Forgery) атаки. Суть даної атаки полягає в непомітності браузерами дії користувача. Так, браузери не розуміють, як

розрізнити, чи було дію явно скоєно користувачем (як, скажімо, натискання кнопки на формі або перехід за посиланням) або користувач ненавмисно виконав цю дію (наприклад, при відвідуванні bad.com, ресурсом був відправлений запит на good.com/some_action, в той час як користувач вже був авторизований на good.com). На поточний момент існує 3 методи захисту веб-ресурсу від даного виду атак, заснованого на токенах: Synchronizer Tokens, Double Submit Cookie, Encrypted Token, реалізація яких описана в [187];

- атаки «ін'єкції коду» (XSS (Cross-Site Scripting), HTML / SQL ін'єкції). Дана категорія атак полягає в можливості введення в текстові поля і значення аргументів запиту виконуваний код (наприклад, на мовах Javascript, SQL), який дозволить отримати неавторизований доступ до ресурсів. Існуючі методи захисту від даного виду атак представлені у вигляді практичних рекомендацій [157, 185];
- атаки Standalone-застосунків. У світі Java сучасною технологією розробки графічного інтерфейсу є технологія JavaFX, однак дослідження літератури [119] показує, що вона має ряд непереборних вразливостей, які на теперішній момент не виправлені і можуть бути використані в злочинних цілях. На додаток, методи і засоби атак на системному рівні проектування також мають місце бути.

Таким чином, існуючі методи захисту програмних засобів на прикладному рівні проектування (які притаманні Web-застосункам) представлені у вигляді практичних рекомендацій на сайті owasp.org. У свою чергу, засоби розробки графічного інтерфейсу для Standalone-застосунків мають ряд не усунених вразливостей, які можуть бути використані зловмисниками.

Аналіз системного рівня проектування показав, що прямий доступ до виконуваного коду програмного продукту для Standalone-застосунків дозволяє використовувати механізми трасування і байт-код маніпуляції, які

дозволяють порушити такі послуги безпеки програмного продукту як цілісність, конфіденційність і управління доступом. Послуги безпеки, механізми їх забезпечення та механізми формування загроз послуг безпеки представлені на рис. 1.8.

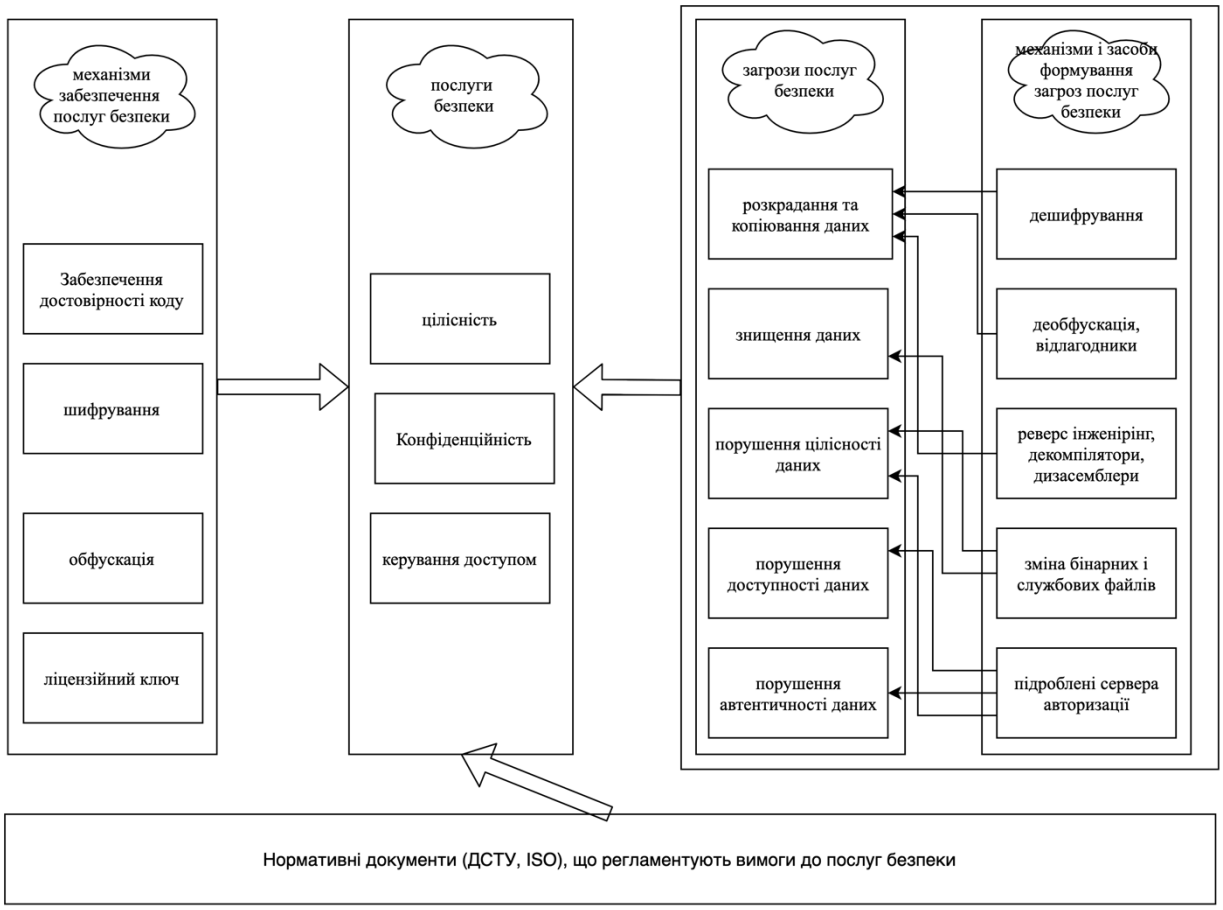


Рис. 1.8. Загрози безпеки на системному рівні проектування і їх зв’язок з характеристиками захищеності ПЗ

Так,

– для забезпечення цілісності використовуються механізми забезпечення достовірності коду. Порушення даної послуги безпеки може бути забезпечено механізмами зміни бінарних і службових файлів, а також декомпіляторами;

– для забезпечення конфіденційності використовуються механізми шифрування і обфускації. Порушення даної послуги безпеки може бути

забезпечено використанням деобфускаторів, трасувальників, реверс-інжинірингу, підробленими серверами авторизації;

– для забезпечення послуги управління доступом використовуються механізми ліцензійних ключів. Порушення даної послуги безпеки може бути забезпечено використанням підроблених серверів авторизації і зміною бінарних файлів [22, 122, 161].

Дослідження ринку програмних засобів, що забезпечують вищевказані послуги безпеки, показало:

– наявність наступних обфускаторів (як комерційних, так і з відкритим вихідним кодом): ProGuard, JODE, RetroGuard, Klassmaster, Allatori, Stinger. Аналіз даних програмних засобів показав високий ступінь обфускованості коду. Однак, дослідження ринку деобфускаторів і декомпіляторів (https://github.com/java-deobfuscator/deobfuscator, JD-GUI, Fernflower, Threadtear) показало наявність ряду відкритих програм, які справляються з більшою частиною захистів зазначених обфускаторів. Решту зловмисник буде здатний обробити за допомогою реверс-інжинірингу за менший період часу (в залежності від його кваліфікації);

– на поточний час немає уніфікованих бібліотек формування ліцензійного ключа - кожна компанія використовує свій підхід до його формування. Однак, реверс-інжиніринг ряду існуючих комерційних програмних засобів дозволив виділити наступні типи ліцензійних ключів:

– статичний ліцензійний ключ. Даний ліцензійний ключ не має інформації про власника, а отже, схильний до несанкціонованого тиражування;

– інформація про власника знаходиться в зашифрованому симетричним ключем вигляді. Недолік даного типу ліцензійного ключа полягає в тому, що ключ для його шифрування / дешифрування і алгоритм шифрування знаходяться в самому програмному продукті. Маючи ключ шифрування і алгоритм дешифрування, зловмисник має можливість створити

свій ліцензійний ключ, який буде достовірним з точки зору роботи програмного забезпечення;

– інформація про власника знаходиться у відкритому вигляді, проте вона має цифровий підпис. Публічний ключ для верифікації знаходиться всередині програмного продукту. Маючи можливість змінити публічний ключ, зловмисник може згенерувати свою пару ключів, сформувати підроблені дані, підписати своїм ліцензійним ключем, і замінити публічний ключ в самому програмному продукті;

– дані про користувача і програмний продукт знаходяться на віддаленому сервері, до якого звертається поточний програмний продукт. Найчастіше, даний підхід використовується для ліцензійних ключів типу «підписка» (на місяць, рік). Його недоліком є можливість формування зловмисником підробленого сервера, який буде повертати завжди позитивну відповідь;

– всі алгоритми шифрування даних і коду, з можливістю їх повного відновлення мають симетричний ключ і симетричний алгоритм шифрування (найпоширеніший - AES). Знаючи алгоритм шифрування і ключ, зловмисник зможе без проблем відновити оригінальний код;

– забезпечення достовірності коду і методи обходу даних описані в роботі [239]. На поточний момент відсутній механізм забезпечення достовірності коду програмного продукту, який дозволив би забезпечити послугу безпеки цілісності програмного продукту.

Таким чином, системний рівень проектування для Standalone-застосунків має високий коефіцієнт небезпеки, пов’язаний з відсутністю існуючих засобів, в повній мірі дозволяючих усунути вплив засобів формування загроз відповідних загроз безпеки.

Слід зазначити, що для Web-застосунків атаки, спрямовані на даний шар мають низький рівень ефективності, тому що даний рівень знаходиться на віддалених серверах. При дотриманні заходів безпеки операційного та

ресурсного шарів, ймовірність виникнення загроз безпеці на даному рівні проектування мінімальна.

Аналіз ресурсного шару ПЗ показав низький рівень наявності загроз безпеки. Це пов’язано з тим, що дані загрози безпеки орієнтовані на зовнішніх постачальників (наприклад, MySQL, Oracle), які постійно випускають оновлення «латок» безпеки. Таким чином, підтримка програмними продуктами зовнішніх постачальників в оновленому (up-to-date) стані мінімізує ризик виникнення атак. На додаток, наприклад, ресурс [4] показує можливі атаки, спрямовані на СУБД MySQL, що дає можливість системним адміністраторам перевірити систему на вразливість до найбільш поширених атак, спрямованих на дану СУБД. Даний вид атак характерний для Standalone-застосунків, де має місце можливість отримання інформації про СУБД, яка «зашита» в конфігурації програмного продукту. Існуючі на даний момент механізми обходу даної вразливості - використання зовнішніх Vault [139, 233] систем і зберігання СУБД в «хмарі», проте, даний механізм відрізняється дорожнечею використання і застосовності виключно для Web-застосунків.

Аналіз операційного шару показав наявність атак, пов’язаних з моніторингом мереж, брутфорсом паролів, а також DoS-атаки. Методи захисту від даних видів атак розглянуті в [152, 182, 198, 200] застосування яких дає підстави вважати, що ймовірність виникнення загроз на даному рівні прагне до 0.

Таким чином, проведені дослідження показали, що для Standalone-застосунків саме системний рівень є одним з найбільш вразливих. Саме на цьому рівні можливі злочинні дії, спрямовані на дискредитацію наступних послуг безпеки: конфіденційність, цілісність, непідробленість, справжність, захищеність від помилки.

1.3 Постановка науково-технічної проблеми

Незважаючи на позитивні результати існуючих досліджень в галузі інформаційних систем, аналіз відомих моделей та методів показав, що сьогодні не в повній мірі вирішено питання забезпечення безпеки байт-код орієнтованих програмних засобів в умовах кібератак, внаслідок чого значно знижується якість програмних продуктів (рис. 1.9).

У зв'язку з повсюдним поширенням комп'ютерних систем та технологій, збільшення попиту на інформаційні послуги, впровадження обчислювальних та інших засобів обробки даних у ключових сферах життя, актуальність використання програмного забезпечення зростає. З одного боку, зросла інтенсивність кібератак на послуги автентифікації та конфіденційності, з'являються нові методи і засоби злочинного впливу. Динамічне збільшення вимог до захищеності програмного забезпечення, що регламентуються законами України та міжнародними стандартами, стимулює адаптацію програмних засобів до цих змін протягом усього життєвого циклу розвитку програмного продукту. З іншого боку, існуючий стан теоретичного обґрунтування, синтезу і практичної реалізації підсистем забезпечення автентифікації та конфіденційності байт-код орієнтованих програмних засобів в умовах кібератак не дозволяє реалізувати забезпечення цих підвищених та нових вимог.

Таким чином, на сьогоднішній день в теорії і практиці забезпечення безпеки програмних засобів загострилося протиріччя між підвищенням вимог до безпеки програмного забезпечення, збільшенням кіберзагроз на послуги автентифікації та конфіденційності та недостатньою якістю сучасних моделей та методів забезпечення безпеки байт-код орієнтованих програмних засобів, які не в змозі забезпечити необхідні якісні характеристики.

Рис. 1.9. Основні існуючі протиріччя в галузі забезпечення безпеки байт-код орієнтованих програмних засобів. Наукова проблема

Подолати цю суперечність можна шляхом вирішення актуальної науково-практичної проблеми підвищення безпеки байт-код орієнтованого програмного коду в умовах кібератак на основі синтезу підсистеми забезпечення конфіденційності та автентичності програмних продуктів.

Для вирішення поставленої науково-технічної проблеми виникає необхідність у вирішенні таких взаємопов'язаних завдань (рис. 1.10):

– розроблення показника безпеки програмних засобів;

– розроблення методу перевірки логіко-смислової подібності програм;

– розроблення GERT-моделі процесу обфускації програмних модулів з використанням парадигми математичного апарату гамма-розподілу в якості ключового на всіх етапах моделювання процесу обфускації;

– розроблення критерію якості обфускації коду для багатопроєктного рішення на основі дослідження оцінки показників якості коду;

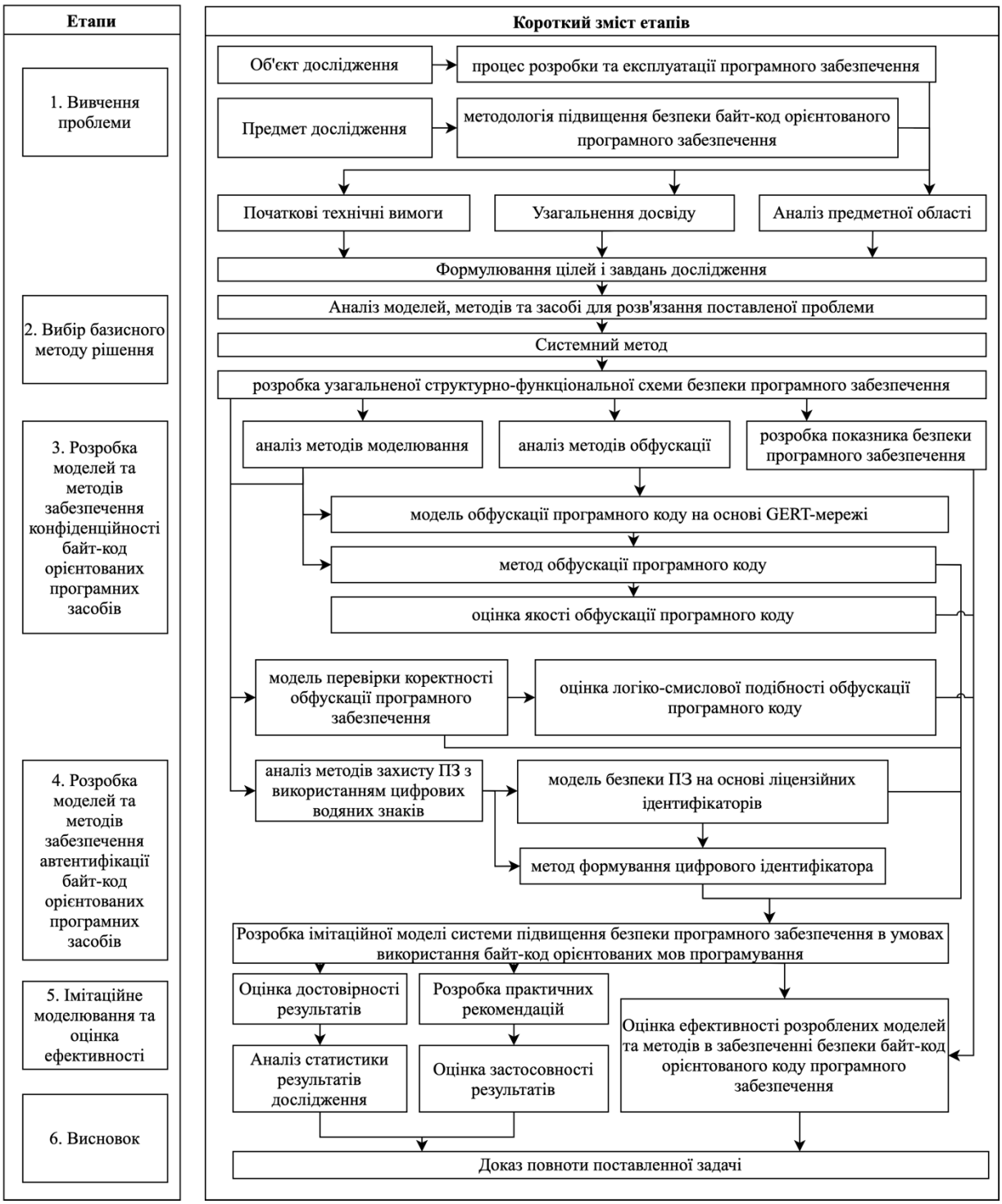


Рис. 1.10. Структурна схема дисертаційного дослідження

– розроблення методу обфускації програмних модулів для підвищення безпеки байт-код орієнтованого програмного забезпечення на основі розробленої GERT-моделі та критерію якості обфускації програмних модулів;

– розроблення моделі системи безпеки програмного забезпечення на основі математичного апарату моделювання GERT-мереж з використанням операцій безпечного переходу і кодування ліцензійних ідентифікаторів;

– розроблення методу формування цифрового ідентифікатора програмного забезпечення для захисту авторських прав;

– розроблення імітаційної моделі систем формування вимог до безпеки та захисту програмного забезпечення;

– обґрунтування достовірності одержаних результатів наукових досліджень;

– розроблення практичних рекомендації щодо застосування розробленого методу підвищення безпеки байт-код орієнтованого програмного забезпечення;

– впровадження розроблених моделей та методів підвищення безпеки байт-код орієнтованого програмного забезпечення в умовах кібератак.

Дисертаційні дослідження повинні здійснюватися на основі систематизації вже відомих підходів у предметній області безпеки інформаційних систем, розробки нових методів та засобів їх аналізу і проектування.

Висновки за розділом 1

1. У ході роботи було досліджено модель забезпечення безпеки програмного забезпечення. Аргументовано доказано необхідність розвитку даної моделі шляхом адаптації існуючих вимог до безпеки програмних засобів протягом усього життєвого циклу розробки програмного забезпечення.
2. Досліджено характеристики якості програмного забезпечення. Доказано, що для забезпечення захищеності необхідно підвищити якість таких послуг безпеки як: цілісність, автентифікація, конфіденційність,

управління доступом. Однак, показано, що покращення цих послуг безпеки може спричинити за собою погіршення інших показників якості програмного забезпечення: переносимості, супроводжуваності, продуктивності. Саме це підтверджує необхідність адаптації існуючих підходів захисту програмного забезпечення.

3. Сформовано універсальний показник безпеки програмних засобів, що відрізняється від відомих зменшенням фактору суб'єктивності вагових коефіцієнтів показника. Проведено розрахунки складових коефіцієнтів показника для Standalone і Web-застосунків. Показано необхідність підвищення безпеки саме Standalone типу застосунків. Також доказано вразливість системного рівня Standalone застосунків.

4. Виявлено протиріччя між підвищенням вимог до безпеки програмного забезпечення, збільшенням кіберзагроз на послуги автентифікації та конфіденційності та недостатньою якістю сучасних моделей та методів забезпечення безпеки байт-код-орієнтованих програмних засобів, які не в змозі забезпечити необхідні якісні характеристики. Зазначено, що подолати цю суперечність можна шляхом вирішення актуальної науково-практичної проблеми підвищення безпеки байткод-орієнтованого програмного коду в умовах кібератак на основі синтезу підсистеми забезпечення конфіденційності та автентичності програмних продуктів.

РОЗДІЛ 2. МОДЕЛЬ ПЕРЕВІРКИ ОБФУСКАЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ ТЕОРІЇ ПОДІБНИХ ФУНКЦІЙ

При проектуванні програмного забезпечення часто виникають ситуації, коли різні фрагменти програми виконують схожі дії з даними, що до них надходять. Така поведінка окремих фрагментів програм призводить не лише до труднощів розуміння логіки програми, але і до збільшення витрат на її подальшу розробку і супровід. Статистичні дослідження відкритих вихідних кодів вільно поширюваного програмного забезпечення на мовах C і Java, результати яких опубліковані в роботах [105, 169, 209], свідчать про те, що сумарний обсяг дубльованого коду в великих програмних проектах зазвичай становить від 7 до 23% вихідного коду. Крім труднощів, що виникають у розробників у зв'язку з підтримкою такого коду, дубльовані фрагменти вимагають надлишкових ресурсів, що витрачаються компілятором і системою програмування в цілому на компіляцію і зберігання об'єктного коду.

Дослідження показали, що пошук таких фрагментів-клонів, навіть в невеликих програмах, є досить трудомісткою операцією. Здійснення ж такого пошуку вручну на Enterprise проектах фактично неможливо.

Даний факт дозволив висунути припущення, що використання підходу формування подібних фрагментів для прикладного ПЗ дозволить підвищити рівень якості обфускованості коду. Це знаходить застосування в галузі комп'ютерної безпеки [78, 79, 117, 118, 138]. Під обфускацією розуміється таке перетворення вихідного тексту програми, при якому її функціональність зберігається, але отримання інформації про специфічні особливості алгоритмів і структур даних, що містяться в програмі, стає значно більш трудомістким завданням. Щоб конструювати обфускуючі перетворення

програм, необхідно мати засіб для доказу того, що обфускація коректна, тобто програма, отримана в результаті обфускуючого перетворення, подібна вихідній програмі. Крім того, складність завдання перевірки подібності програм може служити мірою якості обфускації програм: якщо вдається легко довести подібність двох різних обфускованих варіантів однієї і тієї ж програми, то це є ознакою того, що даний метод обфускації не володіє хорошою стійкістю.

Так, наприклад, можна замінити виклик процедури з різними значеннями її параметрів на N різних функцій. Це призведе до збільшення розміру програми і додаткову надмірність логіки, що ускладнить її аналіз.

2.1 Дослідження моделей та методів перевірки подібності програмного коду

Варто відзначити, що клас задач, в яких важливу роль відіграє ставлення подібності, не обмежується лише завданнями обфускації як зворотною стороною рефакторінгу [18, 32, 106, 132, 177, 208, 210, 211]. Так, необхідність перевірки фрагментів коду на подібність виникає при побудові оптимізуючих компіляторів як послідовних, так і паралельних програм [98, 163, 170, 202, 243].

Крім того, при розробці обфускуючих перетворень програм необхідно перевіряти, що функціональність коду, отриманого в результаті перетворень, не змінюється, тобто адаптований код подібний вихідному.

Також, в рамках обфускації коду може виступати процес розгортання гілок обчислень [232].

Алгоритми перевірки подібності програм також знаходять застосування в області комп’ютерної безпеки. Зокрема, потреба в них виникає при вирішенні завдання обфускації програм.

Одним з популярних і ефективних засобів пошуку шкідливого коду на сьогоднішній день є перевірка відповідності фрагментів підозрілого коду сигнатурам відомих шкідливих програм, що також робить задачу пошуку шкідливого програмного забезпечення залежною від ефективних алгоритмів перевірки подібності програм [113, 114, 237, 240]. Особливо це важливо для пошуку метаморфних вірусів, здатних модифікувати свій код в процесі реплікації і не маючих з цієї причини постійної сигнатури [44].

Для того, щоб створити засіб автоматичного пошуку клонів, необхідно в першу чергу вибрати відповідну математичну модель програм, в рамках якої можна було б не лише адекватно формалізувати відношення подібності програм, але також мати можливість ефективно розпізнавати це відношення.

Під програмою в даному випадку може розумітися не лише система команд на деякій мові програмування, а й об’єкт будь-якої обчислювальної моделі, наприклад, машина Тюрінга, машина Посту, кінцевий автомат, нормальний алгоритм і так далі. При цьому, в залежності від обраної моделі і поставленого завдання, визначається і відношення подібності, яке вивчається на об’єктах даної моделі. Найбільш природним для задач такого типу є ставлення функціональної подібності програм, при якому кожна програма розглядається як опис функції, перетворюючої набір вхідних даних в набір вихідних даних, а подібність двох заданих програм визначається як збіг реалізованих ними функцій. Однак з теореми Райса-Успенського [205] випливає, що функціональна подібність нерозв’язна в будь-якої моделі обчислень, в якій реалізовані всі обчислювані за Тюрінгом функції. Це означає, що для аналізу програм необхідно вибирати більш суворі, але розв’язні види подібності, які апроксимували б функціональну подібність. Фактично, саме аналіз і дослідження різних видів подібності на різних моделях обчислень лежить в основі завдань семантичного аналізу програм. Далі наведені приклади моделей програм і введених для них видів подібності; деякі із зазначених видів подібності виявилися розв’язними, в

окремих випадках показана можливість вирішення завдань перевірки подібності за поліноміальний час.

Однією з перших математичних моделей програм вважаються схеми програм Ляпунова [64, 65]. У шістдесяті роки починається швидкий розвиток програмованої обчислювальної техніки і мов програмування. Схеми Ляпунова відображали основні принципи програмування, що існували на той момент. Ці принципи фактично лягли в основу парадигми імперативного процедурного програмування. У формалізації схем Ляпунова, запропонованої А. П. Єршовим [37], схема являє собою граф з вузлами двох типів: вузли-перетворювачі, відповідні елементарним операторам, і вузли-розпізнавачі, відповідні логічним умовам або тестам. Вузли цього графа відображають інструкції зміни стану змінних (функціональні символи для операторів присвоювання) або перевірки деяких умов (предикатні символи логічних операторів) над змінними реальної програми. Дуги графа при цьому описують правила передачі управління між вузлами в залежності від значень предиката, приписаного вузлу-розпізнавачу. Перед виконанням схеми задається інтерпретація, яка наділяє сенсом функціональні і предикатні символи. Саме виконання схеми полягає в обході графа і накопиченні інформації про зміну станів. При цьому шлях, за яким відбувається обхід графа, визначається відповідно до інтерпретації.

Схеми Ляпунова вважаються подібними, якщо при будь-якій інтерпретації або обхід цих схем нескінченний, або накопичені в процесі обходу цих схем послідовності операційних символів збігаються. В роботі [96] був розроблений алгоритм перевірки подібності схем Ляпунова, тим самим була показана можливість розв’язання проблеми подібності для цієї моделі програм.

Теорія дискретних перетворювачів [15, 16, 58, 59, 60] передбачає наступну формалізацію поняття програми. Програма представляє собою систему з двох взаємодіючих автоматів. Перший з них описує потік

управління в програмі, в той час як другий описує семантику її операторів і предикатів. Фактично, другий автомат визначає клас інтерпретацій, для яких може розглядатися задача перевірки подібності програм. Виявилося, що в загальному випадку проблема подібності для дискретних перетворювачів нерозв'язна, але для деяких автоматів, які задають семантику операторів, алгоритми перевірки подібності дискретних перетворювачів можуть бути побудовані. Так, наприклад, в роботах [16, 61] встановлені необхідні і достатні умови можливості розв'язання проблеми подібності для випадків, коли семантика базових операторів програм визначається за допомогою півгруп.

В роботі [123] вперше були описані схеми програм, орієнтовані на функціональну парадигму програмування – рекурсивні схеми. Більш детально вони були вивчені в роботах [21, 101, 137, 195, 227]. В роботі [137] був побудований підклас рекурсивних схем, за виразними можливостями подібний схемами Ляпунова. Для цього підкласу була показана можливість розв'язання проблеми подібності. Крім того, проблема подібності виявилася розв'язною також для більш виразних класів рекурсивних схем [101, 137, 62] і дослідження цього завдання тривають [72, 73, 74]. Але в загальному випадку проблема подібності для рекурсивних схем програм виявилася нерозв'язною [134].

Алгебраїчні моделі програм, вперше введені в роботі [66], узагальнили ідеї моделі дискретних перетворювачів і схем Ляпунова. Особливістю алгебраїчних моделей програм по відношенню до задачі перевірки подібності можна вважати той факт, що вони дозволяють використовувати семантичні властивості складових програми при вирішенні зазначеної задачі [67, 68, 69, 71]. Центральним завданням при вивченні даної моделі була побудова системи подібних перетворень, що неминуче призвело до появи інтересу до задачі перевірки подібності в цій моделі. Це завдання було успішно вирішено в роботі [70].

Динамічні моделі програм, введені в роботі [41], описують семантику програм на мові динамічної логіки. В роботі [42] описана методика дозволу подібності в цій моделі.

Модель програм, використана в даній роботі, була вперше представлена в 1967 році в роботі [37]. Дана модель, названа моделлю стандартних схем програм, є подальшим розвитком концепції схем програм Ляпунова за допомогою використання мови першого порядку для опису операторів і логічних умов і теоретико-графового представлення схем, що значно підвищило рівень деталізації і спростило розуміння вихідної моделі. Надалі методи, запропоновані для стандартних схем програм, часто використовувалися в теорії статичного аналізу програм [53]. Функціональна подібність для даної моделі виявилася нерозв'язною [172, 194]. Однак для інших видів подібності, апроксимуючих функціональну, була показана їх розв'язність. Так, наприклад, логіко-смислова подібність, введена в роботі [46], виявилася не лише розв'язною [7], але для неї вдалося побудувати алгоритми, що вимагають поліноміального щодо розмірів програм часу [53, 80].

Таким чином, з теореми Райса-Успенського зовсім не випливає неможливість побудови достатніх умов функціональної подібності програм, що ефективно перевіряються. Теорема швидше стверджує, що навіть якщо такі умови будуть побудовані, вони будуть лише достатніми, але не будуть необхідними.

Стандартні схеми програм, введені в роботі [37], дозволили значно спростити розуміння моделі програм Ляпунова завдяки графовому представленню.

Послідовна імперативна програма розглядається як розмічений орієнтований граф p. Кожному вузлу цього графа приписана деяка логічна умова – предикат. З кожного вузла, за винятком однієї, виходить дві дуги, при цьому одна з дуг позначена символом 0, а інша – символом 1. Крім того,

кожній дузі графа приписана ще одна позначка, що представляє собою заміну, тобто відображення, яке ставить у відповідність кожній змінній програми деяка лексема. Окремо виділені два вузла – вхід і вихід програми, при цьому обидві дуги, які виходять із вхідних вузлів, спрямовані в один і той же вузол, а вихідний вузол – єдиний з вузлів програми, з якого не виходить жодна дуга. Вважається, що для кожного вузла програми існує деякий шлях, який проходить через цей вузол з входу програми в її вихід. Семантично вузли програми відповідають операторам розгалуження або умовним операторам програми. Дуги програми при цьому описують лінійні ділянки, що представляють собою кінцеві списки операторів присвоювання, що змінюють значення змінних, між операторами розгалуження. При цьому заміна кожної конкретної лінійної ділянки будується відповідно до того ефекту, який надає сукупне застосування всіх операторів присвоювання на поточний стан даних.

Програма здійснює обчислення над абстрактними даними, які є значеннями змінних цієї програми. Саме обчислення полягає в обході графа програми і починається з вхідних вузлів графа. Для того, щоб такий обхід був детермінований, необхідно наділити сенсом не лише множину вхідних змінних, а й кожен перетворювач, якому зіставляється деяка функція, і кожен розпізнавач, якому зіставляється деяка логічна функція. Все це задається за допомогою інтерпретації, яка також дозволяє визначити, яка саме з дуг з позначками 0 і 1, що виходять з поточного розпізнавача, повинна бути виконана. Залежно від того, по дугам з якими мітками здійснювався обхід, визначається шлях в програмі. Таким чином, інтерпретація наділяє семантикою всі компоненти програми. Обчислення має на увазі запам’ятовування послідовності станів даних відповідно до перетворень, що відбуваються над змінними під час обходу. Обхід графа триває до тих пір, поки не буде досягнута вихідний вузол. Отриманий до цього моменту стан пам’яті і буде результатом обчислення програми.

Логіко-смислова подібність, вперше введена в роботі [46], не використовує в своєму визначенні інтерпретації функцій і предикатів програми, тому на неї не поширюється результат статті [153] про нерозв'язність невироджених інтерпретаційних відносин подібності на стандартних схемах програм. Дана особливість логіко-смислової подібності стандартних схем програм робить це відношення привабливим для використання при вирішенні деяких завдань статичного аналізу програм.

Перший алгоритм розпізнавання логіко-смислової подібності був представлений в роботі [46], де таке положення було введено. Однак цей алгоритм був досить складним. Пізніше був запропонований алгоритм [7], який зводив задачу розпізнавання логіко-смислової подібності до задачі розпізнавання подібності в класі двохстрічкових автоматів. Цей алгоритм мав експонентну складність за часом виконання.

Перший поліноміальний алгоритм був представлений в роботі [80] і мав оцінку складності $O(n^7)$, де n - максимальний з розмірів вихідних фрагментів програм. Цей алгоритм складається з декількох кроків. Перший крок передбачає перетворення вихідних фрагментів програм p_1, p_2 до наведеного виду, тобто до таких фрагментів, в яких кожний вузол досяжний з входу, з кожного вузла досяжний вихід, кожен предикат містить звернення тільки до змінних, а не до більш складних лексем над ними. Для здійснення цього кроку алгоритм надає вісім подібних перетворень. У тому випадку, якщо побудовані фрагменти p'_1 і p'_2 не ізоморфні, алгоритм оголошує вихідні фрагменти не логіко-смислово подібними. Інакше за наявними перетвореним фрагментами будується граф p, який по суті є декартовим добутком графів p'_1 і p'_2, після чого до цього графу застосовується одне з восьми правил перетворень, зване заміною лексем; за допомогою цього правила будується стаціонарна розмітка графа p. Якщо ж побудова правильної стаціонарної розмітки з використанням правила заміни лексем неможлива, то вихідні фрагменти програм p_1, p_2 визнаються не логіко-смислово подібними.

Доведення коректності цього алгоритму спирається на коректність алгоритму Берда [110] розпізнавання подібності двохстрічкових автоматів.

2.2 Завдання перевірки логіко-смислової подібності програм

Було проаналізовано алгоритм перевірки логіко-смислової подібності стандартних послідовних схем програм, заснований на пошуку найбільш схожих з точки зору лексемної історії шляхів в програмі. Цей алгоритм використовує в якості опорної структури граф спільних обчислень або, інакше, граф узгоджених маршрутів програм. Розмітка цього графу відповідає перетворенням лексемної історій змінних на всіх можливих шляхів програми, а сам алгоритм, в свою чергу, зводиться до обчислення точних нижніх меж на множині замін. Ітеративна процедура, що здійснює це обчислення, і є процедурою побудови розмітки графа.

2.2.1 Граф спільних обчислень

Нехай задані дві програми $p' = \langle X, Y', V', v_{in}', v_{out}', B', \to', L_0' \rangle$ і $p'' = \langle X, Y'', V'', v_{in}'', v_{out}'', B'', \to'', L_0'' \rangle$. Вважаємо, що зазначені дві програми мають однакову множину вхідних змінних X і непересічні множини внутрішніх змінних $Y' \cap Y'' = \varnothing$.

Графом спільних обчислень $\Gamma_{p_1, p_2} = \langle v, w_0, \mapsto, L_0 \rangle$ цих двох програм будемо називати граф з наступними компонентами:

– $V = V' \times V''$ - множина вузлів графа. Сюди входять різноманітні пари $w = (v', v'')$ точок вихідних програм p' і p''. Кожному вузлу графа спільних обчислень приписана пара атомарних формул $(B'(v'), B''(v''))$;

– $w_0 = (v_{in}', v_{in}'')$ – виділений кореневий вузол, йому відповідає пара вхідних вузлів вихідних програм;

– $\rightarrow \subseteq V \times \{0,1\} \times Заміна(Y'\bigcup Y'', Y'\bigcup Y'', F) \times V$ - відношення переходів, що визначає дуги (переходи) графа. Суворе визначення цього відношення наведено нижче;

– $L_0 = L_{0'} \bigcup L_{0''}$ – ініціалізуюча заміна графа спільних обчислень програм.

Для стислості будемо замість запису $(v', v'', d, Q, u', u'') \in \mapsto$ записувати відношення переходів природніше: $(v', v'') \overset{d,Q}{\mapsto} (u', u'')$.

Відношення $\mapsto$ визначається наступним чином: для кожної пари точок (v', v''), (u', u'') графу $\Gamma_{p', p''}$ і заміни $Q, Q \in Заміна(Y'\bigcup Y'', Y'\bigcup Y'', F)$, має бути виконано:

$$(v', v'') \overset{d,Q}{\mapsto} (u', u'') \Leftrightarrow \exists d \in \{0,1\} : v' \overset{d,Q'}{\rightarrow} u' \& v'' \overset{d,Q''}{\rightarrow} u'' \& Q = Q' \bigcup Q'' \qquad (2.1)$$

При цьому, якщо вірно $(v', v'') \overset{\delta,Q}{\mapsto} (u', u'')$, будемо казати, що вузол (u', u'') - спадкоємець вузла (v', v') в графі Γ_{p_1, p_2} і будемо позначати це наступним чином: $(u', u'') = succ((v', v''), \delta)$.

Введемо поняття шляху в графі спільних обчислень. Будь-яку послідовність дуг

$$шлях = (v'_{in}, v''_{in}) \overset{d_0,Q_0}{\mapsto} (v'_1, v''_1) \overset{d_1,Q_1}{\mapsto} \ldots \overset{d_{n-1},Q_{n-1}}{\mapsto} (v'_n, v''_n) \overset{d_n,Q_n}{\mapsto} (v'_{n+1}, v''_{n+1})$$

будемо називати шляхом у графі спільних обчислень Γ_{p_1, p_2}. Для кожного шляху *шлях* у графі Γ_{p_1, p_2} введемо поняття заміни шляху $Q_{шлях}$, що являє собою композицію ініціалізуючої заміни L_0 і всіх замін, що приписані дугам цього шляху: $Q_{шлях} = L_0 Q_1 Q_2 \ldots Q_{n-1} Q_n$.

Два шляхи tr' і tr'' в програмах p' і p'' відповідно будемо називати логічно узгодженими або просто узгодженими, якщо вірно, що

$$tr' = v'_{in} \overset{d_0,Q'_0}{\rightarrow} v'_1 \overset{d_1,Q'_1}{\rightarrow} \ldots \overset{d_n,Q'_n}{\rightarrow} v'_{n+1}$$

і

$$tr'' = v''_{in} \overset{d_0,Q''_0}{\rightarrow} v''_1 \overset{d_1,Q''_1}{\rightarrow} \ldots \overset{d_n,Q''_n}{\rightarrow} v''_{n+1}$$

Вектор $\tilde{d} = (d_0, d_1, \ldots, d_n)$ будемо називати загальним вектором узгоджених маршрутів tr', tr''. Шлях в графі спільних обчислень, що проходить по вузлам:

$$шлях = (v'_{in}, v''_{in}) \overset{d_0, Q'_0 \cup Q''_0}{\mapsto} (v'_1, v''_1) \overset{d_1, Q'_1 \cup Q''_1}{\mapsto} \ldots \overset{d_n, Q'_n \cup Q''_n}{\mapsto} (v'_{n+1}, v''_{n+1})$$

будемо називати відповідним парі узгоджених маршрутів tr' і tr'' за вектором d.

Для всіх вузлів $w, w \in V$, графа спільних обчислень Γ_{p_1, p_2} використовуватиме позначення $Шлях(w)$ для множини всіх шляхів графа, що ведуть із вхідного вузла w_0 у вузол w.

Граф спільних обчислень програм p', p'' – граф логічно-узгоджених маршрутів цих програм.

Із введених вище визначень графа спільних обчислень, узгоджених маршрутів і відповідних їм шляхів, а також з умови (2.1) випливає справедливість наступних двох лем.

Лема 1. Нехай tr' і tr'' – узгоджені маршрути в програмах p' і p'' відповідно. Тоді в графі спільних обчислень Γ_{p_1, p_2} існує єдиний шлях, відповідний парі маршрутів tr', tr''.

Лема 2. Нехай $шлях$ – шлях в графі спільних обчислень Γ_{p_1, p_2} програм p' і p''. Тоді у вхідних програмах існує єдина пара маршрутів tr', tr'', якій відповідає цей шлях.

Граф спільних обчислень назвемо коректним, якщо в ньому

– із вхідних вузлів недосяжні вузли виду (v'_{out}, v''_i) і вузли виду (v'_j, v''_{out}), де $v''_i \neq v''_{out}$ і $v'_j \neq v'_{out}$ відповідно;

– із кожного вузла графа досяжний вузол (v'_{out}, v''_{out}).

Теорема 1. Дві програми $p' = \langle X, Y', V', v_{in}', v_{out}', B', \to', L_0' \rangle$ і $p'' = \langle X, Y'', V'', v_{in}'', v_{out}'', B'', \to'', L_0'' \rangle$ логіко-смислової подібні тоді і тільки тоді, коли їх граф спільних обчислень коректний і для будь-якого шляху в ньому

$$шлях = \left(v'_{in}, v''_{in}\right) \overset{d_0, Q'_0 \cup Q''_0}{\mapsto} \left(v'_1, v''_1\right) \overset{d_1, Q'_1 \cup Q''_1}{\mapsto} \ldots \overset{d_n, Q'_n \cup Q''_n}{\mapsto} \left(v'_{n+1}, v''_{n+1}\right)$$

виконано рівність $B'\left(v'_{n+1}\right)Q_{tr'} = B''\left(v''_{n+1}\right)Q_{tr''}$, де $Q_{tr'}$ і $Q_{tr''}$ – заміни маршрутів tr' і tr'' відповідно таких, що шлях з ними погоджено.

Доведення теореми 1. Справедливість $tr'' = v''_{in} \overset{d_0, Q''_0}{\rightarrow} v''_1 \overset{d_1, Q''_1}{\rightarrow} \ldots \overset{d_n, Q''_n}{\rightarrow} v''_{n+1}$ прямого твердження можна показати, використовуючи визначення логіко-смислової подібності програм. З того, що програми p_1 і p_2 логіко-смислово подібні, випливає, що для кожного маршруту

$$tr' = v'_{in} \overset{d_0, Q'_0}{\rightarrow} v'_1 \overset{d_1, Q'_1}{\rightarrow} \ldots \overset{d_n, Q'_n}{\rightarrow} v'_{n+1}$$

в програмі p_1 знайдеться єдиний узгоджений з нею маршрут в програмі p_2 і за лемою 1 цій парі буде відповідати єдиний шлях *шлях* в графі Γ_{p_1, p_2}. Коректність графа спільних обчислень в даному випадку слідує з узгодженості маршрутів tr' і tr''.

Нехай для вузлів v'_{n+1} і v''_{n+1} виконано $A' = B'\left(v'_{n+1}\right)$ і $A'' = B''\left(v''_{n+1}\right)$ відповідно. Останніми парами в відповідних цим маршрутам логіко-смислових історіях будуть пари $\left(A'Q_{tr'}, d_n\right)$ і $\left(A''Q_{tr''}, d_n\right)$. Оскільки програми p_1 і p_2 логіко-смислової подібні, то збігаються логіко-смислові історії узгоджених маршрутів цих програм, а це означає, що збігаються також і їх часткові логіко-смислові історії, тобто $A'Q_{tr'} = A''Q_{tr''}$.

Справедливість зворотного твердження випливає з леми 2, тобто з єдиності пари узгоджених маршрутів в програмах p_1 і p_2, відповідних заданому шляху в графі спільних обчислень. Тоді якщо для всіх шляхів графа спільних обчислень в вузлу $\left(v'_{n+1}, v''_{n+1}\right)$, якій приписані атоми $A' = B'\left(v'_{n+1}\right)$ і $A'' = B''\left(v''_{n+1}\right)$, вірно $A'Q_{tr'} = A''Q_{tr''}$, то для пари узгоджених шляхів в програмах p_1 і p_2 це означатиме збіг їх логіко-смислових історій. Неважко помітити, що це ж виконано і для всіх шляхів, що ведуть у вихідні точки програм, тобто повні логіко-смислові історії всіх маршрутів першої програми будуть

збігатися з точністю до перейменування з повними логіко-смисловими історіями усіх шляхів другої програми, звідки слідує їх логіко-смислова подібність.

Таким чином, перевірка логіко-смислової подібності може бути зведена до аналізу пар узгоджених маршрутів вихідних програм, або аналізу графа їх спільних обчислень. Але в графі спільних обчислень можуть існувати вузли, в які веде нескінченно багато шляхів, що ускладнює проведення перевірки подібності на підставі однієї лише теореми 1. Тому нам буде потрібна наступна допоміжна лема.

Лема 3. Нехай Q_1 і Q_2, $Q_1, Q_2 \in Заміна(X, Y, F)$, – деякі заміни, а атоми A' і A'', $A', A'' \in Atom(Y, F)$ – умовні вирази. Тоді:

$$\begin{cases} A'Q_1 = A''Q_1 \\ A'Q_2 = A''Q_2 \end{cases} \Leftrightarrow A'(Q_1 \downarrow Q_2) = A''(Q_1 \downarrow Q_2).$$

Доведення. Доведемо спершу пряме твердження

$$\begin{cases} A'Q_1 = A''Q_1 \\ A'Q_2 = A''Q_2 \end{cases} \Rightarrow A'(Q_1 \downarrow Q_2) = A''(Q_1 \downarrow Q_2). \qquad (2.2)$$

Для доведення твердження скористаємося теоремою 1 про дистрибутивність композиції відносно операції деуніфікації, помітивши, що всі властивості композиції замін можуть бути перенесені на операцію застосування замін до предикату. Тоді $A'(Q_1 \downarrow Q_2) = A'Q_1 \downarrow A'Q_2 = A''Q_1 \downarrow A''Q_2 = A''(Q_1 \downarrow Q_2)$, отже, твердження (2.2) вірно.

Покажемо тепер справедливість зворотного твердження:

$$\begin{cases} A'Q_1 = A''Q_1 \\ A'Q_2 = A''Q_2 \end{cases} \Leftarrow A'(Q_1 \downarrow Q_2) = A''(Q_1 \downarrow Q_2). \qquad (2.3)$$

Нехай $n = Q_1 \downarrow Q_2$. Тоді за визначенням точної нижньої межі для деяких замін p_1, p_2 справедливі рівності $Q_1 = np_1$ і $Q_2 = np_2$. Отже, вірний наступний ланцюжок висновків:

$$A'(Q_1 \downarrow Q_2) = A''(Q_1 \downarrow Q_2) \Rightarrow A'n = A''n \Rightarrow A'\eta p_1 = A''\eta p_1 \Rightarrow A'Q_1 = A''Q_1.$$

Аналогічний ланцюжок може бути побудований і для заміни Q_2, що завершує доказ леми.

Доведена лема 3 дозволяє переформулювати необхідну і достатню умову логіко-смислової подібності програм, наведену в теоремі 2, в такий спосіб:

Теорема 2. Дві програми p' і p'' логіко-смислової подібні тоді і лише тоді, коли граф їх спільних обчислень коректний і для кожної його вузла $w=(v',v'')$ виконано рівність $B'(v')Q_w = B''(v'')Q_w$, де $Q_w = \downarrow_{шлях \in Шлях(w)} Q_{шлях}$.

Доведення теореми 2. Справедливість цього твердження випливає з теореми 1, леми 3 і облаштування графу спільних обчислень програм.

Наведена вище теорема зводить задачу про перевірку подібності програм до обчислення точних нижніх меж замін, відповідних всіляким шлязам в графі спільних обчислень, ведучим в кожну з вузлів w графу. Далі наведена процедура глобальної розмітки графа, що дозволяє вирішувати цю задачу.

2.2.2 Процедура розмітки графу спільних обчислень

В даному розділі описано алгоритм розмітки графу логічно-узгоджених маршрутів, який, відповідно до теореми 2, послідовно будує наближення зверху до шуканої точної нижньої межі замін, відповідних шляхам в графі в кожний конкретний вузол. Алгоритм працює з графом $\Gamma_{p',p''}$. Не порушуючи спільності міркувань, будемо вважати, що граф $\Gamma_{p',p''}$ коректний, тому що в іншому випадку, у відповідності до теореми 3, його аналіз не потрібен.

Перейменування змінних. Перед початком роботи здійснюється перейменування входжень всіх змінних з множини X в першу програму змінними $\{x'_1, x'_2, \ldots, x'_n\}$, відмінними від змінних множини *X*. Також здійснюється перейменування входжень всіх змінних з множини *X* в другу програму змінними $\{x''_1, x''_2, \ldots, x''_n\}$, також відмінними від змінних множини

X. Домовимося позначати отримані в результаті зазначених перейменувань змінних програми p_1 і p_2 відповідно.

Граф спільних обчислень. Для програм p_1, p_2 будується їх граф спільних обчислень Γ_{p_1,p_2}.

Початкова розмітка графа. Кореневій вузли $w_0 = (v'_{in}, v''_{in})$ приписується заміна $n_{w_0} = \{x'_1/x_1, \ldots, x'_n/x_n, x''_1/x_1, \ldots, x''_n/x_n\}$, а всі інші вузли позначаються максимальними в решітці замін уявною заміною τ: $\eta_w = \tau$ для всіх $w \neq w_0$. Крім того, кореневому вузли присвоюється позначка *.

Ітерації алгоритму. До тих пір, поки в графі Γ_{p_1,p_2} є хоча б один вузол, позначена символом "*" (активний вузол), алгоритм довільним чином вибирає один з таких вузлів (v', v''). Припустимо, що обраному вузлу приписана деяка заміна $n_{(v',v'')}$. Тоді символ "*" знімається з вузла (v', v''), і для кожної дуги $(v', v'') \overset{d,Q}{\mapsto} \Rightarrow (u', u'')$, що веде з вузла (v', v'') в деякий вузол $(u', u'') = succ((v', v''), \delta)$ графу спільних обчислень, виконуються наступні дії:

1) обчислюється заміна $n'_{(u',u'')} = Qn_{(v',v'')} \downarrow n_{(u',u'')}$, де $n_{(u',u'')}$ – заміна, якою позначено вузол (u', u'');

2) якщо $n_{(u',u'')} \neq n'_{(u',u'')}$, то вузлу (u', u'') приписується позначка $n'_{(u',u'')}$. При цьому вузол (u', u'') позначається символом "*". Якщо ж $n_{(u',u'')} = n'_{(u',u'')}$, то заміна не відбувається.

Ці дії повторюються до тих пір, поки в графі Γ_{p_1,p_2} існують активні вузли, помічені "*".

Правило верифікації. Здійснюється перевірка наступної умови: чи існує в графі Γ_{p_1,p_2} хоча б один вузол (u', u''), позначена заміною $n_{(u',u'')}$ така, що $A_{u'} n_{(u',u'')} \neq A_{u''} n_{(u',u'')}$. У тому випадку, якщо такий вузол існує, програми визнаються не логіко-смислової подібними. Якщо ж таких вузлів не виявлено, то програми логіко-смислової подібні.

Далі будуть наведені твердження, що обґрунтовують коректність алгоритму.

Домовимося через $n_v^{[n]}$ позначати заміну приписаних вузлів $v=(v',v'')$ на n-му кроці алгоритму. Тоді справедливе наступне твердження:

Лема 4. Для будь-якого вузла $v=(v',v'')$ і для будь-якого кроку n роботи алгоритму існує таке підмножина P множини $Шлях(v)$, що для заміни $n_v^{[n]}$, обчисленої на етапі n для вузла v, виконано рівність:

$$n_v^{[n]} = \downarrow_{шлях \in P} Q_{шлях}$$

Доведення. Доведемо твердження індукцією за номером кроку алгоритму.

Базис. При $n=0$ твердження справедливо, тому що всім вузлам графа спільних обчислень приписана заміна $n_v^{[n]}=\tau$, для якої виконано рівність $\tau = \downarrow_{шлях \in Q} Q_{шлях}$.

Індуктивний перехід. Нехай твердження справедливо для деякого кроку n. Покажемо, що воно вірне і для $n+1$. Нехай u – вузол-попередник вузла v в графі спільних обчислень, $v=succ(u,d)$, і нехай дузі, що йде з u в v, приписана заміна Q, тобто в графі спільних обчислень знайдеться шлях $шлях \overset{d,Q}{\mapsto} v$. Зауважимо, що згідно з описом алгоритму в цьому випадку має місце рівність $n_v^{[n+1]} = n_v^{[n]} \downarrow Q n_u^{[n]}$.

За індуктивним припущенням, існують множини $P_u \subseteq Шлях(u)$ і $P_v \subseteq Шлях(v)$, для яких $n_v^{[n]} = \downarrow_{шлях \in P_v} Q_{шлях}$ і $n_u^{[n]} = \downarrow_{шлях \in P_u} Q_{шлях}$. Розглянемо множину $P'_v = P_v \cup \left\{ шлях \overset{d,Q}{\mapsto} : шлях \in P_u \right\}$. Покажемо, що ця множина і є шуканою. Для цього можна скористатися законом лівої дистрибутивності деуніфікації замін відносно операції композиції, встановленим в теоремі 1:

$$\downarrow_{шлях \in P'_v} Q_{шлях} = \left(\downarrow_{шлях \in P_v} Q_{шлях} \right) \downarrow \left(\downarrow_{шлях \in P_u} Q Q_{шлях} \right) =$$
$$\left(\downarrow_{шлях \in P_v} Q_{шлях} \right) \downarrow \left(\downarrow_{шлях \in P_u} Q_{шлях} \right) = n_v^{[n]} \downarrow Q n_u^{[n]}$$

Отже, множина P'_v і є шуканою.

Введемо ще одне позначення. Для кожного вузла v будемо позначати $In(v)$ множину всіх вузлів u таких, що $v = succ(u,d)$. Умовимося також заміну, що приписана переходу $u \overset{d,Q}{\mapsto} v$, позначати Q_{uv}. Тоді з попередньої леми слідує:

Лема 5. Для будь-якого вузла v

$$\downarrow_{шлях \in Шлях(v)} Q_{шлях} = \downarrow_{u \in In(v)} Q_{uv} \left(\downarrow_{шлях \in Шлях(u)} Q_{шлях} \right).$$

Скористаємося записом $Q(v)$ для позначення точної нижньої межі композиції замін, обчислених за всіма шляхами з множини $Шлях(v)$ у графі спільних обчислень, тобто $Q(v) = \downarrow_{шлях \in Шлях(v)} Q_{шлях}$.

Як показує лема 5, множина замін $\{Q(v), v \in V\}$, де V – множина вузлів графу спільних обчислень, є рішенням системи рівнянь

$$\bigcup_{v \in V \setminus entry} \left[X_v = \downarrow_{u \in In(v)} Q_{uv} X_u \right] \cup \left(X_{entry} = L \right), \qquad (2.4)$$

де L – заміна констант першого кроку алгоритму.

Лема 6. Припустимо, що на певному кроці з номером N алгоритм завершив свою роботу. Тоді $\left\{ n_v^{[N]} : v \in V \right\}$ - рішення системи рівнянь (2.4).

Доведення. Оскільки за умовою леми на N-му кроці роботи алгоритму не залишилося активних вузлів, рівність $n_v^{[N]} = n_v^{[N-1]}$ виконується для всіх вузлів графу спільних обчислень.

Розглянемо довільний вузол v графу спільних обчислень програм. Нехай для цього вузла останнім кроком, на якому відбувалися зміни приписаної їй заміни був крок N_1, тобто N_1 – найменший номер, для якого $n_v^{[N_1]} = n_v^{[N_1+1]} = \ldots = n_v^{[N]}$. Як вже було показано вище, заміна $n_v^{[N_1]}$ може бути подана в вигляді $\downarrow_{u \in In(v)} Q_{uv} n_u^{[N_1-1]}$. Але тоді, в силу вибору кроку N_1, для будь-якого номера i такого, що $i \geq N_1$, виконано $Q_v^{[i]} = \downarrow_{u \in In(v)} Q_{uv} n_u^{[i]}$. А оскільки $N_1 \leq N$,

то з цієї рівності слідує $n_v^{[N]} = \downarrow_{u \in In(v)} Q_{uv} n_u^{[N]}$ для будь-якого вузла v графу спільних обчислень програм. Таким чином, $\left\{ n_v^{[N]} \right\}$ – рішення системи (2.4).

Лема 7. Припустимо, що алгоритм зупинився на кроці N. Тоді для будь-якого вузла v графу спільних обчислень вірна рівність

$$n_v^{[N]} = \downarrow_{шлях \in Шлях(v)} Q_{шлях}.$$

Доведення. Припустимо, що існує такий вузол v, для якої

$$n_v^{[N]} \neq \downarrow_{шлях \in Шлях(v)} Q_{шлях}. \qquad (2.5)$$

Тоді існує такий шлях $шлях'_v$ з вхідного вузла графу спільних обчислень в вузол v, що $n_v^{[N]} \downarrow Q_{шлях'_v} \neq n_v^{[N]}$. Оберемо найкоротший з таких шляхів $шлях'_v$ по всім вузлам, що задовольняють співвідношенню (2.5). Розглянемо вузол u_0 такий, що він є попередником вузла v у шляху $шлях'_v$: $шлях'_v = шлях_{u_0} \overset{dQ_{u_0 v}}{\rightarrow} v$. Шлях $шлях_{u_0}$ не може бути найкоротшим з розглянутих шляхів, так як тоді цей шлях був б коротшим за $шлях'_v$, що протирічить вибору $шлях'_v$. Отже, для нього не виконана умова (2.5), тобто $n_{u_0}^{[N]} = n_{u_0}^{[N]} \downarrow Q_{шлях_{u_0}}$. Скориставшись лемою 6, отримуємо наступний ланцюжок рівностей:

$$n_v^{[N]} = \downarrow_{u \in In(v)} Q_{uv} n_u^{[N]} = \left(\downarrow_{\substack{u \in In(v) \\ u \neq u_0}} Q_{uv} n_u^{[N]} \right) \downarrow Q_{u_0 v} n_{u_0}^{[N]} = \left(\downarrow_{\substack{u \in In(v) \\ u \neq u_0}} Q_{uv} n_u^{[N]} \right) \downarrow \left(Q_{u_0 v} \left(n_{u_0}^{[N]} \downarrow Q_{шлях_{u_0}} \right) \right) =$$

$$= \left(\downarrow_{\substack{u \in In(v) \\ u \neq u_0}} Q_{uv} n_u^{[N]} \right) \downarrow Q_{u_0 v} n_{u_0, v} \downarrow Q_{u_0 v} Q_{шлях_{u_0}} = \left(\downarrow_{u \in In(v)} Q_{uv} n_u^{[N]} \right) \downarrow Q_{шлях'_v} = n_v^{[N]} \downarrow n_{шлях'_v},$$

що протирічить зробленому раніше припущенню $Q_v^{[N]} \neq Q_v^{[N]} \downarrow Q_{шлях'_v}$.

Скористаємося наведеними вище лемами для обґрунтування коректності алгоритму перевірки логіко-смислової подібності програм.

Теорема 3. Для будь-яких двох програм p_1 і p_2 таких, що їх граф спільних обчислень коректний, алгоритм побудови розмітки графу спільних обчислень завжди завершує свою роботу.

Доведення теореми 3. Покажемо, що обчислення нової заміни для кожного вузла не може повторюватися безліч разів. Розглянемо вузол графа спільних обчислень v. Припустимо, що в ній зміна приписаної заміни відбувається нескінченно довго. Нехай на i-му кроці роботи алгоритму їй була приписана деяка заміна n_v, а на кроці з номером $i+1$ був здійснений перехід з вузла u з приписаною їй заміною n_u по дузі з під-заміною Q в вузол v. За описом алгоритму в вузла v повинна бути обчислена точна нижня межа замін n_v і $n_u Q$. З припущення про те, що зміна приписаної цьому вузлі заміни відбувається нескінченно, слідує, що $n'_v \neq n_v \downarrow n_u Q$, але тоді $n'_v < n_v$.

Як вже було сказано, решітка замін $\left(Заміна^{\sim}(X,Y,F),\leq\right)$ має властивість обриву спадаючих ланцюгів, тобто для будь-якої заміни n довжина будь-якого ланцюга між класами $n^{\sim}$ і $\varepsilon^{\sim}$ скінченна. Але тоді зміна довжин замін, приписаних вузла v, не може відбуватися нескінченно довго, а значить, через кінцеву кількість кроків в кожному вузлу графу спільних обчислень заміна, позначає цей вузол, перестає змінюватися.

Теорема 4. Для двох програм p_1 і p_2, чий граф спільних обчислень коректний, зупинка алгоритму перевірки логіко-смислової подібності програм з позитивним результатом відбувається тоді і лише тоді, коли програми p_1 і p_2 – логіко-смислово подібні.

Доведення теореми 4. Зупинка алгоритму з позитивним результатом означає, що в графі спільних обчислень більше не залишилося активних вузлів (вузлів, мають позначку "*") і для кожного вузла, якій приписана пара атомів (A', A''), на одному з кроків алгоритму була обчислена заміна $n_v^{[N]}$ така, що $A' n_v^{[N]} = A'' n_v^{[N]}$. Ця заміна, за лемою 7, являє собою точну нижню межу замін всіх шляхів, що ведуть в вузел v, то p_1 і p_2 логіко-смислово подібні.

Припустимо тепер, що для двох логіко-смислово подібних програм p_1 і p_2 алгоритм зупинився з негативним результатом. Нехай стабілізація замін сталася на кроці з номером n. З опису алгоритму випливає, що для деякого

вузла v з приписаною їй парою атомів $(A', A")$ є $n_v^{[n]} = \downarrow_{шлях \in P} Q_{шлях}$. Це, відповідно до теореми 2, означає, що для будь-якого шляху *шлях* в графі спільних обчислень з вхідного вузла в вузол v виконано рівність $A'Q_{шлях} = A"Q_{шлях}$. Тоді по теоремі 2 програми і заміною $n^{[n]}$ виконано $A'n_v^{[n]} \neq A"n_v^{[n]}$. За лемою 4, $n_v^{[n]} = \downarrow_{шлях \in P} Q_{шлях}$, де P – це деяка підмножина шляхів, що ведуть в вузол v. Тоді, з огляду на затвердження леми 3, існує шлях *шлях*, $шлях \in P$, такий, що для відповідної йому заміни $Q_{шлях}$ рівність порушується, тобто $A'Q_{шлях} \neq A"Q_{шлях}$. Але звідси в силу теореми 2 випливає, що програми p_1 і p_2 не є логіко-смислово подібними, що суперечить вихідному припущенню.

В основі наведеного алгоритму лежить ітеративна процедура, яка на кожному кроці обчислює заміну: кожен раз для деякої трійки замін Q_1, Q_2, Q_3 обчислюється заміна $(Q_1Q_2)\downarrow Q_3$. Але відповідно до формули (2.2), розмір ациклічного орієнтованого графа, що реалізує заміну $(Q_1Q_2)\downarrow Q_3$ можна оцінити величиною, пропорційною розмірам графів вихідних замін. У гіршому випадку наведений в цьому розділі алгоритм повинен здійснити кількість ітерацій, пропорційну розмірам вихідних програм. З урахуванням згаданої вище складності обчислення точної нижньої межі і з урахуванням можливого зростання розміру ациклічного орієнтованого графу замін виходить, що алгоритм має експонентну складність. Для того, щоб усунути цей недолік алгоритму, далі розглянута операція, що представляє скорочену деуніфікацію замін. Ця операція також обчислює нижню межу двох замін, при цьому вона не суперечить теоремі про коректність алгоритму, але є простішою: ця операція здійснюється за лінійний час, а ациклічний орієнтований граф, що представляє її результат, за розміром не перевищує мінімальний з графів вихідних замін.

2.3 Скорочені заміни і алгоритм скорочення

Розглянемо множину замін $Заміна(X,Y,F)$, визначених на кінцевій множині змінних $X=\{x_1,x_2,\ldots,x_n\}$ і нескінченній множині змінних $Y=\{y_1,y_2,\ldots\}$. Назвемо змінні множини X головними змінними, а змінні множини Y, використовувані для побудови лексем – допоміжними змінними. Для кожної заміни Q, $Q\in Заміна(X,Y,F)$, розіб'ємо множину змінних $Змінна_Q$ на дві підмножини: $НЗмінна_Q=\{y:\exists x(x\in X\ \&\ Q(x)=y)\}$ і $NЗмінна_Q=Змінна_Q\setminus V$. Зауважимо, що в графовій реалізації заміни Q всі вузли, помічені змінними з множини $НЗмінна_Q$ мають заголовки. Введене розбиття множини змінних дозволяє виділити в множині $Заміна(X,Y,F)$ підмножина скорочених замін.

Визначення 1. Заміна Q, $Q\in Заміна(X,Y,F)$ називається скороченою, якщо для неї виконано рівність: $Змінна_Q=НЗмінна_Q$.

З визначення 1 слідують наступні характеристичні властивості скорочених замін:

– в ациклічному орієнтованому графі, що реалізує скорочену заміну, всі листові вузли озаглавлені;

– кожна заміна θ має хоча б один скорочений прототип. Так, наприклад, порожня заміна $\{x_1/y_1,x_2/y_2,\ldots,x_n/y_n\}$ є скороченою.

Нехай задана деяка заміна $Q=\{x_1/t_1(\ldots),x_2/t_2(\ldots),\ldots,x_n/t_n(\ldots)\}$, $Q\in Заміна(X,Y,F)$. Процесом скорочення заміни Q назвемо наступну послідовність дій. Виберемо з множини Y змінну y таку, що $y\notin Змінна_Q$. Припустимо, що входить в l-у зв'язку заміни Q лексема t_l містить під-лексему $f(y_1,y_2,\ldots,y_n)$ таку, що хоча б для однієї j, $1\le j\le k$, вірно, що $y_i\notin НЗмінна_Q$. Для кожної t_i, $1\le i\le n$, побудуємо лексему t'_i відповідно до одного з наступних правил:

– якщо t_i не містить під-лексему $f(y_1, y_2, \ldots, y_n)$, то $t'_i = t_i$;

– якщо t_i містить під- лексему $f(y_1, y_2, \ldots, y_n)$, то t'_i виходить з t_i синхронною заміною всіх входжень під-лексеми $f(y_1, y_2, \ldots, y_n)$ на змінну y.

Розглянемо нову заміну $Q' = \{x_1/t'_1(\ldots), x_2/t'_2(\ldots), \ldots, x_n/t'_n(\ldots)\}$. З побудови t'_i видно, що $Q = Q'\{y/f(y_1, \ldots y_k)\}$. Крім того, $Змінна_{Q'} = Змінна_Q \cup \{y\}$.

Покажемо, що для побудованої таким чином пари замін Q і Q' і довільної скороченою заміною n справедливо наступне твердження.

Лема 8. Скорочена заміна n є прототипом заміни Q в тому і лише тому випадку, коли n є прототипом заміни Q'.

Доведення. Розглянемо довільний скороченої прототип n заміни Q. З огляду на те, що n – прототип Q, існує деяка заміна ρ, $\rho \in Заміна(Y, Y, F)$ така, що $n\rho = Q = Q'\{y/f(y_1, \ldots, y_k)\}$. Без обмеження спільності будемо вважати, що заміна ρ не містить зв'язок-перейменувань. Але тоді одна з замін n або ρ повинна містити в одній зі своїх зв'язок під-лексем $f(y_1, y_2, \ldots, y_n)$. Але з огляду на те, що заміна n є скороченою, а змінна y_j не входить до множини $НЗмінна_Q$, лексема $f(y_1, y_2, \ldots, y_n)$ не може входити до складу жодної зі зв'язок заміна n. Значить, віна є під-лексемою деякої лексеми однієї зі зв'язок заміни ρ. Але тоді заміну ρ можна представити у вигляді $\rho = \rho'\{y/f(y_1, y_2, \ldots, y_n)\}$, $\rho' \in Заміна(Y, Y, F)$, де ρ' отримана в результаті тих же дій, що і Q' з заміна Q. Виходить, вираз $f(y_1, y_2, \ldots, y_n)$ не входить до складу жодної під-лексеми замін ρ', Q'. Але тоді $n\rho' = Q'$, тобто n є прототипом Q'.

Протилежне твердження випливає з того факту, що $Q = Q'\{y/f(y_1, \ldots, y_k)\}$, тобто кожен прототип Q' є прототипом Q.

Визначення 2. Найбільш спеціальним скороченням заміни Q назвемо таку скорочену заміну Q', що Q – приклад заміни Q' і будь-яка скорочена заміна, що є прототипом Q, буде прототипом заміни Q'.

Введемо також для найбільш спеціального скорочення позначення: $Q' = нсс(Q)$.

Лема 9. Для кожної заміни Q, $Q \in Заміна(X, Y, F)$ існує її найбільш спеціальне скорочення $нсс(Q)$. Найбільш спеціальне скорочення єдине з точністю до перейменування змінних.

Доведення. Доказ цього твердження випливає з леми 8.

Опишемо алгоритм побудови найбільш спеціального скорочення заміни Q. Цей алгоритм працює з ациклічним орієнтованим графом Γ_Q заміни Q, проводячи його фарбування.

1. Вузли графа, що мають заголовки, фарбуються в синій колір. Всі вузли, яким приписані допоміжні змінні з множини $NЗмінна_Q$, фарбуються в червоний колір.

2. Будуємо фарбування графа:

– кожний вузол червоного кольору фарбує в червоний колір всі вхідні в неї дуги;

– якщо з вузла виходить хоча б одна червона дуга, то всі виходячі з неї дуги фарбуються в червоний колір;

– якщо вузол не синього кольору, а всі вихідні з неї дуги -–червоні, то цей вузол фарбується в червоний колір;

– дії повторюються до тих пір, поки можливе застосування хоча б одного із зазначених вище правил фарбування.

3. Для тих вузлів, у яких є заголовки, і у тому числі виходять лише червоні дуги, вводиться в якості позначки нова допоміжна змінна.

4. Всі червоні дуги і вузли віддаляються з графа.

Зауваження. Даний алгоритм може працювати з графами, які не є коректною реалізацією заміни: такі графи можуть містити вузли, недосяжні з озаглавлених вузлів. Для цього перед початком роботи першого кроку алгоритму всі такі вузли повинні бути пофарбовані в червоний колір.

Отриманий в результаті описаної вище процедури розмічений орієнтований граф є графовою реалізацією заміни $нсс(Q)$. Приклад роботи алгоритму для заміни $Q=\left\{x_1/f^2\left(h^1\left(f^2\left(y_1,y_2\right)\right),g^1\left(y_2\right)\right),x_2/g^1\left(y_2\right),x_3/h^1\left(f\left(y_1,y_2\right)\right),x_4/y_1\right\}$ наведено на рис. 2.1. Результатом є заміна $нсс(Q)=\left\{x_1/f\left(y_3,y_4\right),x_2/y_4,x_3/y_3,x_4/y_1\right\}$. Час роботи описаного алгоритму пропорційний до розміру графа G_0 вихідної заміни.

Наведемо далі кілька важливих властивостей найбільш спеціальних скорочень замін. Обґрунтування цих властивостей спирається на запропонований алгоритм побудови найбільш спеціального скорочення заміни.

Лема 10. Для будь-яких двох замін Q_1 і Q_2 таких, що $Q_1 \le Q_2$, вірно, що їх скорочення знаходяться в співвідношенні $нсс(Q_1) \le нсс(Q_2)$.

Доведення. Зауважимо, що з факту $Q_1 \le Q_2$ слідує, що існує така заміна ρ, $\rho \in Заміна(Y,Y,F)$, що виконується рівність: $Q_2 = Q_1\rho$.

Тоді для обґрунтування справедливості леми достатньо показати нерівність

$$нсс(Q_1) \le нсс(Q_1\rho)$$

Розглянемо процес побудови заміни $нсс(Q_1\rho)$. Множина "синіх" вузлів в ациклічному орієнтованому графі Q_1 за побудовою буде тією ж, що і в ациклічному орієнтованому графі $Q_1\rho$. Це, в свою чергу, означає, що отримані в результаті скорочення ациклічні орієнтовані графи будуть відрізнятися лише наборами дуг. При цьому в скороченому графі, відповідному Q_1, таких дуг не може бути більше, ніж в ациклічному орієнтованому графі $нсс(Q_1\rho)$. А це, в свою чергу, означає, що $нсс(Q_1\rho)$ буде прикладом $нсс(Q_1)$. Наступна лема встановлює властивості скорочення щодо операції композиції замін.

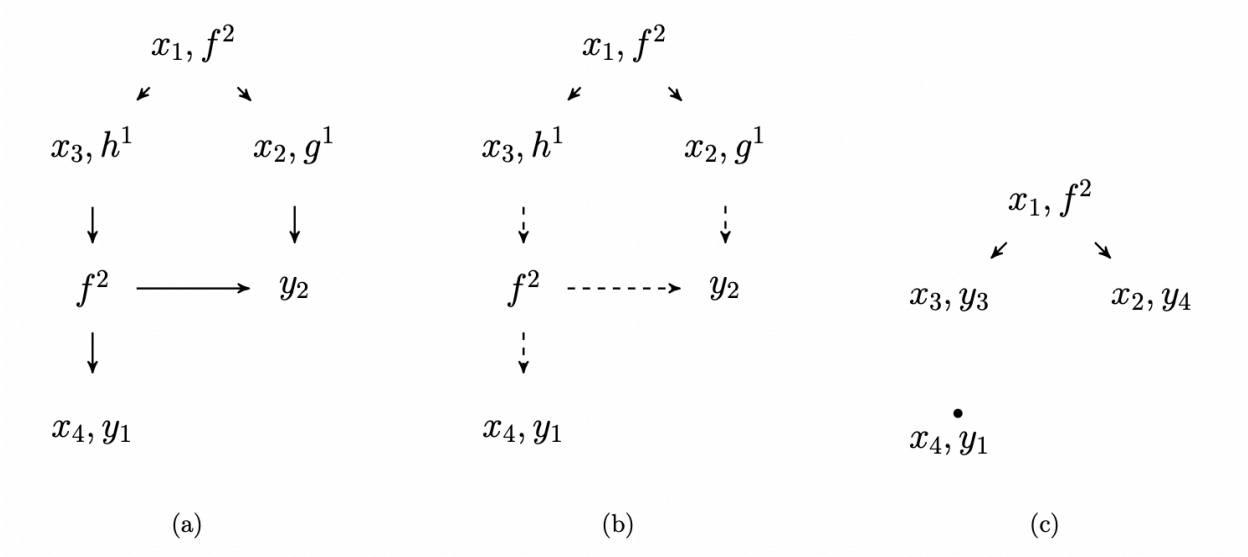


Рис. 2.1. Приклад побудови найбільш спеціального скорочення: (a) ациклічний орієнтований граф вхідної заміни Q; (b) результат фарбування дуг; (c) ациклічний орієнтований граф найбільш спеціального скорочення заміни Q

Лема 11. Нехай $Q \in \textit{Заміна}(X, X, F)$, $n \in \textit{Заміна}(X, Y, F)$. Тоді справедливе наступне співвідношення:

$$\textit{нсс}(Qn) \sim \textit{нсс}(Q\textit{нсс}(n)).$$

Доведення. Зауважимо, що існує така заміна ρ, $\rho \in \textit{Заміна}(Y, Y, F)$, що $n = \textit{нсс}(n)\rho$, тобто вираз $\textit{нсс}(Qn)$ представимо у вигляді $\textit{нсс}(Q\textit{нсс}(n)\rho)$. Будемо розглядати лише той випадок, коли ρ не є перейменуванням, тому що інакше співвідношення очевидно. Розглянемо для заміни $Q\textit{нсс}(n)\rho$ процедуру скорочення: скоротимо в ній усі дуги і вузли, відповідні заміні ρ. Тоді за лемою 8 отримуємо, що кожен скорочений прототип заміни $Qn = Q\textit{нсс}(n)\rho$ є скороченим прототипом заміни $Q\textit{нсс}(n)$, а кожен скорочений прототип $Q\textit{нсс}(n)\rho$ в свою чергу є скороченим прототипом Qn, з чого і слідує ствердження леми.

Для подальшого використання скорочення замін нам знадобиться також наступна властивість скорочень щодо атомарних виразів.

Теорема 4. Для будь-якої пари атомарних формул A,B, $A,B \in Atom(X,F)$ і для будь-якої заміни Q, $Q \in Заміна(X,Y,F)$ співвідношення $AQ = BQ$ справедливе тоді і лише тоді, коли $Ансс(Q) = Внсс(Q)$.

Доведення теореми 4. (⇐) За визначенням, заміна $нсс(Q)$ є прототипом заміни Q, що підтверджує $Ансс(Q) = Внсс(Q) \Rightarrow AQ = BQ$.

(⇒) Покажемо, що якщо $AQ = BQ$, то і $Ансс(Q) = Внсс(Q)$. За умовою теореми, атоми A і B уніфіковувані, їх уніфікатором є заміна Q. А це означає, що для них може бути побудований найбільш загальний уніфікатор n, $n \in Заміна(X,X,F)$ такий, що справедлива рівність $Dom_n \cap Змінна_n = \varnothing$. Нехай $Змінна_n = \{x_{j_1}, x_{j_2}, \ldots, x_{j_m}\}$. Розглянемо перейменування цих змінних $L = \{x_{j_1}/y_1, x_{j_2}/y_2, \ldots, x_{j_m}/y_m\}$. Тоді заміна $Q' = nL$ також буде найбільш загальним уніфікатором тієї ж пари атомів, тобто існує така заміна ρ, $\rho \in Заміна(Y,Y,F)$, що $Q = Q'\rho$ і, разом з тим, $AQ' = BQ'$. При цьому θ' буде також скороченою заміною з множини $Заміна(Y,Y,F)$, а отже, буде скороченим прототипом заміни Q. Це, в свою чергу означає, що для деякої заміни ρ', $\rho' \in Заміна(Y,Y,F)$ буде виконано рівність $нсс(Q) = Q'\rho'$, з чого слідує $Ансс(Q) = Внсс(Q)$.

2.4 Скороченна деуніфікація замін

Введемо операцію скорочення деуніфікації замін, яка дозволяє застосовувати апарат скорочених замін до вирішення завдання перевірки логіко-смислової подібності програм надалі.

Визначення 3. Скороченою деуніфікацією замін Q_1, Q_2, $Q_1, Q_2 \in \textit{Заміна}(X, Y, F)$ – така заміна Q, що $Q = \textit{нсс}(Q_1 \downarrow Q_2)$.

Введемо для позначення скороченої деуніфікації замін наступний запис: $Q = Q_1 \Downarrow Q_2$. Виділимо важливу властивість операції скороченої деуніфікації.

Теорема 5. Для будь-якої пари замін n_1, n_2 таких, що $n_1, n_2 \in \textit{Заміна}(Y, Y, F)$ справедлива рівність:

$$n_1 \Downarrow n_2 \sim \textit{нсс}(n_1) \Downarrow \textit{нсс}(n_2).$$

Доведення теореми 5. Так як $n_1 \geq \textit{нсс}(n_1)$ і $n_2 \geq \textit{нсс}(n_2)$, справедлива наступна нерівність: $\textit{нсс}(n_1) \downarrow \textit{нсс}(n_2) \leq n_1 \downarrow n_2$. Тоді, з урахуванням леми 10, справедливе співвідношення $\textit{нсс}(n_1) \Downarrow \textit{нсс}(n_2) \leq n_1 \Downarrow n_2$.

Тепер покажемо, що $\textit{нсс}(n_1) \Downarrow \textit{нсс}(n_2) \leq n_1 \Downarrow n_2$, з чого і буде слідувати ствердження леми. Нам необхідно показати, що кожен скорочений прототип заміни $n_1 \downarrow n_2$ є скороченим прототипом заміни $\textit{нсс}(n_1) \downarrow \textit{нсс}(n_2)$. Для цього розглянемо довільну скорочену заміну Q з множини $\textit{Заміна}(X, Y, F)$, яка є прототипом заміни $n_1 \downarrow n_2$. Але тоді обрана заміна θ також є прототипом самих замін n_1 і n_2. Оскільки вона є скороченою, то, крім цього, вона також є прототипом замін $\textit{нсс}(n_1)$ і $\textit{нсс}(n_2)$, тобто $Q \leq \textit{нсс}(n_1)$ і $Q \leq \textit{нсс}(n_2)$. А з цього слідує справедливість наступного співвідношення: $Q \leq \textit{нсс}(n_1) \downarrow Q \leq \textit{нсс}(n_2)$. Але оскільки заміна Q – довільний скорочений прототип $n_1 \downarrow n_2$ є скороченим прототипом заміни $\textit{нсс}(n_1) \downarrow \textit{нсс}(n_2)$, то і $n_1 \Downarrow n_2 \leq \textit{нсс}(n_1) \Downarrow \textit{нсс}(n_2)$.

Оскільки справедливі нерівності $n_1 \Downarrow n_2 \leq \textit{нсс}(n_1) \Downarrow \textit{нсс}(n_2)$ і $\textit{нсс}(n_1) \Downarrow \textit{нсс}(n_2) \leq n_1 \Downarrow n_2$, замін $\textit{нсс}(n_1) \Downarrow \textit{нсс}(n_2)$ і $n_1 \Downarrow n_2$ подібні.

Наступні дві теореми є аналогами теореми 1 і леми 3 для скороченої деуніфікації замін.

Теорема 6. Для будь-якої заміни Q, $Q \in Заміна(X,X,F)$, і пари замін n_1,n_2, $n_1,n_2 \in Заміна(X,Y,F)$ справедливо $Qn_1 \Downarrow Qn_2 = нсс\left(Q\left(n_1 \Downarrow n_2\right)\right)$.

Доведення теореми 6. З леми 11 і визначення скороченої деуніфікації слідує справедливість такого ланцюжка рівностей: $Qn_1 \Downarrow Qn_2 = нсс\left(Qn_1 \downarrow Qn_2\right) = нсс\left(Q\left(n_1 \downarrow n_2\right)\right) = нсс\left(Qнсс\left(n_1 \downarrow n_2\right)\right) = нсс\left(Q\left(n_1 \Downarrow n_2\right)\right)$.

Теорема 7. Для будь-якої пари атомів A', A'' таких, що $A',A'' \in Atom(X,F)$ і будь-якої пари замін n_1,n_2, $n_1,n_2 \in Заміна(X,Y,F)$ справедливе наступне співвідношення:

$$\begin{cases} A'n_1 = A''n_1 \\ A'n_2 = A''n_2 \end{cases} \Leftrightarrow A'\left(n_1 \Downarrow n_2\right) = A''\left(n_1 \Downarrow n_2\right)$$

Доведення теореми 7. За умовою леми 3 справедливо:

$$\begin{cases} A'n_1 = A''n_1 \\ A'n_2 = A''n_2 \end{cases} \Leftrightarrow A'\left(n_1 \downarrow n_2\right) = A''\left(n_1 \downarrow n_2\right)$$

Але тоді з урахуванням теореми 5 рівність $A'\left(n_1 \downarrow n_2\right) = A''\left(n_1 \downarrow n_2\right)$ виконується тоді і лише тоді, коли справедливо $A'\left(n_1 \Downarrow n_2\right) = A''\left(n_1 \Downarrow n_2\right)$, що доводить ствердження леми.

Визначення 3 скороченої деуніфікації лексем формально задає алгоритм побудови скороченої нижньої межі замін як послідовного застосування спершу операції деуніфікації, а потім алгоритму побудови найбільш спеціальним скороченням заміни. Але тоді обчислення скороченої нижньої межі і розмір результуючої заміни матимуть квадратичну залежність від розмірів ациклічних орієнтованих графів вхідних замін. Але така залежність призведе до того, що алгоритм перевірки логіко-смислової подібності виявиться експоненційним. Тому побудуємо скорочену нижню межу замін, або алгоритм обчислення скороченої деуніфікації, що має лінійну складність.

Нехай задані дві заміни Q_1,Q_2 такі, що $Q_1,Q_2 \in Заміна(X,Y,F)$. Нехай ациклічні орієнтовані графи Γ_{Q_1} і Γ_{Q_2} реалізують відповідно заміні Q_1 і Q_2.

Для заміни $Q = Q_1 \Downarrow Q_2$ буде побудований ациклічний орієнтований граф Γ_Q, вузлами якого будуть впорядковані пари $w = (u, v)$, де u є вузлом графу Γ_{Q_1}, а v – відповідно вузол графа Γ_{Q_2}.

Побудова цього графу буде здійснюватися в три етапи.

1. Побудова множини помічених вузлів. На цьому етапі здійснюється вибір таких пар $w_i = (u_i, v_i)$, що обидва вузла u_i і v_i мають однакові заголовки x_i у відповідних графах. Для кожної з таких пар в графі Γ_Q будується відповідний парі w_i вузол, який позначається тим же заголовком x_i, що і вихідні вузли.

2. Поповнення множини вузлів. На цьому етапі здійснюється додавання вузлів, що входять в граф, але не мають заголовків. Розглянемо пару вузлів (u_0, v_0) таких, що вони позначені одним і тим же функціональним символом f^n, $f^n \in F$. Припустимо, що для будь-якого натурального i, $1 \le i \le n$ вірно, що пара (u_i, v_i) задовольняє наступним двом вимогам: 1) вузол u_i є i-спадкоємцем вузла u_0 в графі Γ_{Q_1}, а вузол v_i є i-спадкоємцем вузла v_0 в графі Γ_{Q_2}; 2) пара $w_i = (u_i, v_i)$ вже внесена до множини вузлу графу Γ_Q. Тоді, якщо вузол $w_0 = (u_0, v_0)$ ще не був внесений до множини вузлів графу Γ_Q, то вона до неї вноситься. Вузол w_0 позначається відповідним функціональним символом f^m, а для кожного i, $1 \le i \le n$ проводиться дуга від (u_0, v_0) в вузол (u_i, v_i). Цей крок повторюється до тих пір, поки до графа Γ_Q можна додати хоча б одну дугу.

3. Всі листові вузли, озаглавлені змінною з множини X, позначаються попарно різними допоміжними змінними з множини Y. Після цього застосовується процедура скорочення з урахуванням зауваження, що дозволяє працювати з надлишковими графами.

Лема 12. Алгоритм побудови скороченої точної нижньої межі двох замін коректний.

Доведення. Для доведення ствердження леми необхідно довести наступні три факти:

1) побудована в результаті роботи алгоритму заміна Q є скороченою;

2) побудована заміна Q є прототипом замін Q_1 і Q_2;

3) зазначена заміна є найбільш спеціальним скороченням заміни $Q_1 \downarrow Q_2$.

Перший пункт виконується відповідно до третього кроку алгоритму побудови скороченого деуніфікатора. Виконання другого пункту також випливає з опису алгоритму. Залишається показати, що отримана в результаті роботи алгоритму заміна Q є не просто скороченою деуніфікатором, але найбільш спеціальним з них. Справедливість цього ствердження, в свою чергу, випливає з правил додавання дуг в граф Γ_Q.

Наведемо оцінку розміру ациклічного орієнтованого графа, що реалізує скорочену деуніфікацію двох замін, а також оцінку часу роботи цього алгоритму.

Лема 13. Кількість кроків алгоритму побудови скороченої деуніфікації не більше, ніж $O\left(|X| + \min\left(\left|\Gamma_{Q_1}\right|, \left|\Gamma_{Q_2}\right|\right)\right)$.

Доведення. На першому кроці своєї роботи вказаний алгоритм здійснює не більше $|X|$ дій. Другий крок роботи алгоритму здійснює поповнення множини вузлів, і кількість ітерацій поповнення, що відбуваються на цьому кроці, не перевищує кількості вузлів в кожному з вихідних графів Γ_{Q_1} і Γ_{Q_2}, тобто $\min\left(\left|\Gamma_{Q_1}\right|, \left|\Gamma_{Q_2}\right|\right)$. Процедура скорочення, що проводиться на третьому кроці, має лінійну складність. Таким чином, кількість кроків алгоритму може бути зверху оцінено величиною $O\left(|X| + \min\left(\left|\Gamma_{Q_1}\right|, \left|\Gamma_{Q_2}\right|\right)\right)$.

Але для того, щоб оцінити розмір графу скороченого деуніфікатора, необхідно ввести ще одну складнісну характеристику АОГ для скорочених замін. Нехай n – довільна скорочена заміна з множини $Заміна(X, Y, F)$, а

ациклічний орієнтований граф Γ_n - її реалізація. Складністю скороченої заміни назвемо величину $l(n) = N_F(n) + |X| - N_X(n)$, де $N_F(n)$ – кількість вузлів в графі Γ_n, помічених функціональними символами з множини F, а $N_X(n)$ - кількість вузлів графа Γ_n, яким приписані змінні з множини X.

Для введеної таким чином складності скороченої заміни справедливо наступне ствердження.

Лема 14. Нехай $Q = Q_1 \Downarrow Q_2$. Тоді справедлива нерівність: $l(Q) \le \min\left(l(Q_1), l(Q_2)\right)$.

Доведення. Оскільки $l(Q) = N_F(Q) + |X| - N_X(Q)$, досить розглянути оцінки величин $N_F(Q)$ і $N_X(Q)$. З опису алгоритму слідує, що кожний вузол, що додається в граф Γ_Q на другому етапі роботи алгоритму, взаємно однозначно відповідає рівно одному вузлу графу Γ_{Q_1} і одному вузлу графа Γ_{Q_2}, з чого слідує нерівність $N_F(Q) \le \min\left(N_F(Q_1), N_F(Q_2)\right)$. Крім того, величина $N_X(Q)$ точно не менше, ніж кількість озаглавлених змінних в кожній з замін Q_1 і Q_2, тобто $N_X(Q) \ge \max\left(N_X(Q_1), N_X(Q_2)\right)$. З цього вже випливає ствердження леми.

Лема 15. Нехай $Q = Q_1 \Downarrow Q_2$, і при цьому $l(Q) = l(Q_i)$, $i \in \{1,2\}$. Тоді $Q \sim Q_i$.

Доведення. Виходячи з того, що $N_F(Q) \le \min\left(N_F(Q_1), N_F(Q_2)\right)$ і $N_X(Q) \ge \max\left(N_X(Q_1), N_X(Q_2)\right)$, зі збігу розмірів $l(Q) = l(Q_i)$, $i \in \{1,2\}$ слідує, що графи, які реалізують заміни Q і Q_i ізоморфні.

Наведені вище ствердження дозволяють застосувати алгоритми, що працюють із скороченими замінами, до перевірки логіко-смислової подібності програм.

2.5 Модифікований алгоритм перевірки логіко-смислової подібності, його коректність і складність

Описані властивості скороченої деуніфікації замін дозволяють внести в алгоритм перевірки логіко-смислової подібності наступну зміну.

Розглянемо алгоритм перевірки логіко-смислової подібності фрагментів програм, замінивши в ньому операцію взяття точної нижньої межі замін на операцію скороченої деуніфікації замін. Отриманий в результаті такої заміни операцій алгоритм будемо далі називати модифікованим алгоритмом перевірки логіко-смислової подібності програм.

Теорема 8. Модифікований алгоритм перевірки логіко-смислової подібності коректний.

Доведення теореми 8. Зауважимо, що леми 4, 5, 6, 7, а також теореми 3, 4 з усіх властивостей операції взяття точної нижньої межі замін спираються лише на властивості, показані в лемах 1 і 3. Але справедливість цих же властивостей для операції скороченої деуніфікації показана в лемі 11 і теоремах 4, 5, 6, 7. Це означає, що зазначені леми зберігають справедливість всіх тверджень, що обґрунтовують коректність алгоритму перевірки логіко-смислової подібності.

Наведені в лемах 13 і 14 оцінки часу виконання скороченої деуніфікації і розміру її результату дозволяють привести оцінку трудомісткості модифікованого алгоритму перевірки логіко-смислової подібності програм.

Теорема 9. Існує алгоритм, який вирішує завдання перевірки логіко-смислової подібності довільної пари програм p_1 і p_2 за $O\left(n^6\right)$ кроків, де $n = \max\left(|p_1|,|p_2|\right)$.

Доведення. Для наведеного в цьому розділі модифікованого алгоритму перевірки логіко-смислової подібності програм вже була обґрунтована його коректність. Наведемо оцінку його складності.

Цей алгоритм працює з графом спільних обчислень вхідних програм, для якого кількість його вузлів свідомо не перевищує величини n^2. Розмір кожної приписаної йому вузлів замін $n_{(v',v'')}$ може бути оцінений зверху величиною n^2. З леми 14 і зауваження до неї слідує, що на кожному кроці розмір хоча б однієї з замін, приписаних до вузлів графу спільних обчислень, зменшується. Це означає, що кількість ітерацій алгоритму може бути оцінена величиною $O\left(n^4\right)$.

На кожній ітерації алгоритм здійснює процедуру обчислення скороченої деуніфікації замін $Qn_{(v',v'')} \Downarrow n_{(u',u'')}$. Лема 13 показує, що складність здійснення цієї процедури лінійно залежить від розмірів замін. Тоді складність всього алгоритму можна оцінити $O\left(n^6\right)$, що й треба було довести.

2.6 Методика отримання результатів

В основі алгоритмів, представлених в даній роботі, лежить метод спільних обчислень. Цей метод неодноразово використовувався при дослідженні проблеми подібності програм [42, 43], схожі ідеї також були представлені в роботі [124].

Основна ідея методу спільних обчислень може бути коротко сформульована наступним чином. При описі моделі, як було сказано вище, задається інтерпретація. В рамках цієї інтерпретації дві програми, проводячи свої обчислення одночасно, будують пари обчислювальних конфігурацій. В результаті проведення аналізу всіх можливих побудованих пар, метод спільних обчислень дозволяє зробити висновки про їх подібність.

У даній роботі метод спільних обчислень розглядається в застосуванні до перевірки логіко-смислової подібності стандартних схем послідовних імперативних програм, а також до задачі перевірки уніфікуючих двох

заданих програм. Покажемо, як саме конкретизується метод спільних обчислень при розгляді поставленого завдання.

Модель стандартних схем програм передбачає, що кожному обчисленню програми може бути поставлений у відповідність деякий бінарний вектор, що описує шлях, за яким відбувається обхід графа відповідно до заданої інтерпретації. Два шляхи в схемах програм p_1 і p_2 вважаються узгодженими, якщо відповідні їм вектори збігаються. Метод спільних обчислень при цьому має на увазі дослідження обчислень, що проходять за узгодженими шляхами в програмах. Для проведення такого дослідження використовується спеціального виду графова структура Γ_{p_1,p_2}, яка називається графом спільних обчислень або графом узгоджених маршрутів вхідних програм. Перед побудовою графу спільних обчислень здійснюється перейменування змінних програм таким чином, щоб множини змінних не перетиналися. Вузлами графу спільних обчислень є елементи декартового добутку вущлів графів, що реалізують програми. Дуги при цьому будуються наступним чином: з вузлів (v',v'') дуга веде в вузол (u',u'') в тому і лише в тому випадку, якщо в графі програми p_1 дуга з бінарної позначкою d веде з вузла v' в вузол u', а в графі програми p_2 дуга з точно такою ж позначкою веде з вузла v'' в вузол u''. При цьому заміна, приписана побудованій в графі спільних обчислень дузі, буде об’єднанням замін на відповідних дугах в початкових програмах.

Зауважимо, що в роботі [80] був запропонований алгоритм перевірки логіко-смислової подібності, що спирається на методику, схожу з методом спільних обчислень. Після попередніх подібних перетворень вхідних фрагментів програм цей алгоритм перевіряє подібність отриманих перетворених фрагментів. Ця перевірка подібності реалізується за допомогою алгоритму, схожого на алгоритм розпізнавання подібності двохстрічкових автоматів, запропонований Бердом [110]. Фактично ця частина алгоритму розпізнавання будує структуру - узгоджену пару

фрагментів, - достатньо схожу на граф спільних обчислень, але для вже наведених фрагментів програм. Ставлячи певні умови коректності цієї структури, застосовуючи перетворення і будуючи її розмітку, алгоритм робить висновки про подібність програм. Було доведено [53], що для вирішення завдання перевірки логіко-смислової подібності може бути успішно застосований апарат обчислення нерухомих точок монотонних операторів на решітках. В роботі [80] в цій якості виступає решітка функціональних мереж.

Алгоритм перевірки логіко-смислової подібності, заснований на методі спільних обчислень, запропонований в даній роботі, спирається на алгоритм, запропонований В. К. Сабельфельдом в роботі [80]. Істотна відмінність полягає в двох моментах. По-перше, алгоритм, запропонований Сабельфельдом, спирається на систему подібних перетворень, кожне з яких використовується на певному етапі алгоритму. По-друге, при побудові стаціонарної розмітки графа узгодженої пари фрагментів, цей алгоритм спирається на властивості напіврешітки функціональних мереж. У даній роботі запропоновано алгоритм перевірки логіко-смислової подібності програм, який не використовує системи подібних перетворень, а в якості підходящої решітки була обрана решітка кінцевих замін, детально досліджена в статтях [129, 192]. Зокрема, в цих роботах було показано, що решітка замін має властивість обриву спадаючих ланцюгів, а також показано, що операція композиції дистрибутивна щодо операції взяття точної нижньої межі на множині замін. Саме ці дві властивості дозволяють використовувати зазначену решітку для вирішення завдань статичного аналізу програм.

Застосування методики спільних обчислень для поставленого завдання виявилося ефективним, так як за допомогою цієї методики і процедури деуніфікації замін вдалося побудувати поліноміальний за часом алгоритм перевірки логіко-смислової подібності програм. Цей алгоритм також ліг в основу алгоритму логіко-смислової уніфікації програм, описаного в третьому

розділі даної роботи. Цей алгоритм, крім методу спільних обчислень і деуніфікації замін, також використовує операцію уніфікації замін, досліджену в роботах [102, 103, 174, 196].

Висновки за розділом 2

Проведено дослідження алгоритмів перевірки логіко-смислової подібності стандартних послідовних схем програм, заснованих на пошуку найбільш схожих смислових траєкторій програми. Наведено основні поняття, пов'язані з графом спільних обчислень програм.

Отримав подальший розвиток метод перевірки логіко-смислової подібності програм складної логічної структури, що відрізняється від відомих розпаралелюванням процесів, що обчислюються при порівнянні подібних елементів програмного коду, а також формуванням та використанням графу спільних обчислень в процесі пошуку подібних елементів коду. Це дозволило розпаралелити дані процеси та досягти поліноміальної складності процесу верифікації, що в свою чергу зменшило час перевірки коду на логіко-смислову подібність для визначення впливу розробленого методу обфускації коду на його коректність.

РОЗДІЛ 3. РОЗРОБКА УНІФІКОВАНОЇ МАТЕМАТИЧНОЇ МОДЕЛІ ПРОЦЕСУ ОБФУСКАЦІЇ ПРОГРАМНИХ МОДУЛІВ НА БАЗІ МЕТОДУ ГРАФІЧНОЇ ОЦІНКИ НА АНАЛІЗ

В умовах повсюдного використання комп'ютерних систем підвищується цінність і роль програмного забезпечення. Це обумовлює зростання кількості злочинних кібератак, спрямованих на дискредитацію програмних продуктів і зниження безпеки функціонування комп'ютерних систем в цілому.

Відповідно до вимог законів та законодавчих актів [38, 39], програмне забезпечення (комп'ютерні програми) є одним з об'єктів, які потребують захисту на всіх етапах розробки і експлуатації.

Для аналізу питання захисту ПЗ необхідно виявити основні завдання інформаційної безпеки. До них належать [13]:

- конфіденційність;
- цілісність;
- доступність;
- невідмовність.

При цьому, процес обфускації програмного забезпечення на етапі розробки може стати одним з ключових для забезпечення послуг безпеки конфіденційності, аутентифікації і цілісності [13]. Пов'язано це, в першу чергу, з реалізацією даних алгоритмів і процедур обфускації на самих ранніх етапах розробки і реалізації коду, а також їх розширеними можливостями при забезпеченні множини послуг безпеки в цілому.

Таким чином, обфускація є важливою складовою забезпечення практичних послуг безпеки.

Аналіз літератури показав, що для ПЗ, написаного на різних мовах програмування, існують різні механізми обфускації. Відмінність механізмів

обфускації пов’язана з варіативністю мов програмування, які можна поділити на 4 групи [18, 34, 201]:

– скриптові (наприклад, Javascript, PHP) та ті, що інтерпретуються (наприклад, PHP, Perl, Ruby). Дана категорія мов програмування характеризується тим, що програмний продукт поставляється у відкритому вигляді та обфускація виконується для початкового коду (source code);

– ті, що компілюються (наприклад, C, C++, Delphi, Assembler). Дана категорія мов програмування характеризується тим, що програмний продукт поставляється у вигляді бінарних (скомпільованих) файлів (файли, що виконуються, бібліотеки), які: 1) важко декомпілювати (перетворювати назад до початкового коду); 2) мають можливість вбудовування системно- та процесорно-специфічних трюків, що дозволяє підняти якість обфускованості коду на високий рівень. Незважаючи на свої беззаперечні переваги, дана категорія мов має наступні недоліки: 1) дороговизна підтримки коду, 2) відсутність кросплатформеності, 3) витіснення мовами програмування з проміжним кодом з ринку прикладних програмних продуктів;

– байт-код орієнтовані (наприклад, Java, C#). Ця категорія потребує особливої уваги, так як програмні продукти, які написані на мовах цієї категорії, мають «проміжний» код відповідної віртуальної машини (JVM, CLR), який близький до компільованого, але простіше декомпілюється. Так як проміжний код виконується певною відповідною віртуальною машиною, то використання хитрощів, як в компільованому коді, не є можливим.

Аналіз літератури [97, 231] показав, що саме байт-код орієнтовані мови програмування мають найвищий рівень популярності серед інших. До того ж, саме з використанням цих мов програмування розробляються Enterprise-застосунки, що формують найбільшу частину фінансової складової ринку IT-індустрії [238] у зв’язку з тим, що вартість розробки та підтримки може в 100 разів перевищувати розробку простих застосунків та сайтів.

Висока вартість програмного забезпечення потребує:

– підвищеного рівня безпеки самого програмного забезпечення для уникнення витоку даних користувача та корпорації;

– підвищеного рівня безпеки модуля авторизації програмного забезпечення, зокрема, модуля ліцензування. У зв'язку з тим, що вартість ліцензій може сягати сотні тисяч доларів, гостро стоїть проблема захисту авторських прав компаній-розробників програмного забезпечення. Одним з напрямів захисту програмного забезпечення та авторських прав на нього є обфускація.

Таким чином, дослідження методів обфускації програмного забезпечення, яке написане на байт-код орієнтованих мовах програмування, є актуальним та потребує проведення досліджень в цьому напрямку.

Процес обфускації програмного забезпечення складається з підпроцесів, які можуть бути використані або невикористані в залежності від бізнес-процесу, загального часу виконання і рівня захисту, що надається. Формування моделей для кожного окремого випадку є витратним процесом, що вимагає уніфікації.

У зв'язку з цим, актуальним завданням є розробка підходу, який базується на уніфікації математичної формалізації процесу захисту програмного забезпечення для оцінки ймовірнісних характеристик часу виконання процедур обфускації програмних модулів.

3.1 Аналіз методів моделювання

Збільшення кількості атак, спрямованих на зниження безпеки функціонування комп'ютерних систем, в ряді випадків пов'язане з розривом між розвитком теорії і практики застосування теоретичних результатів, а також недоскональністю математичних моделей, що не забезпечують підвищених вимог практиків.

Аналіз літератури [51, 76, 126, 128, 156, 168, 222, 225, 228] показав, що

для ряду окремих випадків представлені кінцеві результати дослідження з урахуванням можливих обмежень. Так, в [76, 126, 228] отримані вичерпні результати для випадку, коли випадковий процес, що визначає переходи з одного стану в інший, формалізується експоненціальним законом розподілу. Це дозволяє спростити вирішення поставлених завдань, але заздалегідь вносить похибку в описову частину моделі, формалізує немарківські системи.

В роботах [51, 128, 156, 168, 222, 225] наведені більш складні моделі, що засновані на принципах декомпозиції складних алгоритмів і архітектур приватного рівня. Однак, проблема отримання фінальних співвідношень для розрахунку ймовірнісно-часових характеристик для випадків, коли переходи між станами описуються складнішими, ніж експонентними законами розподілу, але не мають ознак «Марківських», вивчена недостатньо.

Слід зауважити, що крім обґрунтування математичного апарату вирішення поставленого завдання, важливим є вибір засобів математичної формалізації.

Аналіз підходів мережевого стохастичного моделювання показав велику їх різноманітність (на основі мереж Петрі [144], кінцевих дискретних автоматів [223], PERT-мереж [112] та ін.). Однак основним недоліком цього підходу до моделювання є обмеженість практичного застосування в зв’язку з відсутністю властивостей передбачуваності.

У роботах [50, 95, 215, 221] представлені GERT (Graphical Evaluation and Review Technique) моделі складних технічних систем і процесів. Однак введення допущення про експоненційний закон розподілу як основний закон, що характеризує процес переходу зі стану в стан, в значній мірі знижує їх як теоретичну, так і практичну цінність.

В роботі [95] пропонується розгляд ситуації, коли основним ітераційним процесом обфускації є напівмарковський. Однак, показано, що задача знаходження закону розподілу для напівмарковських моделей великої

розмірності вирішується з похибкою в близько 15%, а підсумковий розподіл є дискретним. Показано, що для вирішення завдань, що вимагають точного знання функції або щільності розподілу, дані моделі не підходять.

Аналіз літератури [50, 95, 112, 144, 223] показав, що існуючі методи моделювання мають як переваги, так і недоліки. При цьому слід врахувати той факт, що процес обфускації цими моделями не описується.

Таким чином, проведений аналіз показав, що ряд математичних моделей складних алгоритмів і процесів забезпечення безпеки програмного забезпечення формалізовані в термінах теорії графів. Досить часто передбачається, що функціонування системи в цілому можна описати одним законом розподілу. При цьому, можливі варіанти застосування законів розподілу і їх параметрів при переході зі стану в стан не враховуються. Рішення зазначеного протиріччя можливо шляхом математичної формалізації процесів з використанням GERT-структур. При цьому, теоретичний і практичний інтерес представляє знаходження розподілу ймовірностей переходів зі стану в стан в процесі забезпечення безпеки програмних продуктів, а також кінцевий результат у вигляді закону розподілу зі знайденими параметрами.

Слід зазначити, що дослідження комплексних GERT-мереж ускладнено через високі обчислювальні вимоги стохастичних підходів моделювання. При цьому, проблема розробки спрощених уніфікованих GERT-моделей вивчена недостатньо. Таким чином, виникає необхідність розробки уніфікованої моделі для формалізації процесу обфускації програмних модулів з метою усунення даного недоліку.

3.2 Аналіз методів обфускації

Аналіз літератури [108, 117, 131, 136, 138, 212] показав, що методи обфускації, які застосовуються в байт-код орієнтованих мовах програмування, можна поділити на такі типи:

– вставка мертвого коду. Цей код або нічого не робить, або до нього не доходить процес керування. Цей метод має можливість сконцентрувати увагу зловмисника на аналізі коду, що не має сенсу з точки зору бізнес-логіки. Це підвищує складність та дає можливість збільшити час на аналіз коду в цілому;

– лексичні перетворення. Цей тип найпримітивніший та включає до себе видалення символів, що не відносяться до коду (наприклад, коментарі, відступи);

– зміна послідовності виконання коду;

– заміна інструкцій на еквівалентні вирази. До цього засобу можна віднести заміну ідентифікаторів на довільний набор символів;

– фрагментація даних. Наприклад, код

System.out.println("48656c6c6f20576f726c6421");

замінити на

System.out.println(“48”,”65en”,”6c”,
”6c(fd”,”6f”,”2054”,“7g”,”6f5h”,
”72 __t”,”6c”,”64’h”,”21”).

З точки зору мови програмування Java, функція System.out.println у даному випадку буде виводити лише по 2 символи кожної строки, що дасть нам ідентичний результат;

– кодування/шифрування коду тексту алгоритму (олігоморфічне, поліморфічне, метаморфічне).

Аналіз показав, що в більшості випадків увага приділяється обфускації процесу захисту алгоритму, та практично не приділяється захисту самих даних.

Для оцінки рівня обфускації необхідно сформувати уніфікований композитний показник, який буде показувати ступінь захищеність програмного продукту від злочинного впливу, зокрема, зламу.

Дослідження показали, що якість обфускованості коду зворотньо-пропорційна якості даного коду. У зв'язку з цим, основним завданням даної статті є аналіз метрик оцінки якості та складності вихідного коду [117], які можна застосувати, надалі, і до декомпільованого коду. Надалі, аналіз метрик якості коду буде використовуватися для отримання кількісної оцінки якості захищеності коду на основі рівня обфускованості.

3.3 Розробка алгоритмів обфускації лексем

3.3.1 Розробка алгоритму обфускації строкових виразів

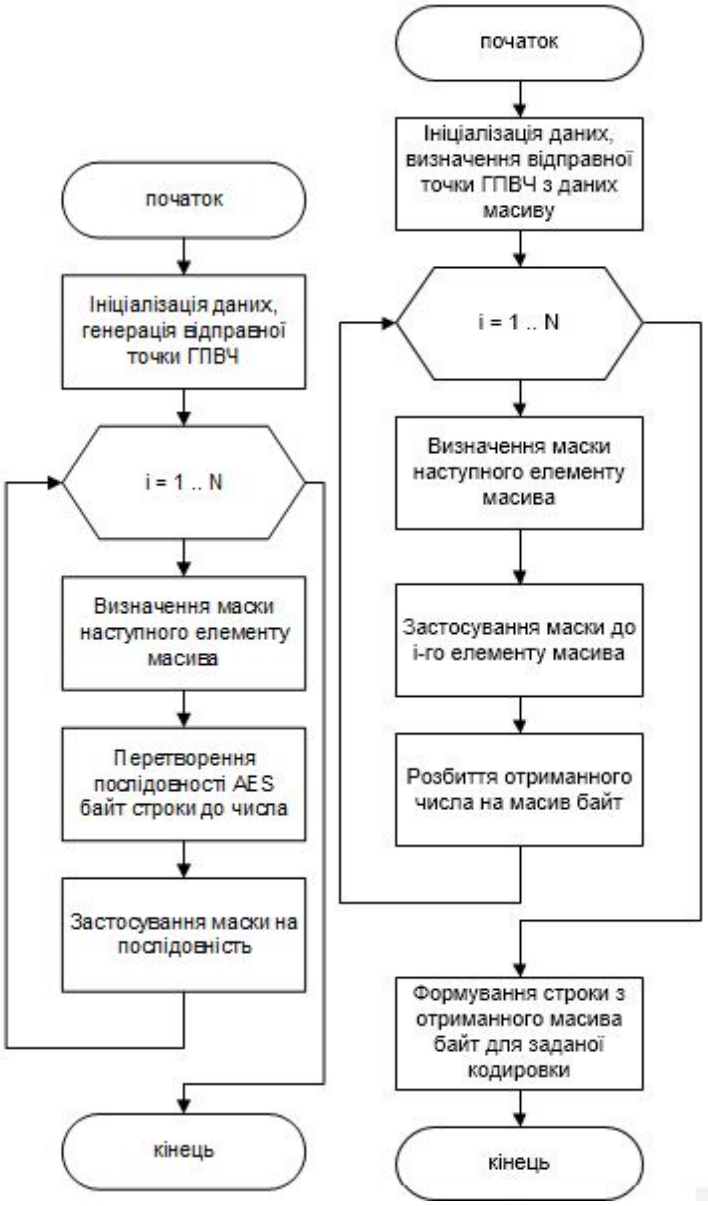
Запропонований метод полягає в особливостях роботи генератора псевдовипадкових чисел (ГПВЧ): при одному і тому ж початковому значенні зерна (*seed*) послідовність чисел завжди виходить однакова. В результаті роботи методу виходить масив чисел, який відображає масив, і завдяки симетричності алгоритму може бути назад перетворений в рядок.

В сучасних операційних системах найбільший за розміром і поширенням тип даних *long*, який займає в пам'яті 8 байт. В залежності від мовних особливостей тексту в наявному обсязі пам'яті можна зберігати до 8 символів (за умови, що всі вони будуть однобайтові, наприклад, символи латинського алфавіту, цифри). При використанні кодування *UTF-8*, кількість символів, які можна записати у 8 байт, зменшується.

Для того, щоб перетворити рядок у число, необхідна кількість змінних типу *long* у 8 разів менша, ніж обсяг рядка в байтах.

У зв’язку з тим, що метод, який розроблюється, базується на початковому значенні для функції генерації псевдовипадкових чисел, то результуючий масив буде містити на одне значення більше очікуваного.

Алгоритм обфускації. Схематичний опис алгоритму обфускації строкових літералів на основі особливостей роботи ГПВЧ наведено на рис. 3.1а.



а) б)

Рис. 3.1. Алгоритм обфускації (а) та деобфускації (б) строкових літералів.

Крок 1. Ініціалізація даних. На даному етапі виділяється пам’ять під результуючий масив згідно до формули:

$$cnt = GVS + (Size / AES) + ((Size\%AES == 0)\,?\,0 : 1),$$

де *Size* – розмір початкового рядка в байтах;

GVS (*Generated Value Size*) – кількість елементів масиву, які будуть займати значення відправної точки генератора псевдовипадкових чисел (в поданому прикладі – 1 елемент);

AES (*Array Element Size*) – розмір одного елемента результуючого масиву в байтах (для типу *long* – 8 байт).

Також генерується псевдовипадкове число, яке буде «відправною» точкою для роботи генератора псевдовипадкових чисел. Дане число генерується в залежності від поточного часу, а це означає, що дві вихідних послідовності одного і того ж рядка будуть повністю відрізнятися.

Перші *GVS* елементів масиву заповнюються псевдовипадковими числами – відправною точкою ГПВЧ.

Крок 2. Заповнення елементів, що залишилися від вихідного масиву.

Крок 2.1. Для кожного наступного елемента масиву відбувається обчислення наступного псевдовипадкового числа, яке буде виступати в ролі «бітової маски» для приховування даних.

Крок 2.2. Перетворення наступних *AES* байт рядка в число шляхом подання їх як складових байтів числа. Наприклад, рядок *ABCD* перетворюється в число: 0x44434241.

Крок 2.3. Накладання маски, отриманої на кроці 2.1, на число, отримане на кроці 2.2.

Алгоритм деобфускації. Для проведення процедури деобфускації з метою відновлення рядків використовується послідовність дій, описана на рис. 3.1(б). Даний алгоритм можна охарактеризувати наступною послідовністю дій.

Крок 1. Ініціалізація даних. На даному етапі виділяється пам’ять під результуючий рядок (в байтах) згідно до формули:

$$Size = (cnt - GVS) * GVS,$$

де *GVS* (*Generated Value Size*) – кількість елементів масиву, які будуть приймати значення відправної точки генератора псевдовипадкових чисел (в поданому прикладі – 1 елемент);

AES (*Array Element Size*) – розмір одного елемента результуючого масиву в байтах (для типу *long* - 8 байт);

cnt - кількість елементів у вихідному масиві.

ГПВЧ ініціалізується згідно до відправної точки, яка розраховується з даних перших *GVS* елементів масиву. Так, в поданому прикладі, 0-й елемент масиву і є відправною точкою.

Крок 2. Формування масиву байтів результуючого рядка.

Крок 2.1. Для кожної наступної послідовності з *AES* байт формується "маска" згідно до алгоритму роботи ГПВЧ.

Крок 2.2. Маска накладається на послідовність з необроблених *AES* байт. В результаті виходить масив, аналогічний отриманому в результаті роботи кроку 2.2 процесу обфускації.

Крок 2.3. Розбиття числа розміром *AES* байт на *AES* окремих байт і додавання їх в результуючий масив байт.

Крок 3. Формування рядку з масиву байт відповідно до обраного кодування. У засобу перевірки правопису *UTF-8* немає чіткої відповідності «1 байт = 1 символ», тому розмір результуючого рядка може бути менше.

Розвитком даного алгоритму є модифікація вихідної послідовності не як масиву чисел (2,4,8-байтних), а як байтового масиву або *BigInteger* числа. Використовуючи даний підхід, необхідність вирівнювання буде усунена, що дає можливість зробити значення відправної точки ГПВЧ практично необмеженої довжини (в тому числі і для використання власного ГПВЧ), а також використання маски, розмір якої не є кратним до обраного типу

елемента масиву (*AES*) – так, при типі елемента результуючого масиву *long*, кожен елемент буде займати 8 байт, і для узгодження вирівнювання необхідно, щоб довжина маски становила 1,2,4,8 байт.

Наступним розвитком запропонованого алгоритму є використання перетворення числових значень на рядок і повторення алгоритму задану кількість разів.

3.3.2 Розробка алгоритму обфускації імен ідентифікаторів

Історично, обфускація імен ідентифікаторів використовувалася при виробництві *J2ME*-застосунків для зменшення обсягу результуючого програмного застосунку шляхом скорочення будь-яких ідентифікаторів до 1-2-буквенних назв.

Так як сучасні компілятори підтримують імена ідентифікаторів в *UTF-8* форматі, розвитком обфускації ідентифікаторів було скорочення імен ідентифікаторів до одного *UTF-8* символу, що візуально «читалося» гірше. Наприклад символ з кодом \u0001 не має репрезентаційного символу, що погіршує сприймання коду, так як зловмисник не бачить назву змінної. Однак, даний підхід до обфускації імен ідентифікаторів мав недолік - була можливість взяти декомпільований код, внести в нього зміни (зокрема, заміну всіх ідентифікаторів з кодом \u0001 на щось більш логічне) і без труднощів перекомпілювати. Розвитком даного підходу стало перейменування імен ідентифікаторів в ключові слова мови, такі як "do", "while", "for", які перешкоджають перекомпіляції.

Аналіз описаних реалізацій обфускації імен ідентифікаторів показав їх спільний недолік – в рамках області видимості вони мають одну й ту саму назву, що дає можливість простежити набір дій, які виконуються з одним і тим же ідентифікатором.

Мови програмування не дають можливості конструювання імені ідентифікатора – воно повинно бути відомо і повноцінно на етапі компіляції.

В рамках підвищення рівня обфускації коду програми пропонується використання глобального асоціативного масиву, де ключ – рядок – ім’я ідентифікатора. Це дає нам можливість отримувати доступ на читання і запис елемента асоціативного масиву по ключу, який буде генеруватися «на льоту», що ускладнить зловмиснику можливість простежити хід програми без відлагодження.

Наприклад, конструкція

```
int value = 10;
System.out.println(value);
```

при переході на асоціативний масив перетвориться в конструкцію:

```
private static final Map<String, Object> GLOBAL_VARIABLES
= new HashMap()<>;
// ...
GLOBAL_VARIABLES.put("value", 10);
System.out.println(GLOBAL_VARIABLES.get("value");
```

Цей варіант використання може бути ще більше обфускований, якщо виконати обфускацію строкових літералів з використанням особливостей генератора псевдовипадкових чисел:

```
GLOBAL_VARIABLES.put(ConvertString(new long[] {1256, 6521}), 10);
System.out.println(GLOBAL_VARIABLES
.get(ConvertString(new long[] {126, 321}));
```

Слід зазначити, що глобальний асоціативний масив має таку особливість: всі ключі даного масиву повинні бути унікальними, а при розробці програмного забезпечення:

- в різних методах і класах можуть бути присутніми ідентифікатори з однаковим ім’ям;
- клас може мати кілька екземплярів, а отже, необхідно ідентифікувати екземпляр класу;
- функції, в яких використовуються змінні, можуть бути перевантажені.

У зв’язку з вищезазначеними обмеженнями, було прийнято рішення використовувати наступний формат ідентифікатора:

classname @ hash:global_class_id

де *classname* - повне ім’я класу (разом з пакетом);

hash - геш поточного екземпляру класу. У разі, якщо ідентифікатор глобальний, - значення порожнє;

global_class_id - унікальний ідентифікатор в контексті поточного класу. Це дає можливість уникнути необхідності визначення області видимості і наявності перевантажених функцій.

Алгоритм заплутування імен ідентифікаторів таким чином можна представити схемою алгоритму, представленою на рис. 3.2.

Рис. 3.2. Схема алгоритму обфускації імен ідентифікаторів

3.3.3 Розробка алгоритму обфускації логічних виразів

В рамках дослідження даного розділу було запропоновано використовувати особливості спрощення логічних виразів. Так, розглянуті наступні конструкції спрощення:

- використання тернарного оператора: вираз

$$\left[(a)?1:0\right]!=0$$

може бути згорнуто в вираз (a), при цьому, виділений фрагмент в квадратних дужках може повторюватися нескінченну кількість разів, підставляючи себе замість числа 1, що в даному випадку, призведе до того, що умова з таким тернарним оператором буде завжди "правда". Таким чином, введення додаткового тернарного оператора в логічне вираження введе додаткове ускладнення в аналіз коду.

- використання особливостей «лінивих» логічних виразів. У сімействі мов Ci (Ci, C ++, Java, C #) існує особливість логічних виразів, що дозволяє прискорити роботу програми в окремих випадках:
  - у виразі (a && b) умова b, яке може включати в себе комплексну умову або навіть виклик функції, не буде викликатися якщо a прийме значення БРЕХНЯ (так як в цьому випадку значення умови b не грає роль – результат вираження буде БРЕХНЯ)
  - у виразі (a || b) умова b, яке може включати в себе комплексне умова або навіть виклик функції не буде викликатися якщо a прийме значення ПРАВДА (так як в цьому випадку значення умови b не грає роль – результат вираження буде ПРАВДА)
- використання законів спрощення логічних операцій:
  - подвійного заперечення. Формула еквівалентності має вигляд:

$$(a)===(!!a)$$

  - виключеного третього. Формули еквівалентності мають вигляд:

$$false===(a\&\&!a)$$

$$true === (a \,||\, !a)$$

– роботи з константами. Формули еквівалентності мають вигляд:

$$(a) === (a \,\&\&\, true),$$

$$false === (a \,\&\&\, false),$$

$$true === (a \,||\, true),$$

$$(a) === (a \,||\, false)$$

– повторення. Формула еквівалентності мають вигляд:

$$(a) === (a \,\&\&a) === (a \,||\, a)$$

– поглинання. Формули еквівалентності мають вигляд:

$$(a) === (a \,||\, (a \,\&\&b))$$

$$(a) === (a \,\&\&\, (a \,||\, b))$$

3.4 Синтез комплексу алгоритмів процесу обфускації / деобфускації програмних модулів

У рамках дослідження було досліджено декомпільований байт-код існуючих програмних продуктів, що використовують для свого захисту процеси обфускації програмних модулів. На основі проведених досліджень були розроблені алгоритми обфускації програмних модулів, представлені на рис. 3.1 та рис. 3.2.

Згідно до поданих алгоритмів, можна описати процес обфускації вихідного коду наступними кроками.

Крок 1. Початковий стан. Є необроблений вихідний код на мові високого рівня, написаного з використанням віртуальної машини (JVM, CLI та ін.). У зв'язку з тим, що скомпільований код має ряд проблем з модифікацією, то процес видозміни коду і його обфускації відбувається, базуючись на «сирому» (вихідному) коді. У зв'язку з тим, що в даній роботі використовується комбінований підхід двох незалежних методів обфускації,

першим кроком може бути або Крок 2, або Крок 4, або Крок 5.

Крок 2. Вихідний код проходить через метод обфускації, заснований на видозміні строкових літералів. Видозмінений вихідний код не матиме сенсу без зворотного алгоритму Кроку 3.

Крок 3. У зв'язку з тим, що алгоритм перетворення строкових літералів симетричний, всередину вихідного коду додається функція зворотного перетворення. Алгоритм зворотного перетворення Кроку 2 є невід'ємним доповненням, однак, його порядок додавання не грає ролі.

Крок 4. Вихідний код проходить через метод обфускації, заснований на «розплутуванні» конструкцій, які містять булеві операції (виконуються операції, зворотні спрощенню). Даний метод не має залежностей і може виконуватися на будь-якому етапі / виконуватися в залежності від правила комбінації.

Крок 5. Вихідний код проходить через метод обфускації, заснований на обфускації імен ідентифікаторів. Даний метод не має залежностей і може виконуватися на будь-якому етапі / виконуватися в залежності від правила комбінації. Даний метод має ряд опцій, що дозволяють обфускувати: локальні змінні, глобальні змінні, функції, класи.

Крок 6. У результаті виконання Кроків 2-5 вихідний код готовий до процесу компіляції. При цьому, для виключення побічних ефектів, необхідно упевнитися в тому, що компілятор не виконуватиме передчасну оптимізацію коду. Також, для зниження читання коду і ускладнення процесу відлагодження, необхідно відключити відлагодження. Так, наприклад, для мов, заснованих на віртуальній машині Java, компіляція з атрибутом «-g: none» відключає інформацію про відлагодження. А опція компілятора «-Xint» відключає Just-In-Time і Ahead-Of-Time компіляції, що призводять до оптимізації коду компілятором.

Згідно до поданих матеріалів, розроблені:

– загальний алгоритм обфускації програмного модуля, що представлений на рис. 3.3;

– GERT-мережа процесів обфускації і деобфускації програмних модулів, що відображена на рис. 3.4.

На рис. 3.4 і відповідній табл. 3.1, на основі розроблених алгоритмів рис. 3.1 та рис. 3.2, сформульовані переходи між станами, які характеризують:

– (1, 2): обробка вихідного коду шляхом кодування строкових літералів, вбудовування в вихідний код функції, яка «на льоту» декодує змінені рядкові літерали в початковий стан;

– (2, 3): після успішного кодування строкових літералів виконуємо процес обфускації булевих операцій, виконання верифікації успішності перетворення булевих операцій;

– (3, 4): після успішної обфускації булевих операцій виконуємо процес обфускації імен ідентифікаторів, виконання верифікації успішності перетворення імен ідентифікаторів;

– (4, 5): після успішної обфускації імен ідентифікаторів переходимо до фінального стану, що готовий до компіляції;

– (1, 3): в залежності від бізнес-сценарію, пропускаємо обробку вихідного коду шляхом кодування строкових літералів і виконуємо процес обфускації булевих операцій;

– (1, 4): в залежності від бізнес-сценарію, пропускаємо обробку вихідного коду шляхом кодування строкових літералів і обфускації булевих операцій. Виконуємо процес обфускації імен ідентифікаторів;

– (2, 4): в залежності від бізнес-сценарію, пропускаємо обробку вихідного коду шляхом кодування обфускації булевих операцій. Виконуємо процес обфускації імен ідентифікаторів;

– (3, 5): в залежності від бізнес-сценарію, пропускаємо обробку вихідного коду шляхом обфускації імен ідентифікаторів;

– (2, 5): в залежності від бізнес-сценарію, пропускаємо обробку вихідного коду шляхом обфускації булевих операцій та імен ідентифікаторів;

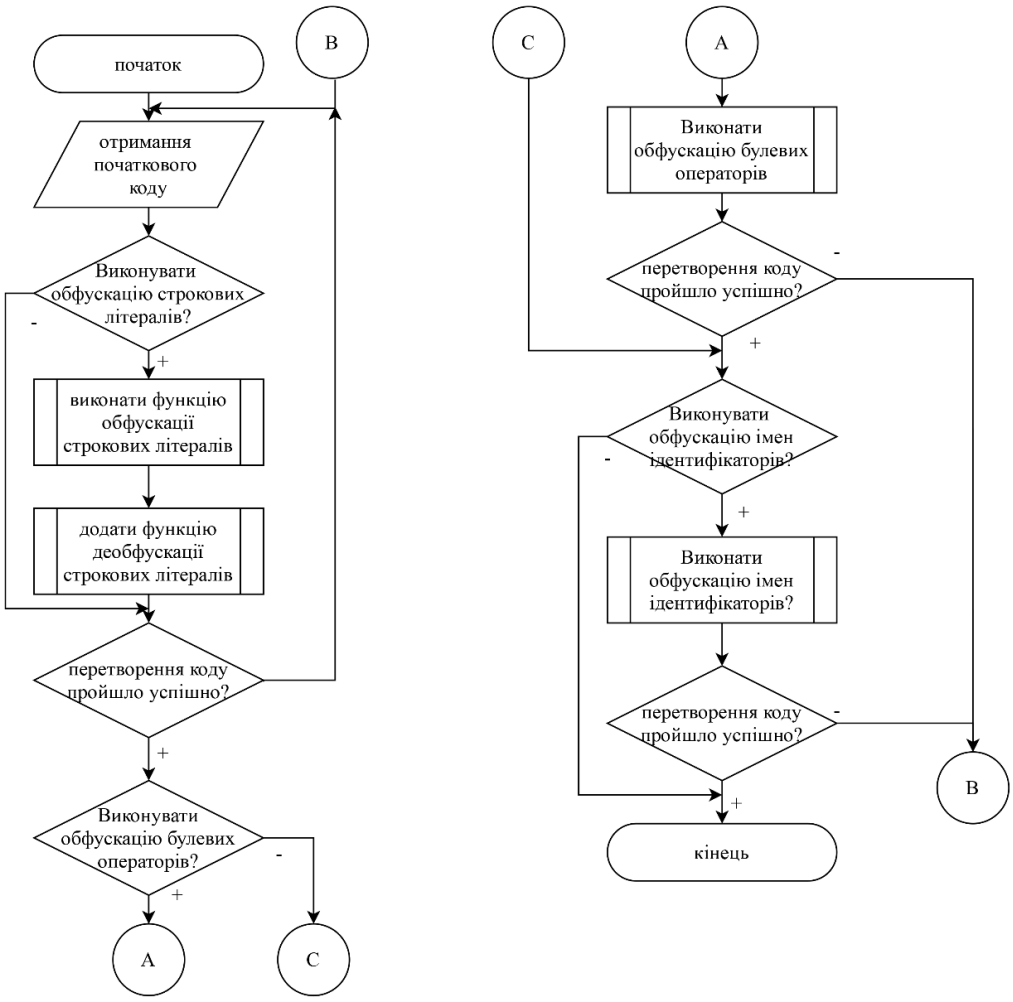


Рис. 3.3. Розроблений алгоритм обфускації програмного модуля

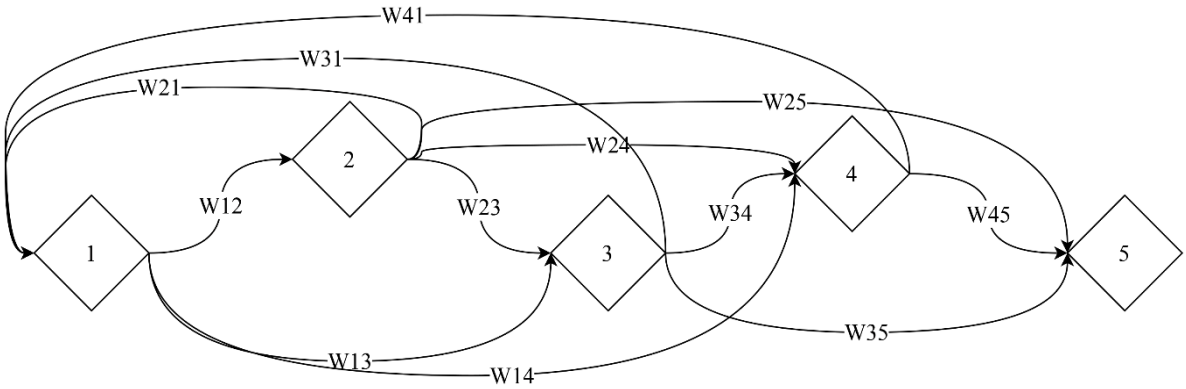


Рис. 3.4. Розроблена GERT-мережа процесів обфускації і деобфускації програмних модулів

– (2, 1): в процесі кодування шляхом видозміни строкових літералів і додавання функції декодування виникла помилка. Повертаємося на початковий стан з метою повторити спробу обфускації;

– (3, 1): в процесі верифікації успішності перетворення булевих операцій виникла помилка. Повертаємося на початковий стан (так як невідома причина виникнення помилки - проблема обфускації булевих операцій або помилка викликана видозміною строкових літералів на попередніх кроках) з метою повторити спробу обфускації;

– (4, 1): в процесі верифікації успішності перетворення імен ідентифікаторів виникла помилка. Повертаємося на початковий стан (так як невідома причина виникнення помилки – проблема обфускації імен ідентифікаторів, проблема обфускації булевих операцій або помилка викликана видозміною строкових літералів на попередніх кроках) з метою повторити спробу обфускації.

Таким чином, синтезовано комплекс алгоритмів обфускації і деобфускації програмних модулів, що дозволило комплексно описати дані процеси на верхньому стратегічному рівні формалізації.

3.5 Розробка GERT-моделі процесу обфускації програмних модулів на основі розроблених алгоритмів

Проведені дослідження показали [82, 171, 178, 235], що загальний алгоритм обфускації програмних модулів має ряд специфічних ітерацій, які в значній мірі ускладнюють загальний процес його математичної формалізації. Тому вважаємо за доцільне розбиття цього процесу на ряд підпроцесів. При цьому, для математичного моделювання процесів обфускації і деобфускації найбільш гнучкими і корисними видаються мережеві стохастичні моделі. Окремим випадком стохастичної моделі є GERT-мережа.

Багато в чому це пов’язано з доступністю математичного апарату

знаходження безперервної щільності розподілу ймовірностей часу проходження GERT-мережі. Однак, це можливо тільки за умови, що множина розподілів, якими можуть характеризуватися окремі дуги моделі, включає в себе відомі розподіли. До них можна віднести такі: дискретний, біноміальний, пуасоновський, геометричний, від'ємний біноміальний, рівномірний, експоненціальний, гамма і нормальний.

Крім цього, існує можливість знаходження і використання безперервних розподілів довільного виду. В роботі [95] показано, що щільність розподілу ймовірностей часу проходження GERT-мережі визначається наступним виразом:

$$\phi(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{-isx} W_E(s)\,\mathrm{d}s, \tag{3.1}$$

де $W_E(s)$ – еквівалентна передатна функція GERT-мережі, s – дійсна змінна.

З топологічного рівняння [144] випливає:

$$W_E(t) = \frac{\sum_{\gamma_1=1}^{\Gamma_1}\prod_{\delta_1=1}^{\Delta_1}\breve{W}^1_{\gamma_1\delta_1}(t) + \ldots + (-1)^m \sum_{\gamma_m=1}^{\Gamma_m}\prod_{\delta_m=1}^{\Delta_m}\breve{W}^m_{\gamma_m\delta_m}(t)}{1 - \sum_{a_1=1}^{A_1}\prod_{\beta_1=1}^{B_1}\bar{W}^1_{a_1\beta_1}(t) + \ldots + (-1)^l \sum_{a_l=1}^{A_l}\prod_{\beta_l=1}^{B_l}\bar{W}^l_{a_l\beta_l}(t)}, \tag{3.2}$$

де $\prod_{\delta_i=1}^{\Delta_i}\breve{W}^i_{\gamma_i\delta_i}$ – W-добуток дуг γ_i-ої петлі i-го порядку, що включає до себе стік t, $1 \le i \le m$;

$\prod_{\beta_j=1}^{B_j}\bar{W}^j_{a_j\beta_j}$ – W-добуток дуг a_j-ої петлі j-го порядку, що не включає до себе стік t, $1 \le i \le l$;

Γ_i – число петель порядків i, що включають до себе стік мережі;

A_i – число петель порядків i, що не включають до себе стік мережі.

У ряді практично важливих випадків розподіли необхідно отримувати у вигляді математичних виразів. До таких завдань можна віднести і завдання дослідження алгоритмів обфускації і деобфускації програмного коду. Дане завдання зводиться до знаходження щільності розподілу випадкової величини часу проходження, сформованої на основі розробленої GERT-

мережі. Слід зазначити, що необхідно ввести допущення, що в безперервній щільності розподілу ймовірностей часу проходження GERT-мережі φ(x) визначається виразом (3.1).

W-функція переходу між *i*, *j* визначається формулою:

$$W_{ij}(x) = P_{ij} \int_{-\infty}^{\infty} e^{isx} \zeta_{ij}(x) \mathrm{d}x, \tag{3.3}$$

де $\zeta_{ij}(x)$ – щільність ймовірності переходу між станами *i*, *j*;

P_{ij} – ймовірність переходу зі стану *i* у стан *j*.

В рамках дослідження приймемо гіпотезу – використання гамма-розподілу при моделюванні в якості ключового при описі ймовірнісних переходів зі стану в стан, дозволить досягти уніфікації моделі процесу обфускації програмних модулів. Уніфікація полягає в тому, що зменшення або збільшення кількості операцій обфускації незначно змінить результати моделювання. При цьому очікується, що при зменшенні кількості вузлів дисперсія і математичне сподівання будуть знижуватися незначно, а при збільшенні – відповідно, збільшуватися. Однак накладаються обмеження на зміну моделі – структурна архітектура моделі (наприклад, ступінь зв'язності вузлів) повинна залишитися незмінною.

Таким чином, в даній GERT-мережі процесів обфускації і деобфускації програмних модулів функції щільності ймовірності переходів визначимо гамма-розподілом зі змінними коефіцієнтами *k* та θ:

$$\zeta(x) = \frac{x^{k-1} \cdot e^{\frac{-x}{\theta}}}{\theta^k \cdot \Gamma(k)}. \tag{3.4}$$

Результуюча *W*-функція має наступний вигляд:

$$W_E(s) = \frac{\bar{W}_1 + \bar{W}_2 + \bar{W}_3 + \bar{W}_{12} + \bar{W}_{13} + \bar{W}_{23} + \bar{W}_{123}}{1 - \left(\bar{W}_{1f} + \bar{W}_{2f} + \bar{W}_{3f} + \bar{W}_{12f} + \bar{W}_{13f} + \bar{W}_{23f} + \bar{W}_{123f}\right)}, \tag{3.5}$$

де:

$\bar{W}_1$ – добуток *W*-функцій, що описують успішне виконання тільки алгоритму обфускації строкових літералів:

$$\bar{W}_1 = W_{12}W_{25}, \tag{3.6}$$

$\bar{W}_{1f}$ – добуток W-функцій, що описують неуспішне виконання тільки алгоритму обфускації строкових літералів:

$$\bar{W}_{1f} = W_{12}W_{21}, \tag{3.7}$$

$\bar{W}_2$ – добуток W-функцій, що описують успішне виконання тільки алгоритму обфускації булевих функцій:

$$\bar{W}_2 = W_{13}W_{35}, \tag{3.8}$$

$\bar{W}_{2f}$ – добуток W-функцій, що описують неуспішне виконання тільки алгоритму обфускації булевих функцій:

$$\bar{W}_{2f} = W_{13}W_{31}, \tag{3.9}$$

$\bar{W}_3$ – добуток W-функцій, що описують успішне виконання тільки алгоритму обфускації імен ідентифікаторів:

$$\bar{W}_3 = W_{14}W_{45}, \tag{3.10}$$

$\bar{W}_{3f}$ – добуток W-функцій, що описують неуспішне виконання тільки алгоритму обфускації імен ідентифікаторів:

$$\bar{W}_{3f} = W_{14}W_{41}, \tag{3.11}$$

$\bar{W}_{12}$ – добуток W-функцій, що описують успішне послідовне виконання алгоритму обфускації строкових літералів і булевих функцій:

$$\bar{W}_{12} = W_{12}W_{23}W_{35}, \tag{3.12}$$

$\bar{W}_{12f}$ – добуток W-функцій, що описують послідовне виконання алгоритму обфускації строкових літералів і булевих функцій:

$$\bar{W}_{12f} = W_{12}W_{23}W_{31}, \tag{3.13}$$

$\bar{W}_{13}$ – добуток W-функцій, що описують успішне послідовне виконання алгоритму обфускації строкових літералів і імен ідентифікаторів:

$$\bar{W}_{13} = W_{12}W_{24}W_{45}, \tag{3.14}$$

$\bar{W}_{13f}$ – добуток W-функцій, що описують послідовне виконання алгоритму обфускації строкових літералів і імен ідентифікаторів:

$$\bar{W}_{13f} = W_{12}W_{24}W_{41}, \tag{3.15}$$

$\bar{W}_{23}$ – добуток *W*-функцій, що описують успішне послідовне виконання алгоритму обфускації булевих функцій та імен ідентифікаторів:

$$\bar{W}_{23} = W_{13}W_{34}W_{45}, \tag{3.16}$$

$\bar{W}_{23f}$ – добуток *W*-функцій, що описують послідовне виконання алгоритму обфускації булевих функцій та імен ідентифікаторів:

$$\bar{W}_{23f} = W_{13}W_{34}W_{41}, \tag{3.17}$$

$\bar{W}_{123}$ – добуток *W*-функцій, що описують успішне послідовне виконання алгоритму обфускації строкових літералів, булевих функцій та імен ідентифікаторів:

$$\bar{W}_{123} = W_{12}W_{23}W_{34}W_{45}, \tag{3.18}$$

$\bar{W}_{123f}$ – добуток *W*-функцій, що описують послідовне виконання алгоритму обфускації строкових літералів, булевих функцій та імен ідентифікаторів.

$$\bar{W}_{123f} = W_{12}W_{23}W_{34}W_{41}, \tag{3.19}$$

Сформована таблиця характеристик гілок розглянутих в GERT-моделі гілок і параметрів розподілу наведена в табл. 3.1.

Як видно з виразу (3.4), математична формалізація результуючої еквівалентної функції моментів представляється громіздким вираженням. У зв'язку з цим, виникає завдання узагальнення математичної формалізації результуючих виразів розрахунку еквівалентних передавальної функцій.

Підставляючи значення виразів (3.6) – (3.19) в результуючу *W*-функцію (3.5), формула виходить значних розмірів. Для її «нормалізації», введемо наступну заміну:

$$p_{\bar{b}}^{\bar{a}}(t) = P_{a_1}\int_{-\infty}^{\infty} e^{isx}\frac{x^{k_{b_1}-1}\cdot e^{\frac{-x}{\theta_{b_1}}}}{\theta_{b_1}^{k_{b_1}}\cdot\Gamma\left(k_{b_1}\right)}\mathrm{d}x \times P_{a_2}\int_{-\infty}^{\infty} e^{isx}\frac{x^{k_{b_2}-1}\cdot e^{\frac{-x}{\theta_{b_2}}}}{\theta_{b_2}^{k_{b_2}}\cdot\Gamma\left(k_{b_2}\right)}(x)\mathrm{d}x\cdot\ldots\times P_{a_n}\int_{-\infty}^{\infty} e^{isx}\frac{x^{k_{b_n}-1}\cdot e^{\frac{-x}{\theta_{b_n}}}}{\theta_{b_n}^{k_{b_n}}\cdot\Gamma\left(k_{b_n}\right)}(x)\mathrm{d}x, \tag{3.20}$$

де $p_{\bar{b}}^{\bar{a}}(t)$ відображає добуток *W*-функції, описують успішне (і неуспішне) виконання алгоритмів, описаних змінними виразів (3.6) – (3.19);

$\bar{a} = (a_1, a_2, a_3, .., a_n)$ – перелік (масив) коефіцієнтів ймовірностей переходів (дуг петель),

$\bar{b} = (b_1, b_2, b_3, .., b_n)$ – перелік (масив) коефіцієнтів відповідної виробляючої функції моментів переходу.

Таблиця 3.1. Характеристики переходів між станами GERT-мережі процесів обфускації і деобфускації програмних модулів

№ з/п	Гілка	*W*-функція	Ймовірність переходу	Коефіцієнти щільності ймовірності (3.3)
1.	(1, 2)	W_{12}	P_1	$k=k_1$; $\theta=\theta_1$
2.	(2, 3)	W_{13}	P_2	$k=k_1$; $\theta=\theta_1$
3.	(1, 4)	W_{14}	$P_3=1-P_1-P_2$	$k=k_1$; $\theta=\theta_1$
4.	(2, 4)	W_{24}	P_4	$k=k_2$; $\theta=\theta_2$
5.	(2, 4)	W_{23}	P_5	$k=k_1$; $\theta=\theta_1$
6.	(3, 4)	W_{34}	P_8	$k=k_1$; $\theta=\theta_1$
7.	(4, 5)	W_{45}	P_{11}	$k=k_1$; $\theta=\theta_1$
8.	(3, 5)	W_{35}	P_9	$k=k_2$; $\theta=\theta_2$
9.	(2, 5)	W_{25}	P_6	$k=k_2$; $\theta=\theta_2$
10.	(4, 1)	W_{41}	$P_{12}=1-P_{11}$	$k=k_3$; $\theta=\theta_3$
11.	(3, 1)	W_{31}	$P_{10}=1-P_8-P_9$	$k=k_3$; $\theta=\theta_3$
12.	(3, 2)	W_{32}	$P_7=1-P_4-P_5-P_6$	$k=k_3$; $\theta=\theta_3$

Таким чином, результуючий вираз розрахунку еквівалентних передавальних функцій можна описати як:

$$W_E(t) = \frac{\left(p_{(1,2)}^{(1,6)}(t) + p_{(1,2)}^{(2,9)}(t) + p_{(1,1)}^{(3,1)}(t) + p_{(1,2,1)}^{(1,4,1)}(t) + p_{(1,1,2)}^{(1,5,9)}(t) + p_{(1,1,1)}^{(2,8,11)}(t) + p_{(1,1,1,1)}^{(1,5,8,11)}(t)\right)}{1-\left(p_{(1,3)}^{(1,7)}(t) + p_{(1,3)}^{(1,10)}(t) + p_{(1,3)}^{(1,12)}(t) + p_{(1,2,3)}^{(1,4,12)}(t) + p_{(1,1,3)}^{(1,5,10)}(t) + p_{(1,1,3)}^{(2,8,12)}(t) + p_{(1,1,1,3)}^{(1,5,8,12)}(t)\right)}. \quad (3.21)$$

Використовуючи щільність розподілу ймовірностей (3.4), отримуємо графік розподілу щільності ймовірності, відображений на рис. 3.5. При цьому, ймовірно вибрали такий спосіб:

$$P_{1=}P_{2=}P_3 = 1/3; P_{4=}0.05; P_{5=}0.8;$$
$$P_{6=}P_{12=}0.1; P_{7=}P_{10=}0.05; P_{8=}0.8;$$
$$P_{9=}0.15; P_{11=}0.9.$$

Мережа може бути побудована так, що якщо в деякому стані i можливий початок однієї з кількох наступних операцій, то ймовірності запуску p_{ij} будь-якої з цих операцій утворюють повну групу несумісних подій:

$$\sum_j p_{ij} = 1, \forall i. \tag{3.22}$$

У такому випадку ймовірність виконання всієї мережі від витоку до стоку дорівнює 1.

Щоб показати, що всі вузли задовольняють умові (3.22), розрахуємо ймовірність виконання всього процесу, яка розраховується за формулою:

$$p_E = \left.\frac{W_E(s)}{M_E(s)}\right|_{s=0} = W_E(0) = 1, \tag{3.23}$$

де $M_E(s)$ – твірна функція, при цьому:

$$M_E(s)\big|_{s=0} = \left.\int_{-\infty}^{\infty} e^{sx} f_E(x)\,\mathrm{d}x\right|_{s=0} = \int_{-\infty}^{\infty} f_E(x)\,\mathrm{d}x = 1, \tag{3.24}$$

де $f_E(x)$ – функція щільності розподілу.

На рис. 3.5 показаний графік щільності розподілу часу виконання всього процесу обфускації і деобфускації програмного модуля з урахуванням варіацій значень змінних k та θ. Інтегрувавши щільність розподілу ймовірностей, одержимо функцію розподілу, графік якої відображений на рис. 3.6.

Математичне сподівання і дисперсія отриманих функцій розраховуються відповідно за формулами:

$$\mu = W_E(t)\,\mathrm{d}t\,|_{t=0}, \tag{3.25}$$

$$\sigma^2 = W_E(t)\,\mathrm{d}^2 t\,|_{t=0} - \mu^2. \tag{3.26}$$

Отримані результати розрахунків описані в табл. 3.2.

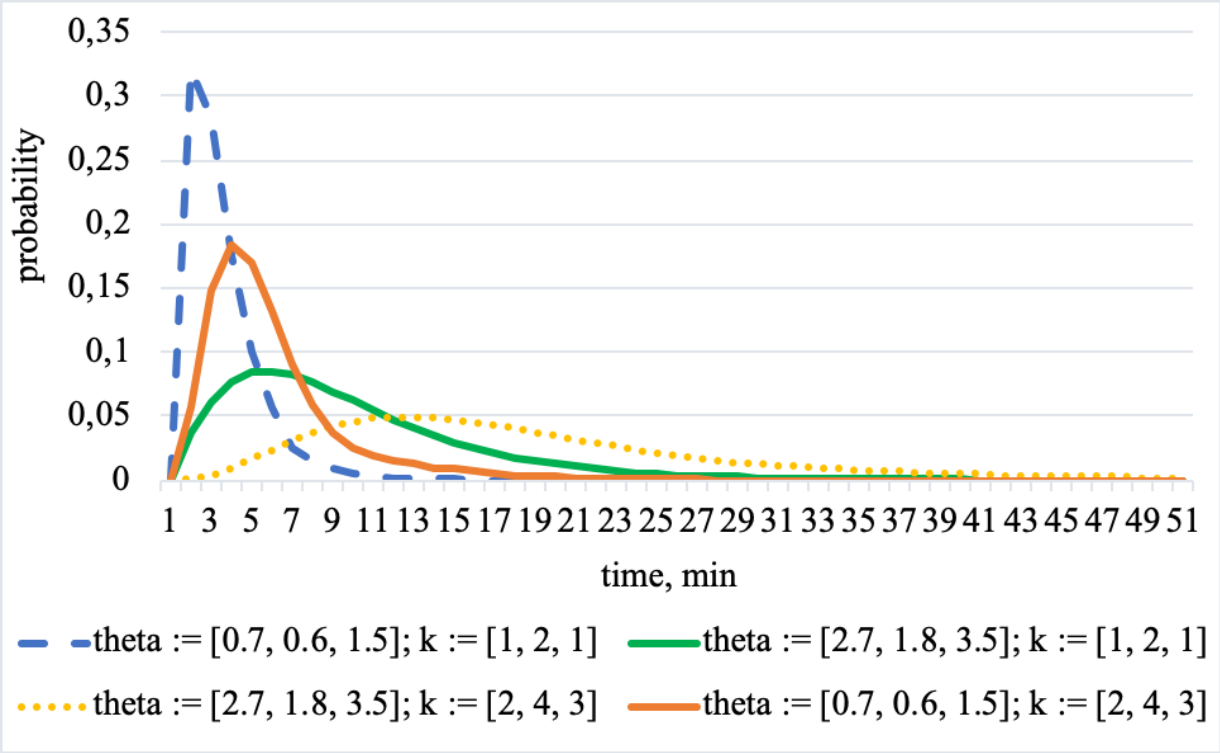


Рис. 3.5. Графіки щільності розподілу часу виконання процесу обфускації і деобфускації програмних модулів при використанні GERT-мережі, гамма-розподілу ймовірностей переходів з різними значеннями коефіцієнтів k та θ

Таким чином, в рамках дослідження була розроблена уніфікована GERT-модель процесу обфускації програмних модулів. Дана модель відрізняється від відомих реалізацією парадигми використання математичного апарату гамма-розподілу в якості ключового на всіх етапах моделювання процесу обфускації. Це дозволило досягти уніфікації моделі в умовах модифікації GERT-мережі.

Таблиця 3.2. Характеристики щільності ймовірності – математичне сподівання та дисперсія

№ з/п	k	θ	μ	σ^2
1.	[1, 2, 1]	[0.7, 0.6, 1.5]	2.42	3.4
2.	[1, 2, 1]	[2.7, 1.8, 3.5]	8.8	39.51
3.	[2, 4, 3]	[2.7, 1.8, 3.5]	18.19	133.54
4.	[2, 4, 3]	[0.7, 0.6, 1.5]	5.07	13.34

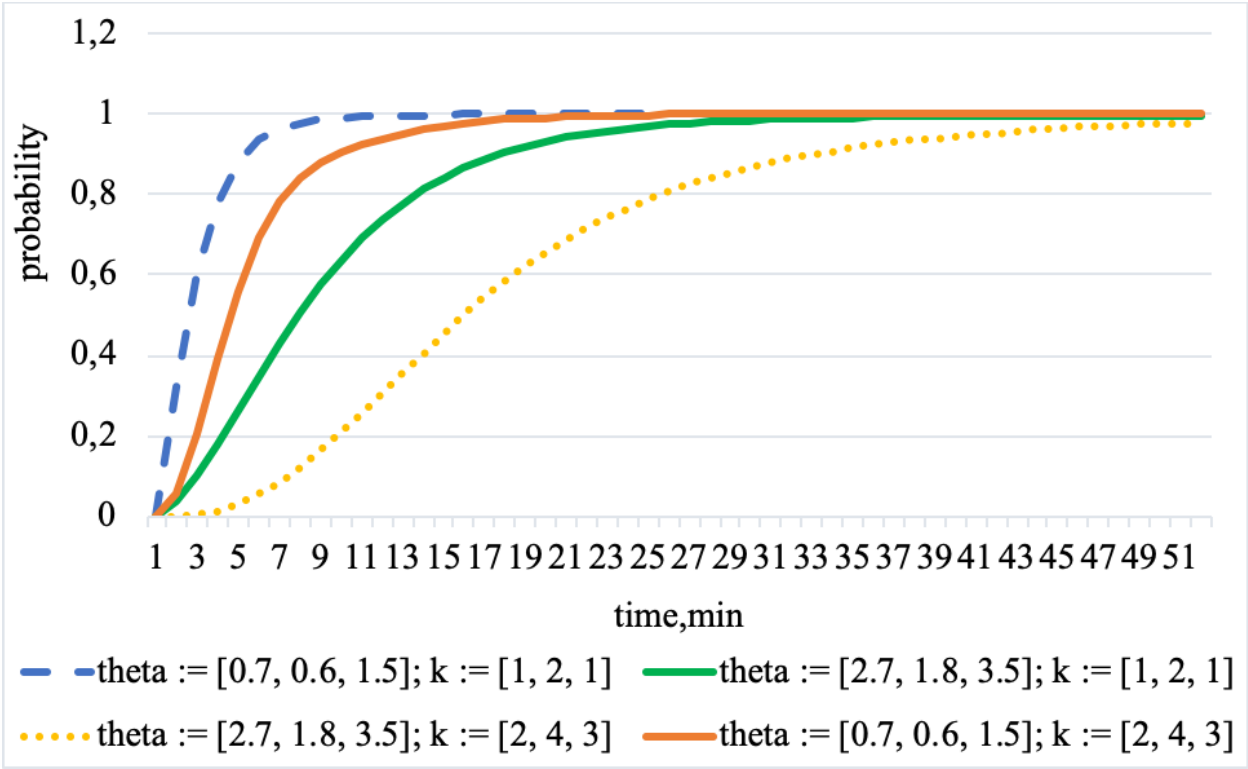


Рис. 3.6. Графіки функцій розподілу процесу обфускації і деобфускації програмних модулів при використанні GERT-мережі, гамма-розподілу ймовірностей переходів з різними значеннями коефіцієнтів k та θ.

3.5.1 Дослідження уніфікованої GERT-моделі зі зміненою кількістю вузлів

Розроблений процес обфускації і деобфускації програмних модулів складається з 5 вузлів. Розглянемо поведінку системи при зміні кількості вузлів.

Розроблені GERT-мережі процесів обфускації і деобфускації програмних модулів зі зміненою кількістю вузлів представлені на рис. 3.7, 3.8.

При зміні кількості вузлів враховувалися такі фактори:

– ступінь зв’язності вузлів нового процесу порівнянна з оригінальною;

– зміна складності процесу призводить до зміни кількості елементів масиву коефіцієнтів k, при цьому значення першого і останнього елементів

масиву k ідентичні оригінальним;

– зміна складності процесу призводить до зміни кількості елементів масиву коефіцієнтів θ, при цьому значення першого і останнього елементів масиву θ ідентичні оригінальним.

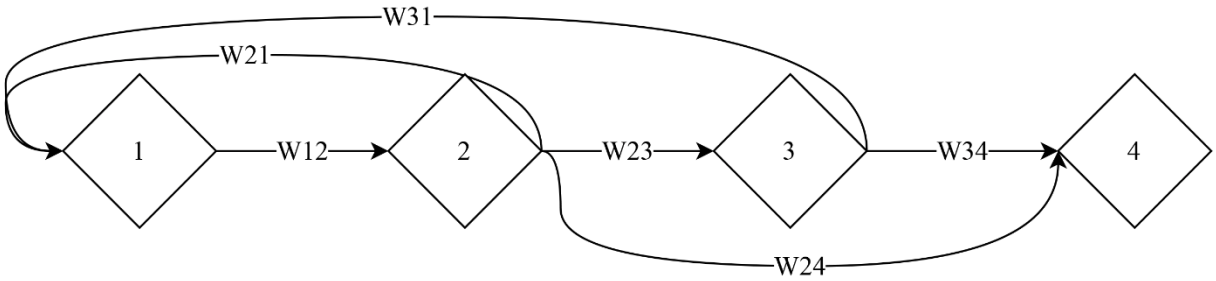


Рис. 3.7. Розроблена GERT-мережа процесів обфускації і деобфускації програмних модулів зі зменшеною кількістю оперованих функцій обфускації

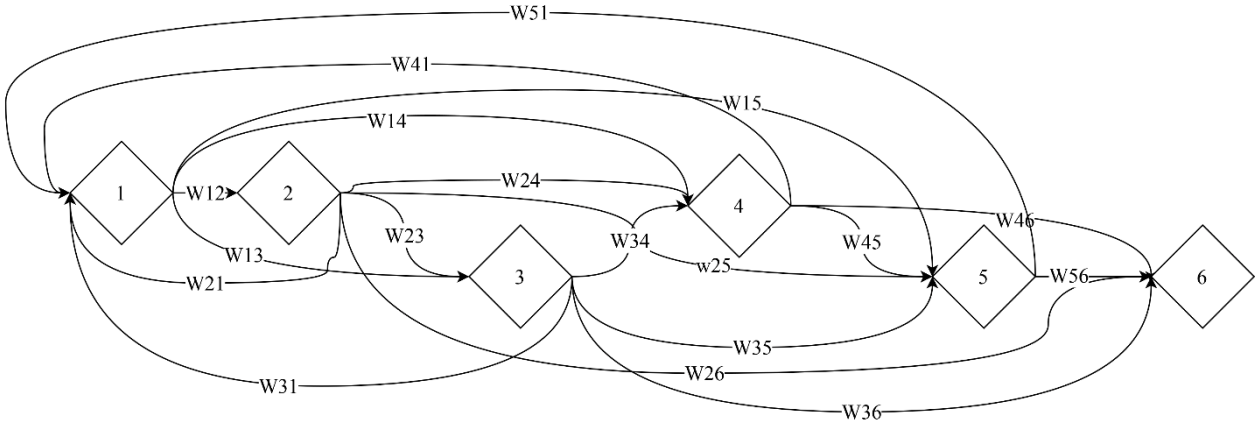


Рис. 3.8. Розроблена GERT-мережа процесів обфускації і деобфускації програмних модулів з доданою додатковою функцією обфускації

Сформовані таблиці характеристик гілок розглянутих в GERT-моделі гілок і параметрів розподілу представлені в табл. 3.3, 3.4.

Для кожного нового процесу була вирахувана ймовірність виконання всього процесу p_E, що дорівнює 1. Також були розраховані значення математичного сподівання і дисперсії, що представлені в табл. 3.5.

Таблиця 3.3. Характеристики переходів між станами GERT-мережі процесів обфускації і деобфускації програмних модулів зі зменшеною кількістю оперованих функцій обфускації

№ з/п	Гілка	*W*-функція	Ймовірність переходу	Коефіцієнти щільності ймовірності (3.3)
1.	(1, 2)	W_{12}	P_1	$k=k_1$; $\theta=\theta_1$
2.	(2, 3)	W_{13}	P_2	$k=k_1$; $\theta=\theta_1$
3.	(3, 4)	W_{34}	P_3	$k=k_2$; $\theta=\theta_2$
4.	(2, 4)	W_{23}	P_4	$k=k_3$; $\theta=\theta_3$
5.	(2, 1)	W_{41}	$P_5=1-P_2-P_4$	$k=k_3$; $\theta=\theta_3$
6.	(3, 1)	W_{31}	$P_6=1-P_3$	$k=k_3$; $\theta=\theta_3$

Використовуючи вираз (3.20), результуючі вираження розрахунку еквівалентних передавальних функцій можна описати як:

$$W_{E-1}(t)=\frac{p_{(1,2)}^{(1,4)}(t)+p_{(1,1,2)}^{(1,2,3)}(t)}{1-\left(p_{(1,3)}^{(1,5)}(t)+p_{(1,1,3)}^{(1,2,6)}(t)\right)}, \tag{3.28}$$

$$W_{E+1}(t)=\frac{\left(\begin{array}{l} p_{(3,2)}^{(9,5)}(t)+p_{(3,2)}^{(12,8)}(t)+p_{(3,2)}^{(14,7)}(t)+p_{(1,1,2)}^{(1,2,7)}(t)+ \\ +p_{(1,1,3,2)}^{(1,2,11,5)}(t)+p_{(1,3,2)}^{(1,10,5)}(t)+p_{(3,3,2)}^{(14,11,5)}(t)+ \\ +p_{(3,1,2)}^{(12,4,5)}(t)+p_{(1,1,1,2)}^{(1,2,3,8)}(t)+p_{(3,1,2)}^{(14,3,8)}(t)+ \\ +p_{(1,3,2)}^{(1,13,8)}(t)+p_{(1,2)}^{(1,6)}(t)+p_{(1,1,1,1,2)}^{(1,2,3,4,5)}(t)+ \\ +p_{(1,3,1,2)}^{(1,13,4,5)}(t)+p_{(3,1,1,2)}^{(14,3,4,5)}(t) \end{array}\right)}{1-\left(\begin{array}{l} p_{(1,4)}^{(1,15)}(t)+p_{(1,1,4)}^{(1,2,16)}(t)+p_{(1,1,1,4)}^{(1,2,3,17)}(t)+ \\ +p_{(1,1,1,1,4)}^{(12,3,4,18)}(t)+p_{(3,4)}^{(14,16)}(t)+p_{(1,3,4)}^{(1,3,17)}(t)+ \\ +p_{(3,1,4)}^{(14,3,17)}(t)+p_{(3,4)}^{(12,17)}(t)+p_{(3,4)}^{(9,18)}(t)+ \\ +p_{(1,3,4)}^{(1,10,18)}(t)+p_{(3,3,4)}^{(14,11,18)}(t)+p_{(3,1,4)}^{(12,4,18)}(t)+ \\ p_{(1,1,3,4)}^{(1,2,11,18)}(t)+p_{(1,3,1,4)}^{(1,13,4,18)}(t)+p_{(3,1,1,4)}^{(14,3,4,18)}(t) \end{array}\right)}. \tag{3.29}$$

На основі отриманих еквівалентних передавальних функцій було побудовано графіки щільності розподілу часу виконання всього процесу обфускації і деобфускації програмного модуля з урахуванням варіацій значень змінних k та θ. Дані графіки, а також відповідні графіки функції розподілу, представлені на рис. 3.9, 3.10.

Таблиця 3.4. Характеристики переходів між станами GERT-мережі процесів обфускації і деобфускації програмних модулів з доданою додаткової оперується функцією обфускації

№ з/п	Гілка	W-функція	Ймовірність переходу	Коефіцієнти щільності ймовірності (3.3)
1.	(1, 2)	W_{12}	P_1	$k=k_1$; $\theta=\theta_1$
2.	(2, 3)	W_{13}	P_2	$k=k_1$; $\theta=\theta_1$
3.	(3, 4)	W_{34}	P_3	$k=k_1$; $\theta=\theta_1$
4.	(4, 5)	W_{45}	P_4	$k=k_1$; $\theta=\theta_1$
5.	(5, 6)	W_{56}	P_5	$k=k_2$; $\theta=\theta_2$
6.	(2, 6)	W_{26}	P_6	$k=k_2$; $\theta=\theta_2$
7.	(3, 6)	W_{36}	P_7	$k=k_2$; $\theta=\theta_2$
8.	(4, 6)	W_{46}	P_8	$k=k_2$; $\theta=\theta_2$
9.	(1, 5)	W_{15}	P_9	$k=k_3$; $\theta=\theta_3$
10.	(2, 5)	W_{25}	P_{10}	$k=k_3$; $\theta=\theta_3$
11.	(3, 5)	W_{35}	P_{11}	$k=k_3$; $\theta=\theta_3$
12.	(1, 4)	W_{14}	P_{12}	$k=k_3$; $\theta=\theta_3$
13.	(2, 4)	W_{24}	P_{13}	$k=k_3$; $\theta=\theta_3$
14.	(1, 3)	W_{13}	$P_{14}=1-P_1-P_{12}-P_9$	$k=k_3$; $\theta=\theta_3$
15.	(2, 1)	W_{21}	$P_{15}=1-P_6-P_2-P_{13}-P_{10}$	$k=k_4$; $\theta=\theta_4$
16.	(3, 1)	W_{31}	$P_{16}=1-P_3-P_7-P_{11}$	$k=k_4$; $\theta=\theta_4$
17.	(4, 1)	W_{41}	$P_{17}=1-P_4-P_8$	$k=k_4$; $\theta=\theta_4$
18.	(5, 1)	W_{51}	$P_{18}=1-P_5$	$k=k_4$; $\theta=\theta_4$

Таблиця 3.5. Характеристики щільності ймовірності – математичне сподівання і дисперсія

№ з/п	Тип	μ	σ^2
1.	Оригінальний θ=[2.7, 1.8, 3.5]; k=[1, 2, 1]	8.8	39.51
2.	Доданий вузол θ=[2.7, 1.8, 1.8, 3.5]; k=[1, 2, 2, 1]	11.11	44.68
3.	Видалений вузол θ=[2.7, 3.5]; k=[1, 1]	8.08	34.88

Результати дослідження показали, що для розробленої математичної моделі при додаванні ще одного процесу обфускації дисперсія збільшується на 12%, а при його видаленні з системи – зменшується до 13%. Математичне сподівання змінюється в геометричній прогресії – так, при видаленні вузла отримуємо зменшення математичного сподівання на 9%, а при збільшенні на 1 вузол – збільшення математичного сподівання на 26%. Це показує незначність змін досліджуваних показників в умовах модифікації моделі і підтверджує гіпотезу про уніфікацію моделі в умовах використання математичного апарату гамма-розподілу як основного. Дані результати дають розробнику можливість спрогнозувати поведінку системи захисту програмних модулів з точки зору часу виконання.

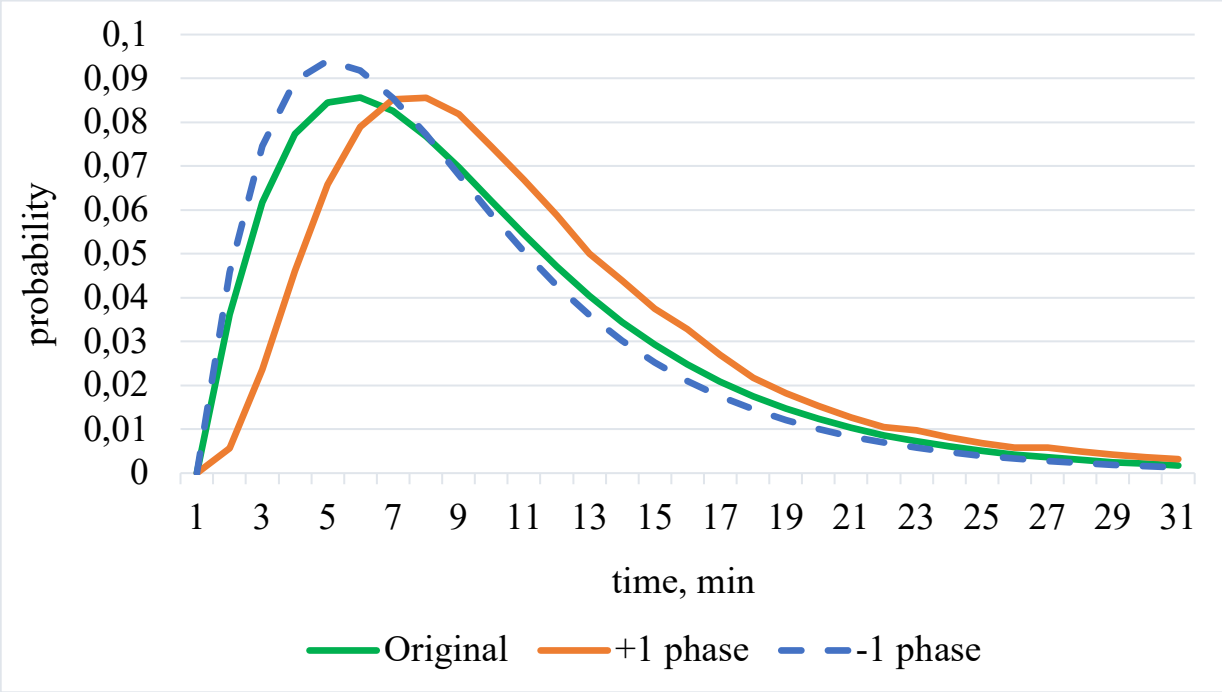


Рис. 3.9. Графіки щільності розподілу часу виконання процесу обфускації і деобфускації програмних модулів при використанні GERT-мережі, гамма-розподілу ймовірностей переходів

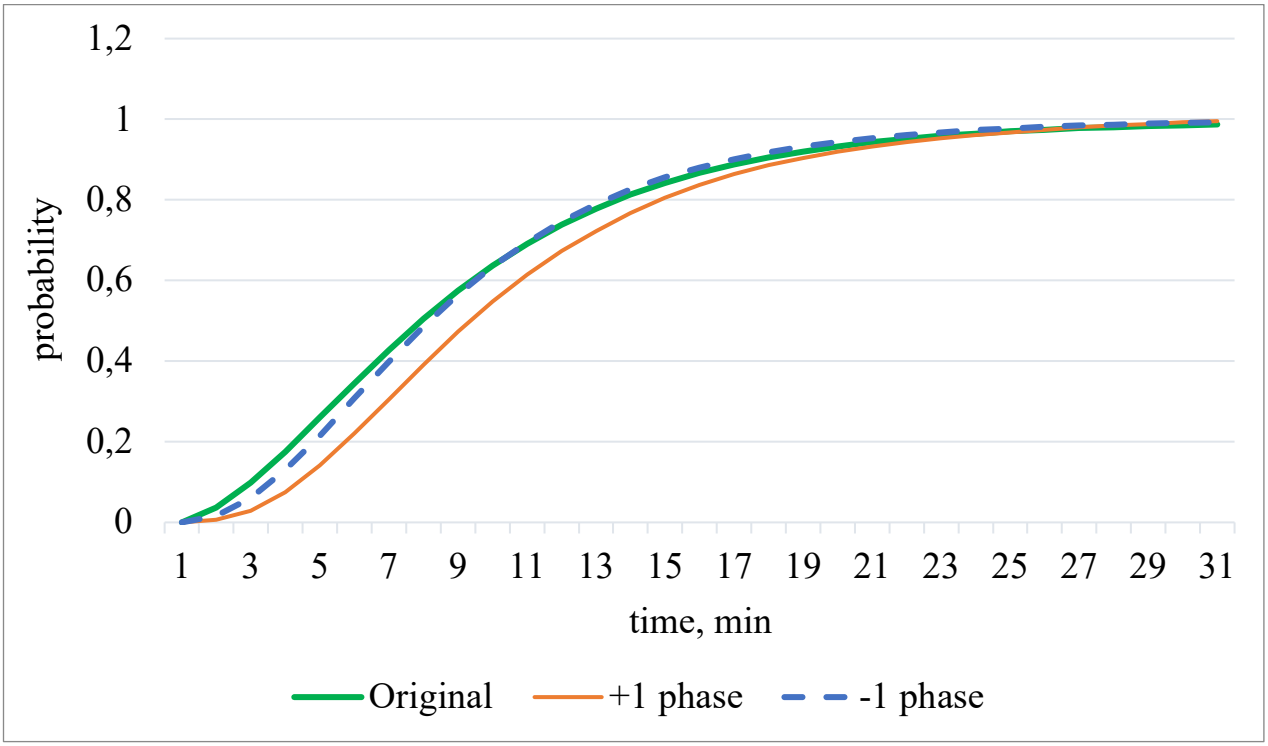


Рис. 3.10. Графіки функцій розподілу процесу обфускації і деобфускації програмних модулів при використанні GERT-мережі, гамма-розподілу ймовірностей переходів

3.5.2 Дослідження результатів дослідження розробленої GERT-моделі процесу обфускації програмних модулів

Синтезовано комплекс алгоритмів обфускації програмних модулів (рис. 3.2), що відрізняється від відомих урахуванням варіативності типів даних. Синтез дозволив представити процес обфускації як єдине ціле (рис. 3.3), а також формалізувати процес обфускації в зручному вигляді для подальшого використання в розроблюваних GERT-моделях (рис. 3.4).

В рамках дослідження була розроблена уніфікована GERT-модель процесу обфускації програмних модулів. Дана модель відрізняється від відомих реалізацією парадигми використання математичного апарату гамма-розподілу (3.4) в якості ключового на всіх етапах моделювання процесу обфускації. Це дозволило досягти уніфікації моделі в умовах модифікації GERT-мережі (рис. 3.5, 3.6). Уніфікація дозволяє адаптувати процес моделювання до можливого ускладнення структури і алгоритмів процесів

обфускації.

Результати дослідження показали, що для розробленої математичної моделі при додаванні ще одного процесу обфускації дисперсія часу виконання збільшується на 12%, а при його видаленні з системи – зменшується до 13% (рис. 3.8, 3.9). Математичне сподівання часу виконання змінюється в геометричній прогресії – так, при видаленні вузла відбувається зменшення математичного сподівання на 9%, а при збільшенні на 1 вузол – збільшення математичного сподівання на 26%. Це показує незначність змін досліджуваних показників в умовах модифікації моделі і підтверджує гіпотезу про уніфікацію моделі в умовах використання математичного апарату гамма-розподілу як основного. Дані результати дають розробнику можливість спрогнозувати поведінку системи захисту програмних модулів з точки зору часу виконання. Це дозволяє зменшити час на прийняття рішення про доцільність використання процесу обфускації в умовах використання гнучких методологій.

Дослідження показали, що розроблену математичну модель доцільно використовувати при математичному моделюванні систем, які формалізуються мінімум чотирма станами. Зменшення кількості станів призводить до лінеаризації процесу, для якого використання стохастичних підходів до математичного моделювання призводить до погіршення точності результатів. Також, додаткове зменшення кількості станів призводить до зменшення безпеки програмних модулів (рис. 3.9, 3.10). Так, на рис. 3.7 представлена модель з чотирма станами, описана шістьма переходами (табл. 3.3). Зменшення кількості станів на 1 призведе до системи, що має 3 переходи.

Рекомендований максимум кількості станів становить 9 вузлів. Подальше збільшення кількості вузлів призводить до надмірного ускладнення математичної моделі, при цьому зберігаються тенденції варіювання математичного сподівання і дисперсії часу виконання процесу.

На даному етапі коефіцієнти k, θ підбиралися емпіричним шляхом за допомогою експертних знань. Так, вхідні дані отримані в результаті експерименту, проведеного групою експертів-розробників безпечного програмного забезпечення фірми NixSolutions. Надалі, це обмеження може бути усунуте методикою обчислення даних коефіцієнтів для конкретних алгоритмів захисту даних. З усуненням цих обмежень пов'язаний напрямок подальших досліджень, який повинен бути орієнтований на розвиток процедури адаптації даних коефіцієнтів в різні бізнес-процеси обфускації програмних модулів.

Розвиток даного дослідження полягає в розробці методики обчислення гамма-розподілу і його адаптації для практичної реалізації алгоритмів захисту даних. Однак можуть виникнути труднощі, пов'язані з існуючими обмеженнями стохастичних підходів до моделювання.

3.6 Оцінка якості обфускації програмних модулів

3.6.1 Синтез апарату оцінки якості обфускації програмних модулів на основі показників якості програмного продукту

На основі аналізу існуючих метрик оцінки якості коду були сформовані показники, що представляють найбільший інтерес для подальшого дослідження [176, 183, 218]:

1) Кількісні метрики:

1.1 Кількість рядків коду. Слід зазначити, що дана характеристика може значно відрізнятися для вихідного і декомпілюваного коду. У зв'язку з тим, що дана метрика не завжди залежить від автора коду (програміста), то ваговий коефіцієнт значущості даної характеристики повинен бути низьким. Дана метрика була спочатку розроблена для оцінки трудовитрат за проєктом. Однак через те, що одна і та ж функціональність може бути розбита на кілька рядків або записана в один рядок, метрика стала практично непридатною з появою мов, в яких в один рядок може бути записано більше однієї команди.

З точки зору оцінки якості обфускації - чим вище (більше) даний показник - тим краще. Незважаючи на свою «непотрібність» в чистому вигляді, дана метрика повинна бути складовою інших метрик, зокрема, для накопичувальних, для усереднення значення метрик великих і дрібних аналізованих проектів.

1.2 Середнє число рядків для функції. Також може застосовуватися для класу, файлу і пакета (package в мові програмування Java) або простору імен (namespace в мові програмування C#). Даний показник є розвитком показника кількості рядків коду, проте, він має додаткову характеристику - зменшення абсолютного значення показника даної метрики показує збільшення декомпозиції і модульності вихідного (декомпілюваного) коду, що призводить до підвищення читаності і зниження часу аналізу коду. У зв'язку з тим, що різні блоки цього блоку мають різні вимоги до забезпечення якості послуг безпеки (наприклад, модуль ліцензування, де вимоги максимальні і модуль відображення звітів, де вимоги мінімальні), дана метрика «витісняє» метрику кількості рядків загального коду.

1.3 Метрика Холстеда – метрика, заснована на показнику унікальності операторів. Детальне дослідження даної метрики показало, що його використання не дає якісних показників, у зв'язку з бізнес-вимогами до процесу розробки програмного продукту. Так, дана метрика не дозволить ідентифікувати добре обфускований код із застосуванням «мертвих» блоків від ідеально написаного модуля управління космічним кораблем, в якому є багато математичних виразів. Також, як і метрика кількість рядків коду, дана метрика не передбачає розвитку в рамках даного дослідження.

1.4 Метрика Джілба. Оцінює складність ПЗ на основі насиченості коду умовними операторами і операторами циклу.

$$CL = (\sum C + \sum L) / T \tag{3.30}$$

З точки зору оцінки якості обфускації, чим нижче дана характеристика - тим вище ступінь обфускації. Дана метрика, не дивлячись на свою

простоту, досить добре відображає складність написання і розуміння програми, а при додаванні такого показника як максимальний рівень вкладеності умовних і циклічних операторів, ефективність даної метрики значно зростає.

1.5 Fitzpatrick метрика. Дана метрика описана формулою:

$$ABC = \sqrt{A^2 + B^2 + C^2} \tag{3.31}$$

де A – кількість присвоювань, B – кількість викликів, C – кількість перевірок.

Так само, як і метрика Джілба, заслуговує уваги для подальшого розвитку, шляхом використання додаткових характеристик. З точки зору оцінки якості обфускації, чим нижче дана характеристика - тим вище ступінь обфускації.

2) Метрики складності потоку керування. Основним представником цієї категорії метрики є цикломатична складність (складність МакКейба), що описана як:

$$V(G) = e - n + 2p \tag{3.32}$$

де e – кількість дуг, n – кількість вершин, p – число компонент зв'язності графа потоку управління (яка в коректно написаних програмах = 1).

Решта метрик цієї категорії є розвитком і вдосконаленням метрики цикломатичної складності. Метрики оцінки складності потоку управління характеризуються складністю обробки, так як для їх реалізації необхідно виконати синтаксичний аналіз і побудувати граф потоку керування. Незважаючи на розвиток метрик даної категорії, вони мають ряд архітектурних недоліків, які призвели до невикористання їх для обчислення ступеню обфускованості коду (однак, це не заважає їх продовжувати використовувати, але при цьому з мінімальним ваговим коефіцієнтом):

- вони не розрізняють циклічні і умовні конструкції;
- вони не розрізняють складність предикатів;
- відсутність обліку складності паралельних взаємодій;
- відсутність визначення атомарності вузла графу;

– високорівневі мови програмування, зокрема, об’єктно-орієнтовані, припускають слабке розгалуження [204], в зв’язку з чим, цикломатична складність буде прагнути до мінімального значення.

3) Метрики складності потоку керування даними. У чистому вигляді ця категорія метрик недостатня для того, щоб охарактеризувати складність аналізу трас програм в бінарних кодах. Застосування метрик складності потоку даних до двійкового коду ускладнюється також тим, що компілятор може використовувати для розміщення змінних як пам’ять, так і регістри процесора, і потрібно додатковий аналіз з метою зведення змінних в єдиний список.

3.1. Метрика Чепіна оцінює інформаційну міцність окремо взятого модуля за характером використання змінних зі списку введення-виведення. Всі множини змінних розбиваються на 4 групи:

i. P (вводяться для розрахунків і для забезпечення виведення);
ii. M (модифікуються, або створюються всередині програми);
iii. C (керуючі);
iv. T (паразитні (що не використовуються)).

Змінні, які виконують кілька функцій, враховуються в кожній функціональній групі. Метрика Чепіна виражається формулою:

$$Q = a_1 \cdot P + a_2 \cdot M + a_3 \cdot C + a_4 \cdot T \qquad (3.33)$$

де a_1, a_2, a_3, a_4 – вагові коефіцієнти. Експериментальним шляхом, коефіцієнти були вирахувані і їм присвоєні значення: 1, 2, 3 и 0.5 відповідно. Паразитні змінні не збільшують складність потоку даних, але ускладнюють розуміння програми.

3.2 Метрика Спена ґрунтується на локалізації звернень до даних усередині кожної програмної одиниці. Спен (span) – це число звернень до змінної між її першою і останньою появою в програмі: змінна, яка зустрілася в блоці коду (наприклад, функції) n раз, має $Спен = n - 1$. Слід зазначити, що використання однієї і тієї ж змінної (ідентифікатора) всередині

коду велику кількість разів ускладнює тестування і відлагодження, і, отже, може характеризувати код або з боку поганої якості, де, найчастіше, буде зустрічатися однакові вирази, або наявністю блоків «мертвого», а також невикористаного і обфускованого коду.

В ході аналізу існуючих метрик якості коду і їх класифікації, запропоновані наступні вдосконалення для впровадження метрик коду:

1) Доповненням метрики середнього значення може бути метрика, що спрямована на визначення рівня вкладеності кожного модуля. У зв'язку з тим, що більшість обфускаторів коду для введення мертвого і невикористаного коду вводять додаткові цикли і умовні оператори, то підвищений рівень вкладеності модуля може свідчити про обфускації коду. Навіть якщо код не обфускований, його якість при великому рівні вкладеності буде падати.

2) Вдосконалення Fitzpatrick метрики може бути виконано з урахуванням розширення опису умовних операторів. З огляду на той факт, що дана метрика включає в себе складову метрики Джімбла, то в рамках розвитку даної метрики пропонується використовувати 4-й параметр – D – кількість викликів операторів циклу. Таким чином, формула буде мати такий вигляд:

$$ABC = \sqrt{A^2 + B^2 + C^2 + D^2} \tag{3.34}$$

До умовних операторів слід відносити такі конструкції:

i. Оператор if – класичний умовний оператор, який використовується в даній метриці;

ii. Тернарний оператор;

iii. Оператори порівняння;

iv. Операції фільтрації в сучасних механізмах обробки даних, такі як 1) filter у Java Stream API [155], 2) Where у C# LINQ [241].

3) Одним з варіантів розвитку метрики цикломатичної складності може бути оцінка ступеня взаємодії між модулями. Дослідження даного

підходу показали, що чим нижче рівень зв’язності – тим вище ступінь обфускації. При збільшенні рівня зв’язності зростає рівень повторного використання коду, а значить, аналіз часто-використовуваних модулів дозволить скоротити час на деобфускацію програмного продукту. Зворотною стороною «медалі» даного варіанту розвитку є факт того, що чим вище зчеплення, тим вище ентропія і тим важче зрозуміти систему і її поведінку. Теоретично, це нововведення може бути покрите метриками Холстера, однак, дана метрика не враховує «мертвий код», який буде враховуватися, на відміну від запропонованого методу.

4) Властивості метрики Спен можна використовувати шляхом використання статистичного аналізу, і середньоквадратичного відхилення зокрема. Дану властивість можна використовувати як індикатор якості оцінки коду - використання однієї і тієї ж змінної (ідентифікатора) всередині коду велику кількість разів ускладнює тестування і відлагодження, і, отже, може характеризувати код або з боку поганої якості, де, найчастіше, будуть зустрічатися однакові вирази, або наявністю блоків «мертвого», а також невикористаного і обфускованого коду.

3.6.2 Розробка алгоритму отримання метрик якості коду програмного продукту

Для аналізу декомпільованих застосунків було розроблено програмну модель, яка для набору проектів робить зведену таблицю з інформацією щодо основних метрик, а також показує інформацію про частоти основних груп операторів. Результуючий файл має розширення .csv, що представляє собою «список, розділений комою», що дозволяє використовувати даний файл для імпорту в якості таблиць Microsoft Excel, при цьому не використовуючи хитромудрі бібліотеки, що представляє функціонал роботи з даним ПЗ.

Так, на першому етапі (рис. 3.11) розроблений програмний продукт отримує на вхід кореневу папку, в якій розташовуються проекти. Передбачається, що проекти будуть розділені на 3 великі категорії:

– оригінальні файли вихідного коду;

– декомпільовані файли скомпільованих оригінальних файлів вихідного коду;

– декомпільовані файли обфускованих файлів.

На другому кроці проходить обхід кожного проекту, при цьому на кроці 3 відбувається обхід дерева каталогів кожного проекту в пошуках файлів з вихідними кодами. Так, при розробці програмного продукту, в проектах можуть брати участь «допоміжні» файли, такі так файли збірки, файли-ресурси (зображення, написи), а також файли, що не припускають обфускації для даної мови програмування. Основним напрямком використання мов, заснованих на проміжному коді, є Enterprise веб-розробка. У зв'язку з цим, з'являється багато коду, який не представляє можливості обфускації (наприклад, веб-сторінки), або не представляє цінності обфускації і / або може бути обфускований іншими типами обфускаторів (наприклад, для скриптових мов, таких як Javascript, який використовується при веб-розробки Enterprise-застосунків). Для мови програмування C#, до файлів, які представляють цінність, відносяться файли з розширенням .cs.

На кроках 4-5 для кожного отриманого файлу обчислюється набір метрик, які акумулюються і для кожного пройденого проекту створюється запис в .csv файлі на кроці 6.

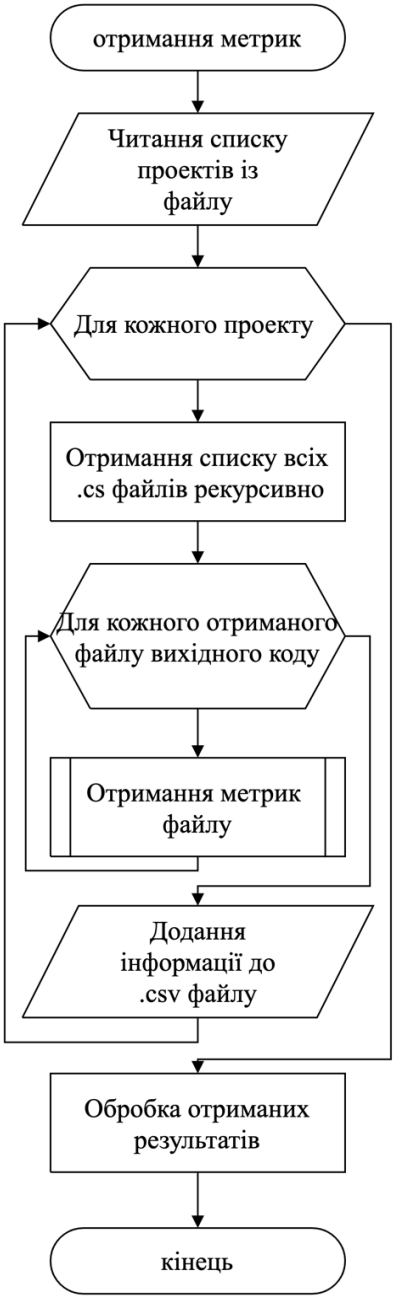


Рис. 3.11. Загальний алгоритм отримання .csv файлу для подальшого дослідження

Після обробки всіх файлів і проектів, результуючий файл готовий до аналізу, зокрема, для отриманих результатів можна побудувати порівняльні графіки частоти народження кожної групи операторів.

Детальне отримання метрик файлу можна описати алгоритмом, зображеним на рис. 3.12.

На першому кроці алгоритму відбувається отримання синтаксичного дерева. У C# для роботи з синтаксичним деревом нещодавно була розроблена бібліотека Roslyn [21, 181]. На жаль, вона на поточний момент підтримує тільки мови сімейства .NET, але при цьому продуктивніша і зручніша у використанні за «кросплатформну» бібліотеку ANTRL [100]. Синтаксичне дерево виходить за допомогою методу класу CSharpSyntaxTree. ParseText() простору імен Microsoft.CodeAnalysis.CSharp, який повертає SyntaxTree, представляє собою кореневий вузол.

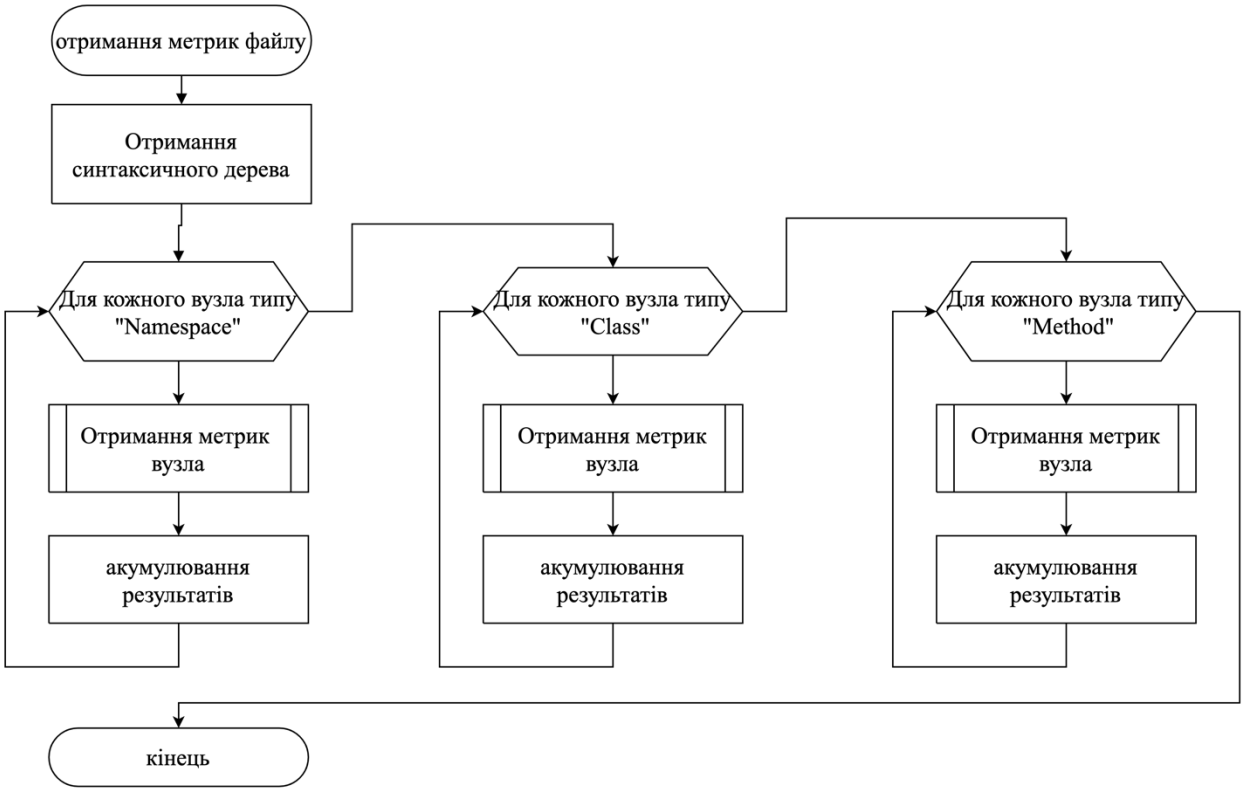


Рис. 3.12. Отримання метрик файлу

Отримання метрик вузла являє собою лінійний алгоритм, основне завдання якого – виклик функцій визначення конкретної метрики. Так, в дослідженні беруть участь метрики Спен, метрика рівня вкладеності, кількісні метрики груп операторів, метрика кількості рядків коду (LoC), FitzPatrik метрика.

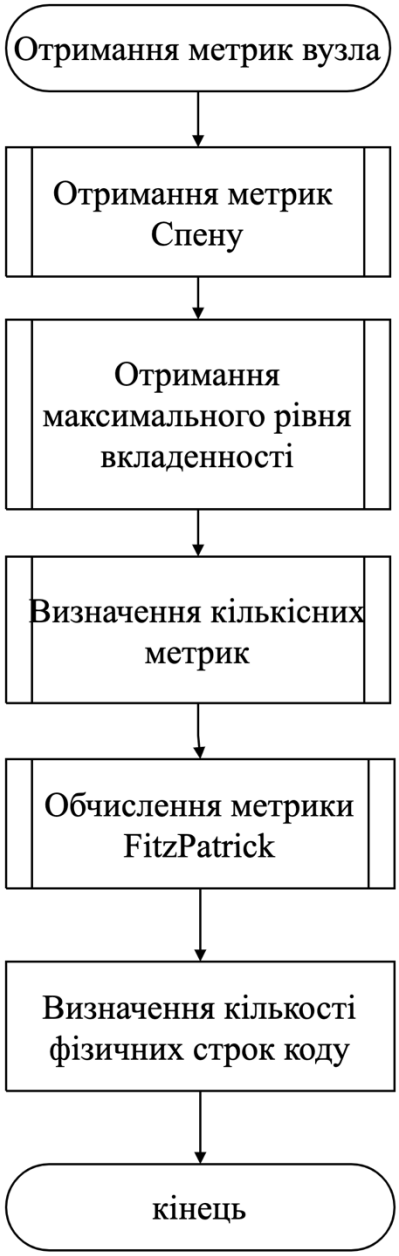


Рис. 3.13. Отримання метрик вузла

На кроках 2, 5, 8 виконується отримання вузлів, що характеризують модульність вузла. В рамках дослідження передбачається використання 3 типів модульності, кожен з яких має відповідний тип вузла синтаксичного дерева бібліотеки Roslyn:

– пространство імен: SyntaxKind.NamespaceDeclaration;
– клас: SyntaxKind.ClassDeclaration;

– метод: SyntaxKind.MethodDeclaration.

На кроках 2, 6, 9 відбувається отримання метрик конкретного вузла, описаного алгоритмом рис. 3.13.

На кроках 3, 7, 10 відбувається накопичення результатів метрик в рамках конкретного типу модульності вузла.

Процес отримання кількісних метрик описаний на рис. 3.14.

Для дослідження було створено 5 груп операторів, які інкрементуються в залежності від типу на кроках 3, 5, 7, 9, 11:

– Оператори циклу. До даної групи операторів відносяться:
  – цикл for (SyntaxKind.ForStatement);
  – цикл while .. do (SyntaxKind.WhileStatement);
  – цикл do .. while (SyntaxKind.DoStatement);
  – цикл foreach (SyntaxKind.ForEachStatement).
– Умовні оператори. До даної групи операторів відносяться:
  – оператор if (SyntaxKind.IfStatement , SyntaxKind.ElseClause);
  – тернарний оператор (SyntaxKind.ConditionalExpression);
  – оператор switch (SyntaxKind.SwitchStatement);
  – оператор фільтрації даних в LINQ C# (SyntaxKind.WhereClause).
– Безумовні переходи (переходи на мітки за допомогою оператора goto: SyntaxKind.GotoStatement.
– Оператори присвоювання, в тому числі суміщені:
  – SyntaxKind.AddAssignmentExpression;
  – SyntaxKind.AndAssignmentExpression;
  – SyntaxKind.CoalesceAssignmentExpression;
  – SyntaxKind.DivideAssignmentExpression;
  – SyntaxKind.ExclusiveOrAssignmentExpression;
  – SyntaxKind.LeftShiftAssignmentExpression;
  – SyntaxKind.ModuloAssignmentExpression;
  – SyntaxKind.MultiplyAssignmentExpression;

– SyntaxKind.OrAssignmentExpression;
– SyntaxKind.RightShiftAssignmentExpression;
– SyntaxKind.SimpleAssignmentExpression;
– SyntaxKind.SubtractAssignmentExpression).

– Виклики функцій (SyntaxKind.InvocationExpression). При цьому слід зазначити, що даний тип вузла включає в себе виклики конструкторів, а також гетери і сетери, які збільшують лічильник даного типу вузла, але не несуть корисної інформації. У зв’язку з цим, при підрахунку кількісних характеристик даного типу вузла, функції-конструктори, а також функції, назва яких починається з «get», «set», "is", "has" повинні бути виключені.

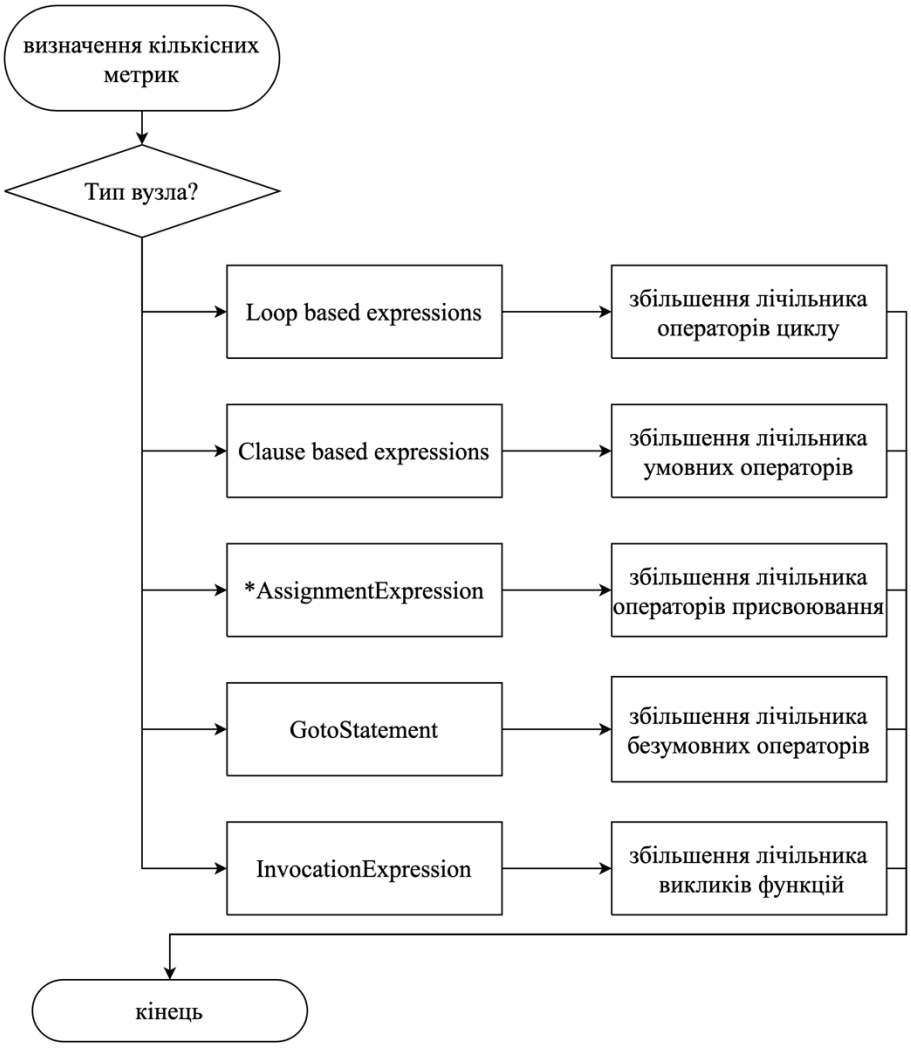


Рис. 3.14. Отримання кількісних метрик

Процес отримання максимального рівня вкладеності вузла описаний на рис. 3.15.

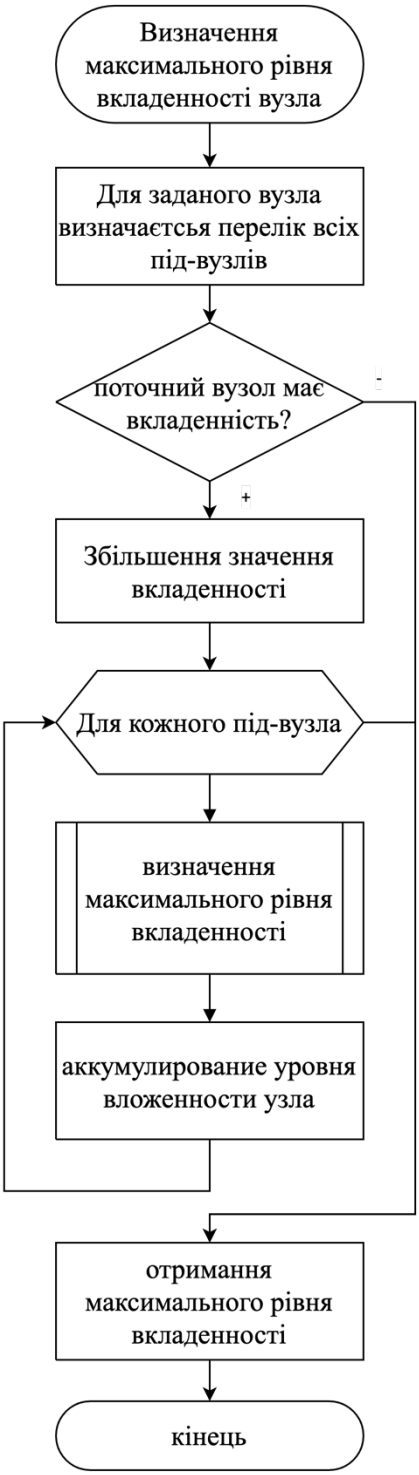


Рис. 3.15. Алгоритм визначення максимального рівня вкладеності

На першому кроці відбувається визначення всіх підсистем (не рекурсивно) для поточного вузла. Якщо поточний вузол має вкладеність, то на кроці 3 збільшуємо лічильник вкладеності і на кроці 5 виконуємо аналогічну процедуру (починаючи з кроку 1) для всіх вкладених вузлів поточного вузла з акумулюванням результату на кроці 6. Після рекурсивного обходу вузлів і визначення рівня вкладеності кожної гілки, визначаємо максимальну з них на кроці 7.

Отримання метрики Спен описано алгоритмом, який наведено на рис. 3.16.

Рис. 3.16. Алгоритм визначення значення Спену для змінних даного вузла

Так, на першому кроці отримуємо перелік оголошень всіх змінних в даному вузлу (при використанні Roslyn, шукаємо всі вузли з типом SyntaxKind.VariableDeclaration). На кроці 2 знаходимо все вузли з типом SyntaxKind.IdentifierName, які будуть відображати використання змінних. Після отримання списку всіх змінних формуємо асоціативний масив (Map / Dictionary), в який на кроці 3-5 накопичуємо абсолютне значення використання в даному вузлу.

Кількісні метрики, отримані в результаті виконання алгоритму, описаного на рис. 3.14, необхідні для отримання значення FitzPatrik вузла, алгоритм якого описаний на рис. 3.17. Даний алгоритм описаний виразом (3.30).

На кроці 1-3 відбувається підготовка даних – отримання необхідних кількісних метрик.

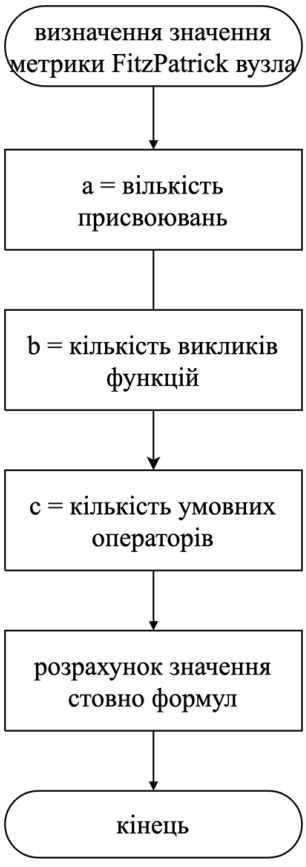


Рис. 3.17. Алгоритм визначення метрики FitzPatrik даного вузла

У рамках дослідження було сформовано інформацію про показники метрик з декомпільованим кодом та обфускованим декомпільованим у порівнянні з вихідним кодом. Результати наведені у табл. 3.6.

Таблиця 3.6. Показники метрик з декомпільованим кодом та обфускованим декомпільованим у порівнянні з вихідним кодом

Метрика	**Декомпільований код**	**Обфускований (декомпільований) код**
Фізична кількість строк коду	На 20% менш, ніж у вихідному коді	У 4 рази більш, ніж у вихідному коді
Максимальна вкладеність	(як у вихідному)	На 20% більш, ніж у вихідному коді
Значення Спену	На 10% менш, ніж у вихідному коді	У 2 рази менш, ніж у вихідному коді
Кількість `IF` блоків	(як у вихідному)	У 8 разів більш, ніж у вихідному коді
Кількість `ELSE` блоків та тернарних операторів	На 20% менш, ніж у вихідному коді	У 7 разів більш, ніж у вихідному коді
Кількість `SWITCH` блоків	(як у вихідному)	У 5 разів більш, ніж у вихідному коді
Кількість `GOTO` переходів та міток	0 (як у вихідному)	Залежить від фізичної кількості строк коду (PLoC * 0.11)
Кількість `WHILE` операторів	У 2 рази менш, ніж у вихідному коді	У 8 разів більш, ніж у вихідному коді
Кількість `Do While` операторів	(як у вихідному)	Залежить від фізичної кількості строк коду (PLoC * 0.01)
Значення метрики FitzPatrik	На 10% менш, ніж у вихідному	У 2 рази більш, ніж у вихідному коді

В рамках дослідження було проведено експеримент. Був розроблений модуль для створення та верифікації ліцензійного ключа на основі характеристик системи, що був обфускований розробниками. Мета експерименту – визначити, скільки часу необхідно IT-фахівцям для того, щоб розплутати алгоритм і автоматизувати процес отримання необхідних даних. Була зроблена вибірка зі 100 осіб.

Результат показав, що для розплутування алгоритму в середньому витрачається 5.3 годин зі значенням середньоквадратичного відхилення

рівного 1.1 години. На рис. 3.18 наведено порівняльну діаграму часу (вісь OY) аналізу обфускованого та необфускованого коду для кожної особи (вісь OX).

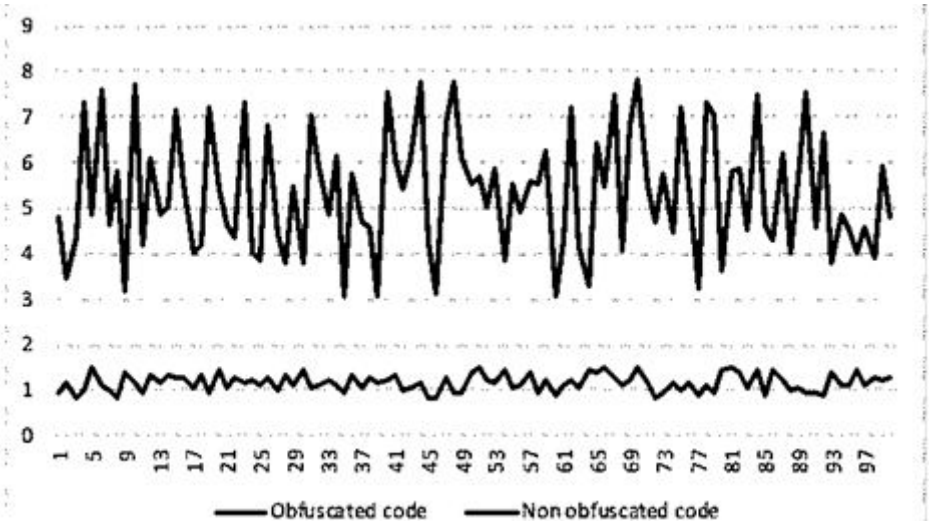


Рис. 3.18. Порівняльна діаграма часу аналізу обфускованого (зверху) та необфускованого (знизу) коду розробленого модуля

Висновки за розділом 3

1. Синтезовано комплекс алгоритмів обфускації і деобфускації програмних модулів, який відрізняється від відомих урахуванням варіативності типів даних. Це дозволило описати дані процеси на верхньому стратегічному рівні формалізації.

2. В рамках дослідження була розроблена уніфікована GERT-модель процесу обфускації програмних модулів. Дана модель відрізняється від відомих реалізацією парадигми використання математичного апарату гамма-розподілу в якості ключового на всіх етапах моделювання процесу обфускації. Це дозволило досягти уніфікації моделі в умовах модифікації GERT-мережі.

3. Результати дослідження показали, що для розробленої математичної моделі при додаванні ще одного процесу обфускації дисперсія часу виконання збільшується на 12%, а при його видаленні з системи – зменшується до 13%. Математичне сподівання часу виконання змінюється в

геометричній прогресії - так, при видаленні вузла відбувається зменшення математичного сподівання на 9%, а при збільшенні на 1 вузол - збільшення математичного сподівання на 26%. Це показує незначність змін досліджуваних показників в умовах модифікації моделі і підтверджує гіпотезу про уніфікацію моделі в умовах використання математичного апарату гамма-розподілу як основного. Дані результати дають розробнику можливість спрогнозувати поведінку системи захисту програмних модулів з точки зору часу виконання. Це дозволяє зменшити час на прийняття рішення про доцільність використання процесу обфускації в умовах використання гнучких методологій.

4. Розроблено методи обфускації строкових літералів та імен ідентифікаторів, доцільність використання яких підтверджено експериментом, в якому наведено, що час, який витрачається на деобфускацію з використанням розроблених засобів обфускації, до 5 разів більше.

5. Проаналізовано основні метрики оцінки якості коду та запропоновано теоретичні вдосконалення на основі практичного досвіду аналізу необфускованого коду. Запропоновані вдосконалення засновані на наступних показниках, які будуть відрізнятися в декомпільованому обфускованому і необфускованому вихідних кодах: 1) кількісні показники частоти виникнення операторів мови, такі як оператори циклу, безумовні переходи і мітки; 2) частоті використання ідентифікаторів, таких як змінні, константи, функції.

6. Розроблено спосіб отримання метрик для багатопроєктного рішення, що дозволяє отримати зведений файл для можливості подальшої обробки. Подальші дослідження припускають: 1) реалізацію математичних моделей вдосконалених метрик; 2) реалізацію порівняльної характеристики отриманих результатів; 3) виявлення характеристик коду, що впливають на його якість і, зокрема, на предмет його обфускації.

РОЗДІЛ 4. РОЗРОБКА МОДЕЛІ ЗАХИСТУ ПРОГРАМНОГО ПРОДУКТУ ВІД НЕЛІЦЕНЗІЙНОГО КОПІЮВАННЯ НА ОСНОВІ GERT-МЕТОДУ

Система безпеки програмного забезпечення на основі ліцензійних ідентифікаторів – це спеціальний компонент програмного або програмно-апаратного комплексу, за допомогою якого здійснюється захист авторських прав на ПЗ як об'єкт інтелектуальної власності. Тобто, запобігання нелегальному розповсюдженню програмного продукту, а також, в разі необхідності, доказ права власності учасника на даний інтелектуальний об'єкт.

Один з перспективних засобів захисту такого роду програмного забезпечення – захист з використанням цифрових водяних знаків [142, 162, 164, 165, 166, 167, 179]. Це дозволяє попередити крадіжку інтелектуальної власності, а, якщо це станеться, довести свої права на її володіння. На поточний момент існує ряд підходів, що дозволяють на практиці вбудовувати цифрові водяні знаки в ПЗ [107, 164, 165, 180]. Великі IT-компанії, такі як StarForce Copy Protection Systems, використовують цифрові водяні знаки як основу свого комерційного програмного продукту по захисту від неліцензійного копіювання.

4.1 Вимоги до розроблюваної моделі безпеки програмного забезпечення на основі ліцензійних ідентифікаторів

4.1.1 Дослідження систем цифрових водяних знаків

Завдання впровадження цифрових водяних знаків в ПЗ можна сформулювати наступним чином. Необхідно вбудувати структуру даних S програми P так, щоб S можна було виявити в P і витягти з неї, навіть якщо P

піддалася деякій модифікації (трансформації, оптимізації, упаковці і т.д.). При цьому:

– структура *S* може мати великий розмір і повинна бути розміщена в *P* приховано;

– програма *P* з уже вбудованою структурою *S* не повинна втрачати в продуктивності;

– структура *S* повинна володіти математичною властивістю, що дозволяє стверджувати, що її наявність в *P* є результатом навмисних дій.

Існує цілий ряд перетворень, що дозволяють збільшити стійкість системи цифрових водяних знаків до певного типу атак (tamper-proofing transformations). У деяких випадках в якості таких перетворень виступають приховуючі перетворення (obfuscating transformations). Вони дозволяють ускладнити аналіз програми (динамічний – відлагодження, статичний – дизасемблювання). Гостра потреба у створенні приховуючих програм-перетворювачів (obfuscator) виникла з появою мов, компільованих в проміжний код для віртуальної машини. Приховуючий перетворювач можна розглядати як компілятор, який на вході отримує якусь програму *P*, а на виході видає нову програму *P*', яка зберігає функціональність *P*, але з деякої точки зору складніше для дослідження (атаки). В роботі [107, 127] доведено, що навіть з найменшим ступенем формалізації неможливо теоретично побудувати приховуюче перетворення, яке б перетворювало вихідну програму *P* в «чорний ящик» (включаючи приховуючі перетворення, які обчислюються не обов'язково за поліноміальний час, в повному обсязі зберігають функціональність вихідної програми *P* і підходять тільки для певних моделей обчислень). Проте, на практиці приховуючі перетворення дозволяють трохи підвищити стійкість систем цифрових водяних знаків до певних видів атак. Формальне визначення приховуючих перетворень і класифікація таких перетворень дані в роботі [117, 242].

Статичні цифрові водяні знаки можуть також бути вбудовані в порядок перерахування виразів в конструкції case, в порядку не залежних один від одного виразів, в порядок інструкцій для роботи зі стеком (push і pop), а також в граф керуючої логіки програми.

Слід зазначити, що статичні цифрових водяних знаків дуже складно зробити більш стійкими за допомогою будь-яких перетворень, що підвищують стійкість або приховують. Так, наприклад, патент [117] описує метод, згідно до якого в зображення (яке розміщується в секції статичних даних програми) впроваджуються не просто цифрові водяні знаки, а фрагмент виконуваного коду програми. Під час виконання програми цей фрагмент витягується і виконується. Однак виконання коду «на льоту» демаскує цифрові водяні знаки і дозволяє противнику локалізувати його.

В роботі [141] була сформульована концепція використання цифрових водяних знаків для захисту ПЗ від нелегального використання та запропоновано алгоритми, що використовують в якості стеганографічного контейнера проміжний байт-код, створюваний компілятором з мови Java для віртуальної машини JVM.

Основи систем цифрових водяних знаків на динамічних графах були закладені в роботах [116, 130, 234]. Основна ідея полягає в тому, щоб вбудувати цифрові водяні знаки в топологію графа. Так, вершини графа пов’язані між собою покажчиками, що дещо ускладнює аналіз цієї структури даних.

Основною науковою задачею є: розробка алгоритмів безпечного переходу і кодування цифрових водяних знаків в GERT-мережах, стійких до існуючих загроз ліцензійної безпеки програмних продуктів.

Основне завдання включає розробку моделі системи безпеки програмного забезпечення на основі ліцензійних ідентифікаторів на основі побудованих алгоритмів, що використовують GERT-мережі.

4.1.2 Вимоги моделі до синтезуючих алгоритмів

Запропонована модель безпеки програмного забезпечення на основі ліцензійних ідентифікаторів заснована на теорії складності. Традиційно, задачі теорії складності використовуються для синтезу криптографічних алгоритмів [142], однак особливість пропонованої моделі полягає в тому, щоб використовувати основні проблеми теорії складності для захисту ПЗ від нелегального використання. При цьому спроектована в рамках моделі система ліцензійної безпеки буде задовольняти вимогам поставленого завдання з побудови.

Розглянемо основні вимоги моделі до синтезованих в її рамках алгоритмів:

1. Наявність механізмів кодування даних за допомогою топології графа керуючої логіки алгоритму (дозволяє вбудовувати в граф цифрові водяні знаки);

2. Наявність механізмів нарощування складності графа керуючої логіки (дозволяє проектувати різні алгоритми безпеки ПЗ);

3. Теоретично обґрунтована нерозв'язність конкретних математичних задач на графі і наявність у графа властивостей цифрового водяного знаку (дозволяє говорити про елементи теоретичної стійкості створеної системи).

В рамках даної моделі пред'являються вимоги і до машинної реалізації алгоритму, яка повинна:

1. Базуватися на динамічних структурах даних (елементи емпіричної стійкості);

2. Використовувати багатопоточну реалізацію графа керуючої логіки (додаткові елементи емпіричної стійкості).

Висування даних вимог до програмної реалізації алгоритму обумовлено емпіричною стійкістю засобів захисту програмних систем, які використовують саме динамічні структури даних і відразу кілька потоків.

Емпірична стійкість таких систем базується на тому, що досліднику набагато складніше аналізувати код програми, що розміщує свої дані в динамічній області пам’яті, а також має декілька потоків, які працюють з цими даними. Це пов’язано, по-перше, з «недовговічністю» динамічних даних. Тобто якщо людина, що аналізує програму, не встигне відстежити операції з даними в «купі» (області динамічної пам’яті), то в міру виконання програми ці дані будуть знищені (на їх місце будуть записані інші дані). По-друге, аналіз коду програми, що використовує кілька потоків, кожен з яких працює з динамічними даними, вимагає постійного перемикання контексту виконання процесу (потоку) і аналізу як виконуваного коду, так і області даних. З емпіричної точки зору значно простіше аналізувати програму, що має один (основний) потік і використовує тільки статичні типи даних.

Виконання зазначених вимог дозволяє гарантувати, що побудовані в рамках моделі алгоритми будуть в змозі не лише захищати ПЗ за допомогою активних методів (безпечної перевірки серійних номерів), а й доводити права на володіння системою як об’єктом інтелектуальної власності:

– синтезовані за допомогою моделі алгоритми повинні мати в наявності механізми кодування даних за допомогою топології графа керуючої логіки алгоритму. Це дозволяє вбудовувати довгі цілі числа в структуру (топологію) графа;

– відповідно до наведеної вище моделі, у проектованої системи ліцензійної безпеки ПЗ повинні бути в наявності механізми нарощування складності графа керуючої логіки. Це, перш за все, створює емпіричні обґрунтування стійкості, так як розгалужений і довгий граф керуючої логіки значно складніше аналізувати на практиці. Відзначимо, що обчислювальні ресурси, доступні сьогодні на стандартних персональних комп’ютерах, дозволяють миритися зі спеціально завищеною складністю графа керуючої логіки;

– модель вимагає, щоб система ліцензійної безпеки використовувала теоретично обґрунтовану нерозв'язність (за прийнятний час) конкретних математичних задач на графі і наявність у графа властивостей цифрового водяного знаку. Дана вимога є ключовою в рамках представленої моделі, так як дозволяє говорити про теоретичну стійкість системи, що розробляється для ліцензійної безпеки, в той час як інші вимоги ставляться лише до емпіричної стійкості і пасивного захисту авторського права (за допомогою цифрових водяних знаків).

Важливо зауважити, що проектована в рамках моделі система ліцензійної безпеки не включає в себе багато інфраструктурних елементів. Тобто не відповідає на такі питання, як отримання серійних номерів, автоматизація засобів кодування унікальних ідентифікаторів в граф керуючої логіки, автоматичне нарощування складності цього графа і т.д.

4.2 Розробка моделі безпеки програмного забезпечення на основі ліцензійних ідентифікаторів

На основі досліджених систем водяних знаків, методів атаки на них, а також висунутих вимог до розроблюваної моделі, було сформовано алгоритм ліцензійної безпеки, заснований на системі водяних знаків. Даний алгоритм наведено на рис. 4.1.

Функціональна схема відповідного алгоритму приведена на рис. 4.2. На функціональній схемі відображено наступні компоненти:

– Accumulated Transition: накопичуваний стан. Являє собою, наприклад, мютекс, який чекає, поки до нього не звернуться N раз з різних джерел з позитивним станом. На рис. 4.2 N задається в блоці в круглих дужках. Наприклад, в зазначеному прикладі, Accumulated Transition 1 чекає, поки з секцій 0 і 1 надійде позитивна інформація. У разі, якщо ліцензійний ключ, і, зокрема, біти 0 і 1, невірний, то даний мютекс «увійде в нескінченний цикл»;

– Action: дія. Одна або декілька проміжних дій. Може бути представлено у вигляді виконання проміжного підготовлюваного коду перед початком роботи основної частини;

– Priority Observer. Пріоритезована дія, спрямована на видачу «позитивного» результату тільки однієї «Accumulated Transition». При цьому, якщо є можливість на поточний момент передати позитивний результат кількох «Accumulated Transition», то перевага віддається зазначеній в дужках.

Відповідно до представленого алгоритму, процес ліцензійної безпеки можна описати наступним чином.

Крок 1. На вхід системи подається вхідна послідовність (біти) ліцензійного ключа.

Кроки 2, 4. Вибирається частина бітів, виставлених в «1» і для них виконується дія, необхідна для продовження роботи системи. У найпростішому випадку – формування деякого мютекса, в який записується інформація про те, що якийсь блок бітів успішний. У просунутому випадку – виконання певного коду, без якого подальша робота системи неможлива. Відмінність Кроку 2 від Кроку 4 – відповідність встановленим вимогам бітів і кількості викликаних блоків коду. Так, можливо декілька варіантів:

– дія виконується лише в тому випадку, якщо всі біти групи правильні;

– дія виконується для кожного правильного біта групи;

– кілька дій виконуються підряд або на вибір (випадковий) для кожного правильного біта групи.

Крок 6. Вибирається частина бітів, виставлених в «0» і для них виконується дія, необхідна для продовження роботи системи.

Кроки 3, 5, 7 – перевірка коректності виконаних умов розроблених процедур.

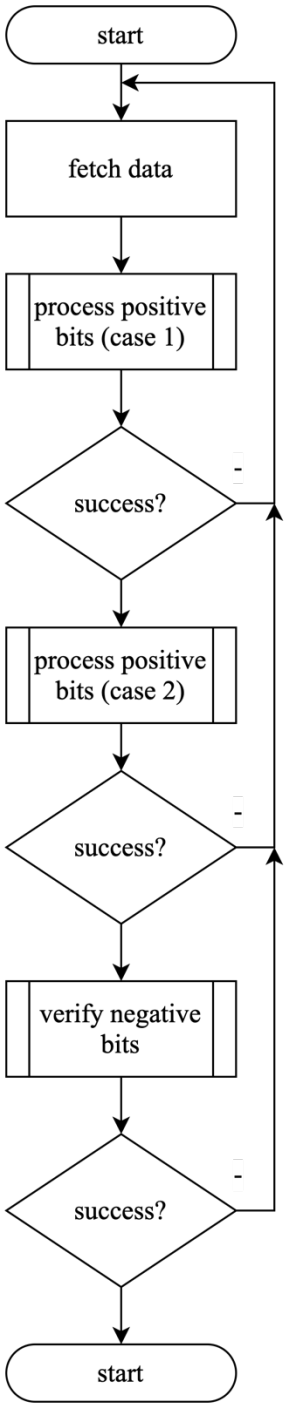


Рис. 4.1. Розроблений алгоритм ліцензійної безпеки на основі водяних знаків

Згідно до поданих матеріалів, розроблена GERT-мережа процесу ліцензійної безпеки програмного забезпечення, відображена на рис. 4.3. Відмінною особливістю даної моделі є наявність станів 5, 8 і 9. Перехід в ці стани відбувається лише в разі, коли система була у всіх попередніх станах попереднього кроку.

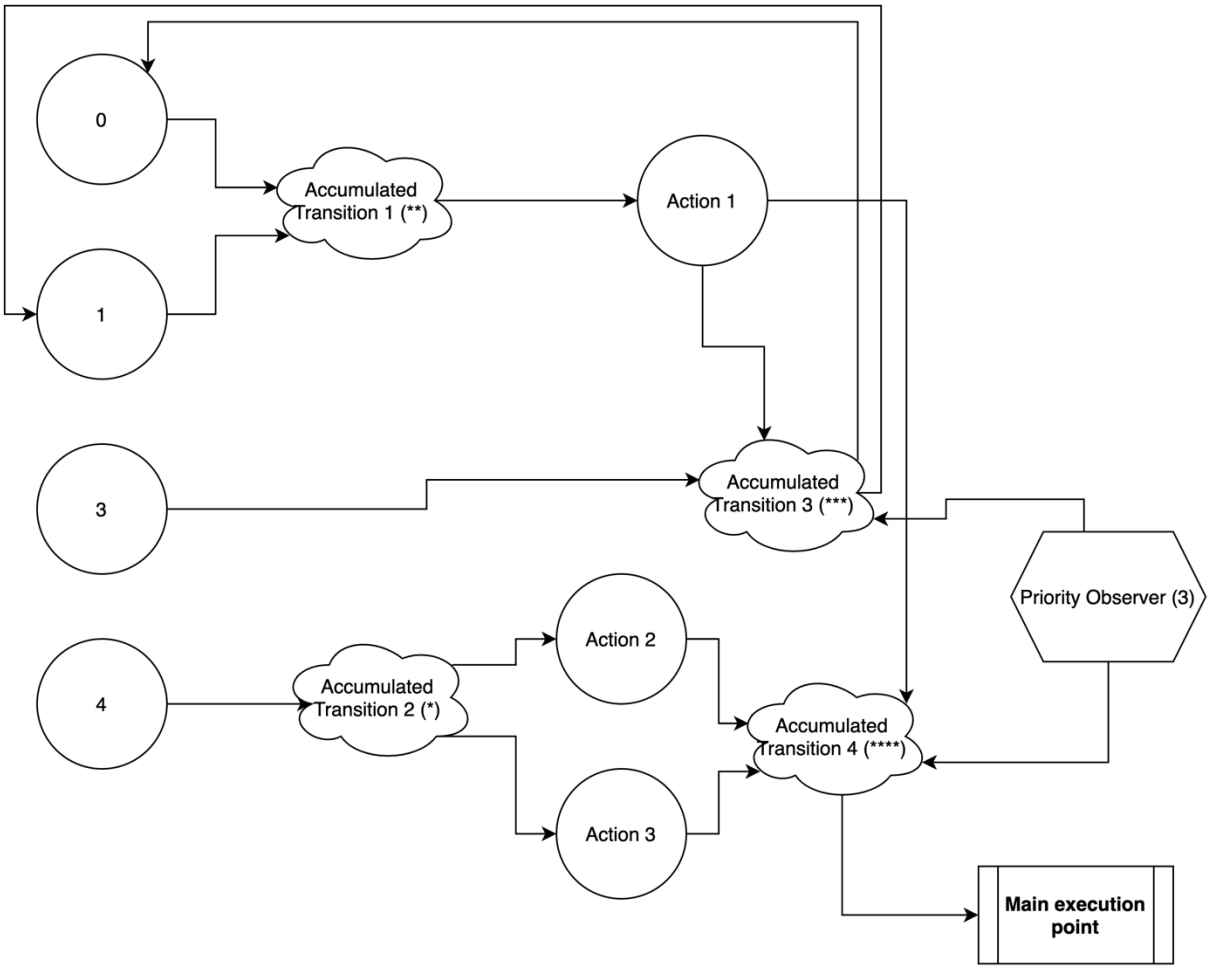


Рис. 4.2. Функціональна схема алгоритму ліцензійної безпеки

Слід зазначити, що розглянута модель рис. 4.3 являє собою процес ліцензійної безпеки програмного забезпечення з використанням 4-х бітного ліцензійного ключа.

На рис. 4.3 і відповідній табл. 4.1, на основі розроблених алгоритмів рис. 4.1, сформульовані переходи між станами, які характеризують:

– (0, 1): взяття 0-го біта ліцензійного ключа;

– (0, 2): взяття 1-го біта ліцензійного ключа;

– (0, 3): взяття 2-го біта ліцензійного ключа;

– (0, 4): взяття 3-го біта ліцензійного ключа;

– (1, 5), (2, 5): обробка двох бітів ліцензійного ключа;

– (5, 0): повернення на початковий стан. Це може статися з двох причин: у разі обробки бітів пройшли або процес обробки бітів ще не повністю

закінчився (наприклад, було оброблено лише 0-й біт і необхідно чекати на обробку 1-го біта);

– (5, 9): інформування системи про те, що група бітів успішно пройшла валідацію;

– (5, 0): повернення на початковий стан. Це може статися у зв’язку з невалідністю вхідних даних;

– (3, 9): інформування системи про те, що група бітів успішно пройшла валідацію;

– (4, 6), (4, 7), (6, 8), (7, 8): обробка старшого біта ліцензійного ключа, в ході якого відбувається виконання корисного коду і переходи на відповідні стани;

– (8, 4): повернення на початковий стан. Це може статися з двох причин: у разі обробки бітів пройшли або процес обробки бітів ще не повністю закінчився (наприклад, був оброблений тільки 0-й біт і необхідно чекати обробку 1-го біта);

– (8, 9): інформування системи про те, що група бітів успішно пройшла валідацію;

– (9, 0): повернення на початковий стан. Це може статися з двох причин: у разі обробки бітів пройшли або процес обробки бітів ще не повністю закінчився;

– (9, 9’): Всі біти успішно пройшли перевірку. Ліцензійний ключ вірний. Перехід на стан штатної роботи програмного продукту;

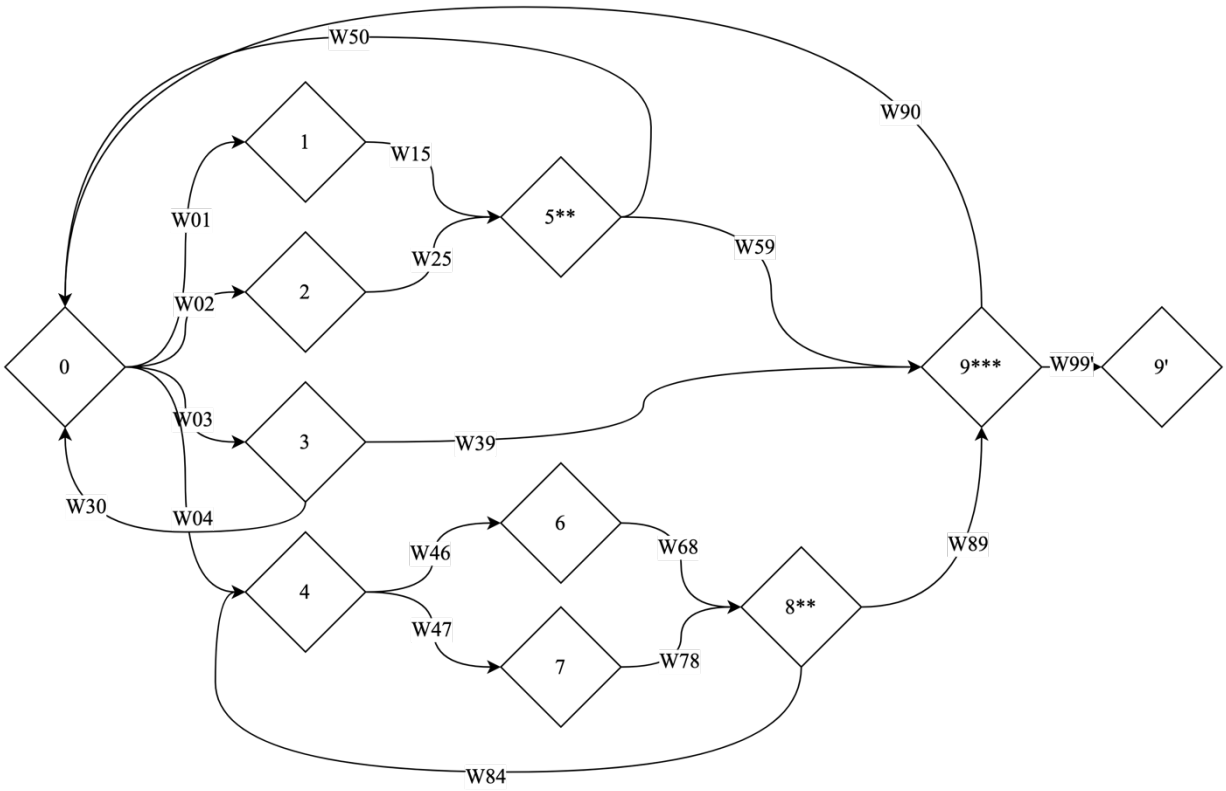


Рис. 4.3. Розроблена GERT-мережа процесу ліцензійної безпеки програмного забезпечення

Дослідження показано, що *W*-функція переходу між станами *i*, *j* визначається формулою:

$$W_{ij}(x) = P_{ij} \int_{-\infty}^{\infty} e^{isx} \zeta_{ij}(x) \mathrm{d}x, \tag{4.1}$$

де $\zeta_{ij}(x)$ – щільність ймовірності переходу між станами *i*, *j*;

P_{ij} – ймовірність переходу зі стану *i* до стану *j*.

В рамках дослідження приймемо гіпотезу – використання гамма-розподілу при моделюванні в якості ключового при описі ймовірнісних переходів зі стану в стан, дозволить досягти уніфікації моделі процесу ліцензійної безпеки програмного забезпечення. Уніфікація полягає в тому, що зменшення або збільшення кількості операцій обфускації незначно змінить результати моделювання при незмінності структурної архітектури моделі.

Таким чином, в даній GERT-мережі функції щільності ймовірності

переходів визначимо гамма-розподілом зі змінними коефіцієнтами k та θ:

$$\zeta(x) = \frac{x^{k-1} \cdot e^{\frac{-x}{\theta}}}{\theta^k \cdot \Gamma(k)}. \tag{4.2}$$

Результуюча W-функція має наступний вигляд:

$$W_E(s) = \frac{W'_E(s)}{1 - W''_E(s)}$$

$$W'_E(s) = W_{01}W_{15}W_{59}W_{99`} + W_{02}W_{25}W_{59}W_{99`} +$$
$$+W_{03}W_{39}W_{99`} + W_{04}W_{46}W_{68}W_{89}W_{99`} + W_{04}W_{47}W_{78}W_{89}W_{99`}$$

$$W''_E(s) = W_{01}W_{15}W_{50} + W_{02}W_{25}W_{50} + W_{03}W_{30} +$$
$$+W_{04}W_{46}W_{68}W_{84} + W_{04}W_{47}W_{78}W_{84} + W_{01}W_{15}W_{59}W_{90} + W_{02}W_{25}W_{59}W_{90} +$$
$$+W_{03}W_{39}W_{90} + W_{04}W_{46}W_{68}W_{89}W_{90} + W_{04}W_{47}W_{78}W_{89}W_{90}$$

Сформована таблиця характеристик розглянутих в GERT-моделі гілок і параметрів розподілу представлена в табл. 4.1.

Використовуючи описаний в розділі 3 вираз, який уособлює добуток W-функцій, що описують успішне і неуспішне виконання алгоритмів, отримаємо результуючий вираз розрахунку еквівалентних передаточних функцій:

$$W_E(t) = \frac{p_{1,2,3,3}^{1,5,7,17} + p_{1,2,3,3}^{2,6,7,17} + p_{1,3,3}^{3,9,17} + p_{1,1,2,3,3}^{4,11,13,15,17} + p_{1,1,2,3,3}^{4,12,14,15,17}}{1 - \left(p_{1,2,4}^{1,5,8} + p_{1,2,4}^{2,6,8} + p_{1,4}^{3,10} + p_{1,1,2,4}^{4,11,13,16} + p_{1,1,2,4}^{4,12,14,16} + p_{1,2,3,4}^{1,5,7,18} + p_{1,2,3,4}^{2,6,7,18} + p_{1,3,4}^{3,9,18} + p_{1,1,2,3,4}^{4,11,13,15,18} + p_{1,1,2,3,4}^{4,12,14,15,18} \right)}$$

Використовуючи щільність розподілу ймовірностей (2), отримуємо графік розподілу щільності ймовірності, відображений на рис. 4.5. При цьому, ймовірно вибрали такий спосіб:

$$P_1 = P_2 = P_3 = P_4 = 0.25; P_5 = P_6 = 1.0; P_7 = 0.3; P_9 = 0.5; P_{11} = 0.5; P_{13} = P_{14} = 1.0; P_{15} = 0.3; P_{17} = 0.2;$$

Мережа може бути побудована так, що якщо в деякому стані і можливий початок однієї з кількох наступних операцій, то ймовірності запуску p_{ij} будь-який з цих операцій утворюють повну групу несумісних подій:

$$\sum_j p_{ij} = 1, \forall i. \tag{4.3}$$

У такому випадку ймовірність виконання всієї мережі від витоку до

стоку дорівнює 1.

Таблиця 4.1. Характеристики переходів між станами GERT-мережі процесу ліцензійної безпеки програмного забезпечення

№ з/п	Гілка	W-функція	Ймовірність переходу	Коефіцієнти щільності ймовірності (4.1)
1.	(0, 1)	W_{01}	P_1	$k=k_1$; $\theta=\theta_1$
2.	(0, 2)	W_{02}	P_2	$k=k_1$; $\theta=\theta_1$
3.	(0, 3)	W_{03}	P_3	$k=k_1$; $\theta=\theta_1$
4.	(0, 4)	W_{04}	$P_4=1-P_1-P_2-P_3$	$k=k_1$; $\theta=\theta_1$
5.	(1, 5)	W_{15}	P_5	$k=k_2$; $\theta=\theta_2$
6.	(2, 5)	W_{25}	P_6	$k=k_2$; $\theta=\theta_2$
7.	(5, 9)	W_{59}	P_7	$k=k_3$; $\theta=\theta_3$
8.	(5, 0)	W_{50}	$P_8=1-P_7$	$k=k_4$; $\theta=\theta_4$
9.	(3, 9)	W_{39}	P_9	$k=k_3$; $\theta=\theta_3$
10.	(3, 0)	W_{30}	$P_{10}=1-P_9$	$k=k_4$; $\theta=\theta_4$
11.	(4, 6)	W_{46}	P_{11}	$k=k_1$; $\theta=\theta_1$
12.	(4, 7)	W_{47}	$P_{12}=1-P_{11}$	$k=k_1$; $\theta=\theta_1$
13.	(6, 8)	W_{68}	P_{13}	$k=k_2$; $\theta=\theta_2$
14.	(7, 8)	W_{78}	P_{14}	$k=k_2$; $\theta=\theta_2$
15.	(8, 9)	W_{89}	P_{15}	$k=k_3$; $\theta=\theta_3$
16.	(8, 4)	W_{84}	$P_{16}=1-P_{15}$	$k=k_4$; $\theta=\theta_4$
17.	(9, 9`)	$W_{99`}$	P_{17}	$k=k_3$; $\theta=\theta_3$
18.	(9, 0)	W_{90}	$P_{18}=1-P_{17}$	$k=k_4$; $\theta=\theta_4$

Щоб показати, що всі вузли задовольняють умові (4.3), розрахуємо ймовірність виконання всього процесу, яка розраховується за формулою:

$$p_E = \left.\frac{W_E(s)}{M_E(s)}\right|_{s=0} = W_E(0) = 1, \tag{4.4}$$

де $M_E(s)$ – твірна функція, при цьому:

$$M_E(s)\big|_{s=0} = \left.\int_{-\infty}^{\infty} e^{sx} f_E(x)\,\mathrm{d}x\right|_{s=0} = \int_{-\infty}^{\infty} f_E(x)\,\mathrm{d}x = 1, \tag{4.5}$$

де $f_E(x)$ – щільності розподілу.

На рис. 4.4 показано графік щільності розподілу часу виконання всього процесу забезпечення ліцензійного захисту програмного забезпечення при k = [2, 4, 4, 3] и θ = [2.7, 1.8, 1.8, 3.5]. Інтегрувавши щільність розподілу

ймовірностей, одержимо функцію розподілу, графік якої відображений на рис. 4.5.

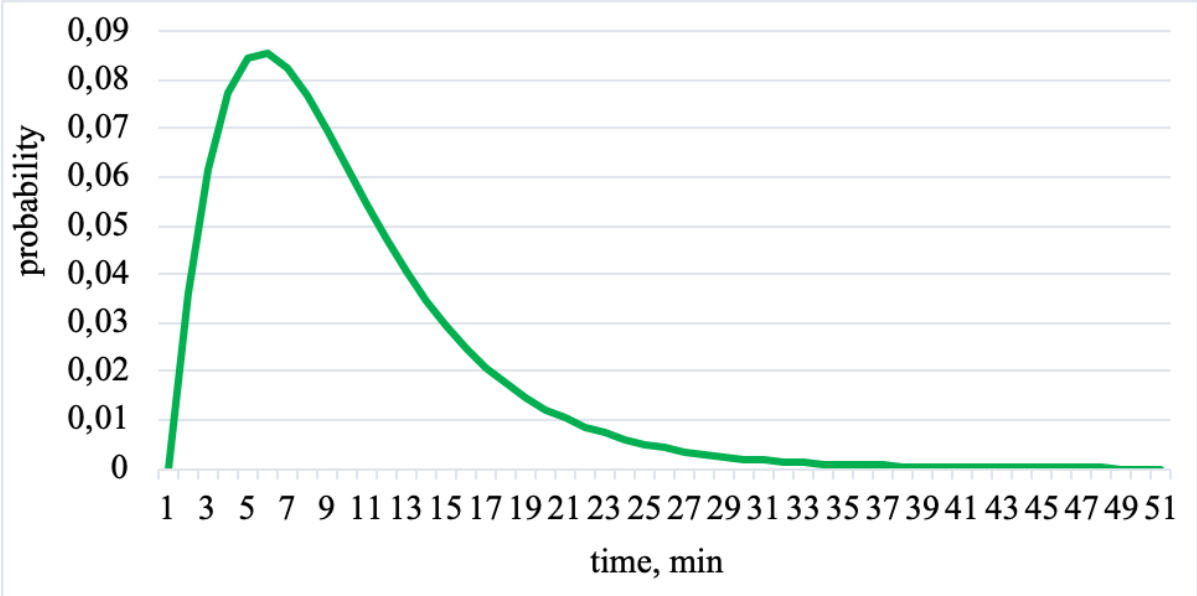


Рис. 4.4. Графік щільності розподілу часу виконання процесу забезпечення ліцензійної безпеки програмного продукту при використанні GERT-мережі

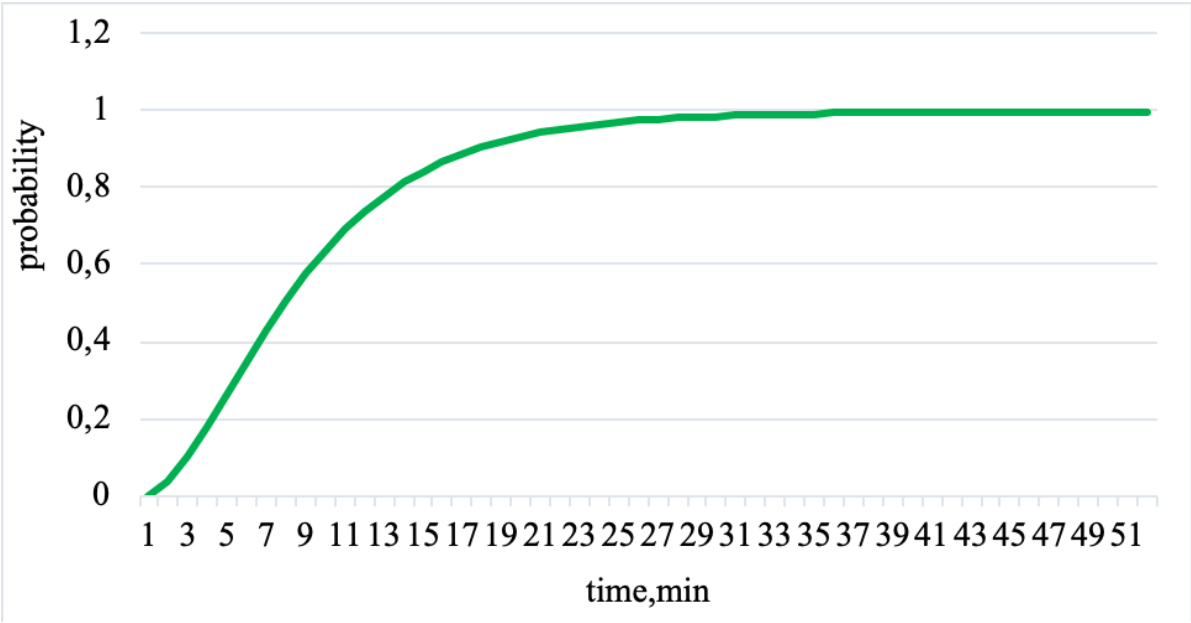


Рис. 4.5. Графік функції розподілу процесу забезпечення ліцензійної безпеки програмного продукту при використанні GERT-мережі

Математичне сподівання і дисперсія отриманих функцій розраховуються відповідно за формулами:

$$\mu = W_E(t)\mathrm{d}t\,|_{t=0}, = 3.52 \tag{4.6}$$

$$\sigma^2 = W_E(t)\mathrm{d}^2t\,|_{t=0} - \mu^2 = \ 117.01 \tag{4.7}$$

4.3 Нарощування складності GERT-мережі

Дослідивши розроблену GERT-мережу, можна помітити, що для того, щоб знайти правильний ключ (тобто правильне початкове маркування) необхідно перебрати лише 16 значень. Використання ключів, довжина яких менше 256 біт, на сьогоднішній день є недостатнім для забезпечення захисту. Отже, запропонована GERT-мережа не годиться для використання на практиці. Щоб збільшити довжину ключа при використанні GERT-мереж, наведену елементарну мережу пропонується нарощувати (масштабувати).

В рамках дослідження було прийнято використовувати 2 типи масштабування: горизонтальне (рис. 4.6) і вертикальне (рис. 4.7).

При горизонтальному масштабуванні частина ключа вклинюється в процес обробки конкретних бітів. Наприклад, з рис. 4.6 видно, що ми перейдемо в стан 5 в разі, коли не тільки біти 0 і 1 валідні, а й коли біти 4-7 валідні (стан 1‘-4’). Характеристики переходів між станами GERT-мережі описані в табл. 4.2. Як видно з даної таблиці, розвинена GERT-мережа позбавлена гілки (0-5). У свою чергу, було додано ряд гілок, що з’єднуються з вершинами 0 і 5 таким же чином, як це робилося для зв’язку 0-ї вершини з вершинами 1-4 і вершини 9 з вершинами 3, 5, 8 відповідно.

Згідно до табл. 4.2, результуюча *W*-функція горизонтального масштабування має такий вигляд:

$$W_{EH}(s) = \frac{W'_{EH}(s)}{1 - W''_{EH}(s)}$$

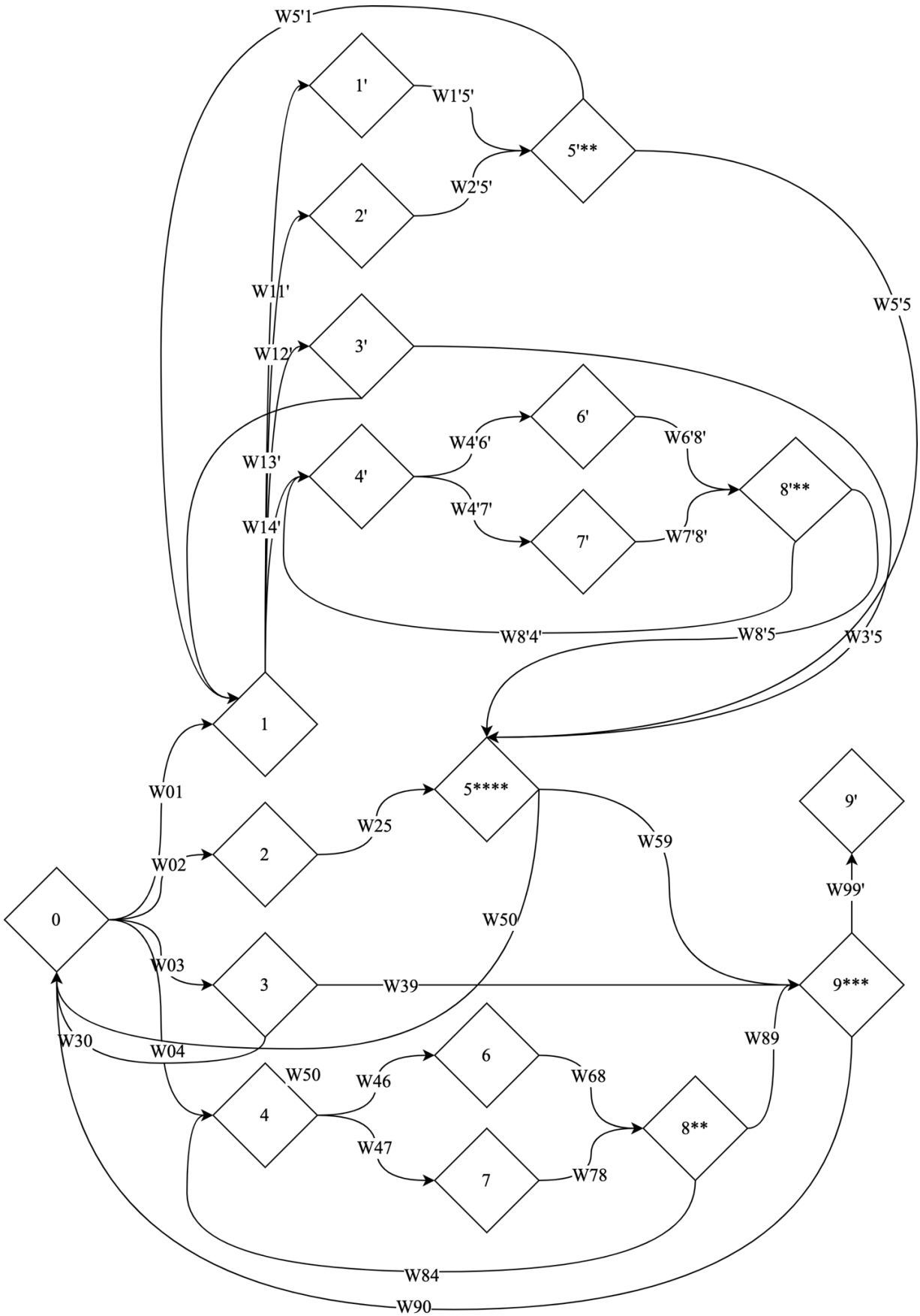


Рис. 4.6. Розроблена GERT-мережа процесу ліцензійної безпеки програмного забезпечення при горизонтальному масштабуванні

$$W'_{EH}(s) = W_{01}W_{11'}W_{1'5'}W_{5'5}W_{59}W_{99'} + W_{01}W_{12'}W_{2'5'}W_{5'5}W_{59}W_{99'} +$$
$$+W_{01}W_{13'}W_{3'5}W_{59}W_{99'} + W_{01}W_{14'}W_{4'6'}W_{6'8'}W_{8'5}W_{59}W_{99'} +$$
$$+W_{01}W_{14'}W_{4'7'}W_{7'8'}W_{8'5}W_{59}W_{99'} + W_{02}W_{25}W_{59}W_{99'} + W_{03}W_{39}W_{99'} +$$
$$+W_{04}W_{46}W_{68}W_{89}W_{99'} + W_{04}W_{47}W_{78}W_{89}W_{99'}$$

$$W''_{EH}(s) = W_{01}W_{11'}W_{1'5'}W_{5'1} + W_{01}W_{12'}W_{2'5'}W_{5'1} + W_{01}W_{11'}W_{1'5'}W_{5'5}W_{50} + W_{01}W_{12'}W_{2'5'}W_{5'5}W_{50} +$$
$$+W_{01}W_{11'}W_{1'5'}W_{5'5}W_{59}W_{90} + W_{01}W_{12'}W_{2'5'}W_{5'5}W_{59}W_{90} + W_{01}W_{14'}W_{4'6'}W_{6'8'}W_{8'4'} + W_{01}W_{13'}W_{3'1} +$$
$$+W_{01}W_{13'}W_{3'5}W_{50} + W_{01}W_{13'}W_{3'5}W_{59}W_{90} + W_{01}W_{14'}W_{4'7'}W_{7'8'}W_{8'4'} + W_{01}W_{14'}W_{4'6'}W_{6'8'}W_{8'5}W_{50} +$$
$$+W_{01}W_{14'}W_{4'7'}W_{7'8'}W_{8'5}W_{50} + W_{01}W_{14'}W_{4'6'}W_{6'8'}W_{8'5}W_{59}W_{90} + W_{01}W_{14'}W_{4'7'}W_{7'8'}W_{8'5}W_{59}W_{90} +$$
$$+W_{02}W_{25}W_{50} + W_{03}W_{30} + W_{04}W_{46}W_{68}W_{84} + W_{04}W_{47}W_{78}W_{84} + W_{02}W_{25}W_{59}W_{90} + W_{03}W_{39}W_{90} +$$
$$+W_{04}W_{46}W_{68}W_{89}W_{90} + W_{04}W_{47}W_{78}W_{89}W_{99}$$

Результуючий вираз розрахунку еквівалентних передавальних функцій матиме наступний вигляд:

$$W_{EH}(t) = \frac{p_{1,1,2,3,3,3}^{1,1,5,7,7,17} + p_{1,1,2,3,3,3}^{1,2,6,7,7,17} + p_{1,1,3,3,3}^{1,3,9,7,17} + p_{1,1,1,2,3,3,3}^{1,4,11,13,15,7,17} + p_{1,1,1,2,3,3,3}^{1,4,13,14,15,7,17}}{+p_{1,2,3,3}^{2,2,7,17} + p_{1,3,3}^{3,9,17} + p_{1,1,2,3,3}^{4,11,13,15,17} + p_{1,1,2,3,3}^{4,12,14,15,17})}$$
$$1 - \left(\begin{array}{c} p_{1,1,2,4}^{1,1,5,8} + p_{1,1,2,4}^{1,2,6,8} + p_{1,1,4}^{1,3,10} + p_{1,1,1,2,4}^{1,4,11,13,16} + p_{1,1,1,2,4}^{1,4,12,14,16} + \\ +p_{1,1,2,3,4}^{1,1,5,7,8} + p_{1,1,2,3,4}^{1,2,6,7,8} + p_{1,1,3,4}^{1,3,9,8} + p_{1,1,1,2,3,4}^{1,4,11,13,15,8} + p_{1,1,1,2,3,4}^{1,4,12,14,15,8} + \\ +p_{1,1,2,3,3,4}^{1,1,5,7,7,18} + p_{1,1,2,3,3,4}^{1,2,6,7,7,18} + p_{1,1,3,3,4}^{1,3,9,7,18} + p_{1,1,1,2,3,3,4}^{1,4,11,13,15,7,18} + p_{1,1,1,2,3,3,4}^{1,4,12,14,15,7,18} + \\ +p_{1,2,4}^{2,6,8} + p_{1,4}^{3,10} + p_{1,1,2,4}^{4,11,13,16} + p_{1,1,2,4}^{4,12,14,16} + \\ +p_{1,2,3,4}^{2,6,7,18} + p_{1,3,4}^{3,9,18} + p_{1,1,2,3,4}^{4,11,13,15,18} + p_{1,1,2,3,4}^{4,12,14,15,18} \end{array} \right)$$

При вертикальному масштабуванні відбувається модифікація переходів на перші стани (наприклад, (0-1), (0-2)), шляхом додавання нових гілок. При цьому, загальна ймовірність переходу зі стану 0 зменшується пропорційно довжині ключа. В даному випадку, складність графа залишається подібною початковій. Характеристики переходів між станами GERT-мережі описані в табл. 4.3.

Таблиця 4.2. Характеристики переходів між станами GERT-мережі процесу ліцензійної безпеки програмного забезпечення при горизонтальному масштабуванні

№ з/п	Гілка	*W*-функція	Ймовірність переходу	Коефіцієнти щільності ймовірності (2)
1.	(0, 1) (1, 1')	W_{01}, $W_{11'}$	P_1	$k=k_1$; $\theta=\theta_1$
2.	(0, 2) (1, 2')	W_{02}, $W_{12'}$	P_2	$k=k_1$; $\theta=\theta_1$
3.	(0, 3) (1, 3')	W_{03}, $W_{13'}$	P_3	$k=k_1$; $\theta=\theta_1$
4.	(0, 4) (1, 4')	W_{04}, $W_{41'}$	$P_4=1-P_1-P_2-P_3$	$k=k_1$; $\theta=\theta_1$
5.	(1`, 5`)	$W_{1'5'}$	P_5	$k=k_2$; $\theta=\theta_2$
6.	(2, 5) (2', 5')	W_{25}, $W_{2'5'}$	P_6	$k=k_2$; $\theta=\theta_2$
7.	(5, 9) (5', 5)	W_{59}, $W_{5'5}$	P_7	$k=k_3$; $\theta=\theta_3$
8.	(5, 0) (5`, 1)	W_{50}, $W_{5'1}$	$P_8=1-P_7$	$k=k_4$; $\theta=\theta_4$
9.	(3, 9) (3', 5)	W_{39}, $W_{3'5}$	P_9	$k=k_3$; $\theta=\theta_3$
10.	(3, 0) (3', 1)	W_{30}, $W_{3'1}$	$P_{10}=1-P_9$	$k=k_4$; $\theta=\theta_4$
11.	(4, 6) (4', 6')	W_{46}, $W_{4'6'}$	P_{11}	$k=k_1$; $\theta=\theta_1$
12.	(4, 7) (4', 7')	W_{47}, $W_{4'7'}$	$P_{12}=1-P_{11}$	$k=k_1$; $\theta=\theta_1$
13.	(6, 8) (6', 8')	W_{68}, $W_{6'8'}$	P_{13}	$k=k_2$; $\theta=\theta_2$
14.	(7, 8) (7', 8')	W_{78}, $W_{7'8'}$	P_{14}	$k=k_2$; $\theta=\theta_2$
15.	(8, 9) (8', 5)	W_{89}, $W_{8'5}$	P_{15}	$k=k_3$; $\theta=\theta_3$
16.	(8, 4) (8', 4')	W_{84}, $W_{8'4'}$	$P_{16}=1-P_{15}$	$k=k_4$; $\theta=\theta_4$
17.	(9, 9`)	$W_{99'}$	P_{17}	$k=k_3$; $\theta=\theta_3$
18.	(9, 0)	W_{90}	$P_{18}=1-P_{17}$	$k=k_4$; $\theta=\theta_4$

Згідно табл. 4.3, результуюча *W*-функція вертикального масштабування має наступний вигляд:

$$W_{EV}(s)=\frac{W'_{EV}(s)}{1-W''_{EV}(s)}$$

$$W'_{EV}(s) = W_{01}W_{15}W_{59}W_{99'} + W_{02}W_{25}W_{59}W_{99'} + W_{03}W_{39}W_{99'} +$$
$$+W_{04}W_{46}W_{68}W_{89}W_{99'} + W_{04}W_{47}W_{78}W_{89}W_{99'} + W_{01'}W_{1'5'}W_{5'9}W_{99'} +$$
$$+W_{02'}W_{2'5'}W_{5'9}W_{99'} + W_{03'}W_{3'9}W_{99'} + W_{04'}W_{4'6'}W_{6'8'}W_{8'9}W_{99'} + W_{04'}W_{4'7'}W_{7'8'}W_{8'9}W_{99'}$$

$$W''_{EV}(s) = W_{01}W_{15}W_{50} + W_{02}W_{25}W_{50} + W_{03}W_{30} + W_{04}W_{46}W_{68}W_{84} + W_{04}W_{47}W_{78}W_{84} +$$
$$+W_{01}W_{15}W_{59}W_{90} + W_{02}W_{25}W_{59}W_{90} + W_{03}W_{39}W_{90} + W_{04}W_{46}W_{68}W_{89}W_{90} + W_{04}W_{47}W_{78}W_{89}W_{99'} +$$
$$+W_{01'}W_{1'5'}W_{5'0} + W_{02'}W_{2'5'}W_{5'0} + W_{03'}W_{3'0} + W_{04'}W_{4'6'}W_{6'8'}W_{8'4'} + W_{04'}W_{4'7'}W_{7'8'}W_{8'4'} + W_{01'}W_{1'5'}W_{5'9}W_{90} +$$
$$+W_{02'}W_{2'5'}W_{5'9}W_{90} + W_{03'}W_{3'9}W_{90} + W_{04'}W_{4'6'}W_{6'8'}W_{8'9}W_{90} + W_{04'}W_{4'7'}W_{7'8'}W_{8'9}W_{99'}$$

Результуючий вираз розрахунку еквівалентних передавальних функцій матиме наступний вигляд:

$$W_{EV}(t) = \frac{\left(\begin{array}{c} p_{1,2,3,3}^{1,5,7,17} + p_{1,2,3,3}^{2,6,7,17} + p_{1,3,3}^{3,9,17} + p_{1,1,2,3,3}^{4,11,13,15,17} + p_{1,1,2,3,3}^{4,12,14,15,17} + \\ + p_{1,2,3,3}^{1',5,7,17} + p_{1,2,3,3}^{2',2,7,17} + p_{1,3,3}^{3',9,17} + p_{1,1,2,3,3}^{4',11,13,15,17} + p_{1,1,2,3,3}^{4',12,14,15,17} \end{array} \right)}{1 - \left(\begin{array}{c} p_{1,2,4}^{1,5,8} + p_{1,2,4}^{2,6,8} + p_{1,4}^{3,10} + p_{1,1,2,4}^{4,11,13,16} + p_{1,1,2,4}^{4,12,14,16} + p_{1,2,3,4}^{1,5,7,18} + \\ + p_{1,2,3,4}^{2,6,7,18} + p_{1,3,4}^{3,9,18} + p_{1,1,2,3,4}^{4,11,13,15,18} + p_{1,1,2,3,4}^{4,12,14,15,18} + \\ + p_{1,2,4}^{1',5,8} + p_{1,2,4}^{2',6,8} + p_{1,4}^{3',10} + p_{1,1,2,4}^{4',11,13,16} + p_{1,1,2,4}^{4',12,14,16} + p_{1,2,3,4}^{1',5,7,18} + \\ + p_{1,2,3,4}^{2',6,7,18} + p_{1,3,4}^{3',9,18} + p_{1,1,2,3,4}^{4',11,13,15,18} + p_{1,1,2,3,4}^{4',12,14,15,18} \end{array} \right)}$$

Використовуючи щільність розподілу ймовірностей (4.4), отримуємо графік розподілу щільності ймовірності для розробленої початкової GERT-мережі процесу ліцензійної безпеки програмного забезпечення і його модифікацій (при горизонтальному і вертикальному масштабуванні). Результати наведені на рис. 4.8. При цьому, ймовірно вибрали такий спосіб:

$$P_1 = P_2 = P_3 = P_4 = 0.25; P_5 = P_6 = 1.0; P_7 = 0.3; P_9 = 0.5; P_{11} = 0.5; P_{13} = P_{14} = 1.0; P_{15} = 0.3; P_{17} = 0.2.$$

Математичне сподівання і дисперсія отриманих функцій:

– для вертикального масштабування:

$$\mu = W_E(t)\mathrm{d}t\,|_{t=0}, = 3.71$$

$$\sigma^2 = W_E(t)\mathrm{d}^2t\,|_{t=0} - \mu^2 = 130.01$$

– для горизонтального масштабування:

$$\mu = W_E(t)\mathrm{d}t\,|_{t=0}, = 12.11$$

$$\sigma^2 = W_E(t)\mathrm{d}^2t\,|_{t=0} - \mu^2 = 230.01$$

Таблиця 4.3. Характеристики переходів між станами GERT-мережі процесу ліцензійної безпеки програмного забезпечення при вертикальному масштабуванні

№ з/п	Гілка	W-функція	Ймовірність переходу	Коефіцієнти щільності ймовірності (3.3)
1.	(0, 1), (0, 1`)	W_{01}, $W_{01`}$	$P_1=P_{1`}$	$k=k_1$; $\theta=\theta_1$
2.	(0, 2) , (0, 2`)	W_{02}, $W_{02`}$	$P_2=P_{2`}$	$k=k_1$; $\theta=\theta_1$
3.	(0, 3) , (0, 3`)	W_{03}, $W_{03`}$	$P_3=P_{3`}$	$k=k_1$; $\theta=\theta_1$
4.	(0, 4) , (0, 4`)	W_{04}, $W_{04`}$	$P_4=P_{1`}=(1-P_1-P_2-P_3-P_{1`}-P_{2`}-P_{3`})/2$	$k=k_1$; $\theta=\theta_1$
5.	(1, 5) , (1`, 5`)	W_{15}, $W_{1`5`}$	P_5	$k=k_2$; $\theta=\theta_2$
6.	(2, 5) , (2`, 5`)	W_{25}, $W_{2`5`}$	P_6	$k=k_2$; $\theta=\theta_2$
7.	(5, 9) , (5`, 9`)	W_{59}, $W_{5`9}$	P_7	$k=k_3$; $\theta=\theta_3$
8.	(5, 0) , (5`, 0)	W_{50}, $W_{5`0}$	$P_8=1-P_7$	$k=k_4$; $\theta=\theta_4$
9.	(3, 9) , (3`, 9`)	W_{39}, $W_{3`9`}$	P_9	$k=k_3$; $\theta=\theta_3$
10.	(3, 0) , (3`, 0)	W_{30}, $W_{3`0}$	$P_{10}=1-P_9$	$k=k_4$; $\theta=\theta_4$
11.	(4, 6) , (4`, 6`)	W_{46}, $W_{4`6`}$	P_{11}	$k=k_1$; $\theta=\theta_1$
12.	(4, 7) , (4`, 7`)	W_{47}, $W_{4`7`}$	$P_{12}=1-P_{11}$	$k=k_1$; $\theta=\theta_1$
13.	(6, 8) , (6`, 8`)	W_{68}, $W_{6`8`}$	P_{13}	$k=k_2$; $\theta=\theta_2$
14.	(7, 8) , (7`, 8`)	W_{78}, $W_{7`8`}$	P_{14}	$k=k_2$; $\theta=\theta_2$
15.	(8, 9) , (8`, 9`)	W_{89}, $W_{8`9}$	P_{15}	$k=k_3$; $\theta=\theta_3$
16.	(8, 4) , (8`, 4`)	W_{84}, $W_{8`4`}$	$P_{16}=1-P_{15}$	$k=k_4$; $\theta=\theta_4$
17.	(9, 9`)	$W_{99`}$	P_{17}	$k=k_3$; $\theta=\theta_3$
18.	(9, 0)	W_{90}	$P_{18}=1-P_{17}$	$k=k_4$; $\theta=\theta_4$

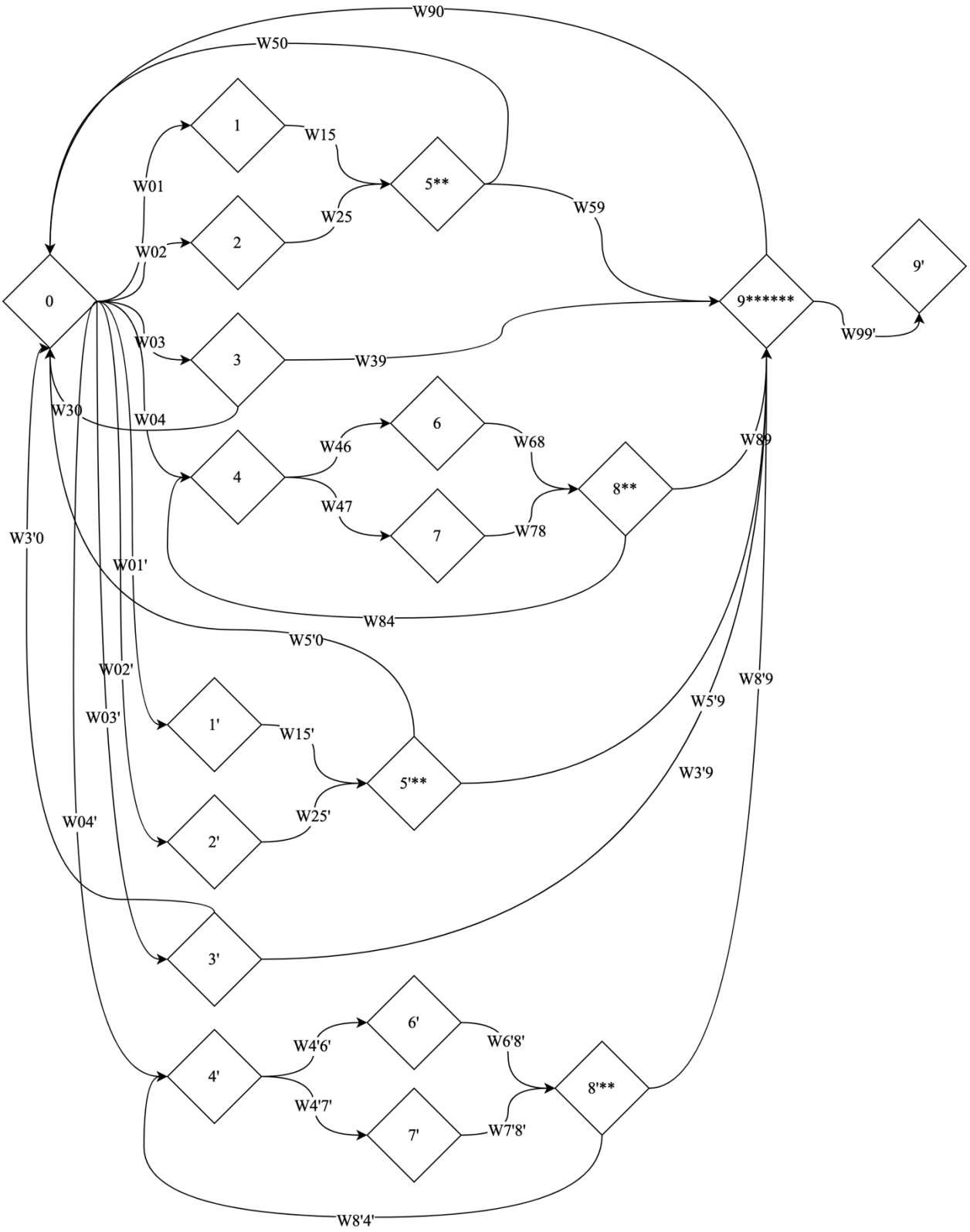


Рис. 4.7. Розроблена GERT-мережа процесу ліцензійної безпеки програмного забезпечення при вертикальному масштабуванні

Наведені графіки щільності розподілу ймовірностей для GERT-мереж

процесу ліцензійної безпеки програмного забезпечення на рис. 4.8 використовують такі коефіцієнти щільності ймовірності: k = [2, 4, 4, 3] та θ = [2.7, 1.8, 1.8, 3.5]. Інтегрувавши щільність розподілу ймовірностей, отримаємо функції розподілу, графіки яких відображені на рис. 4.9.

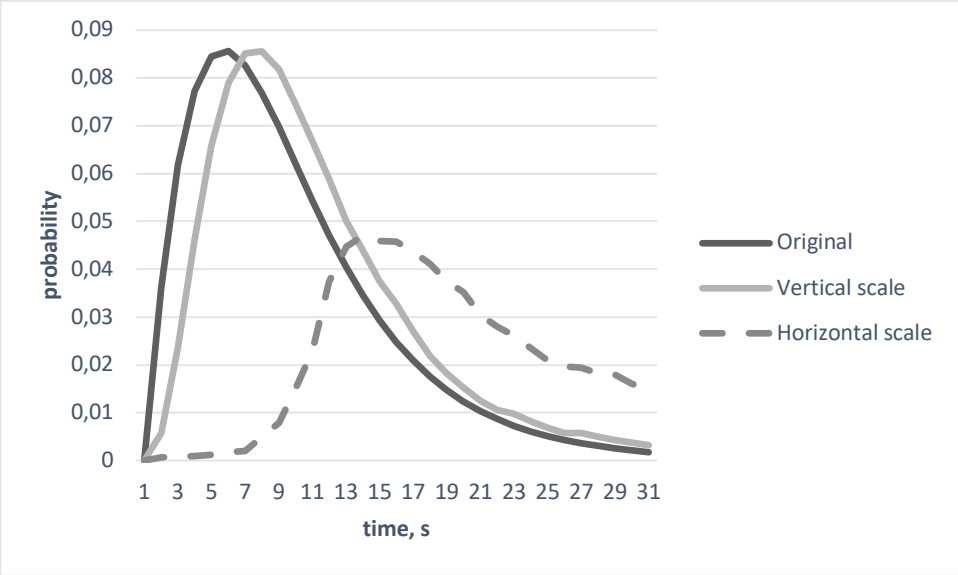


Рис. 4.8. Порівняльні графіки щільності розподілу часу виконання процесу забезпечення ліцензійної безпеки програмного продукту при використанні GERT-мережі

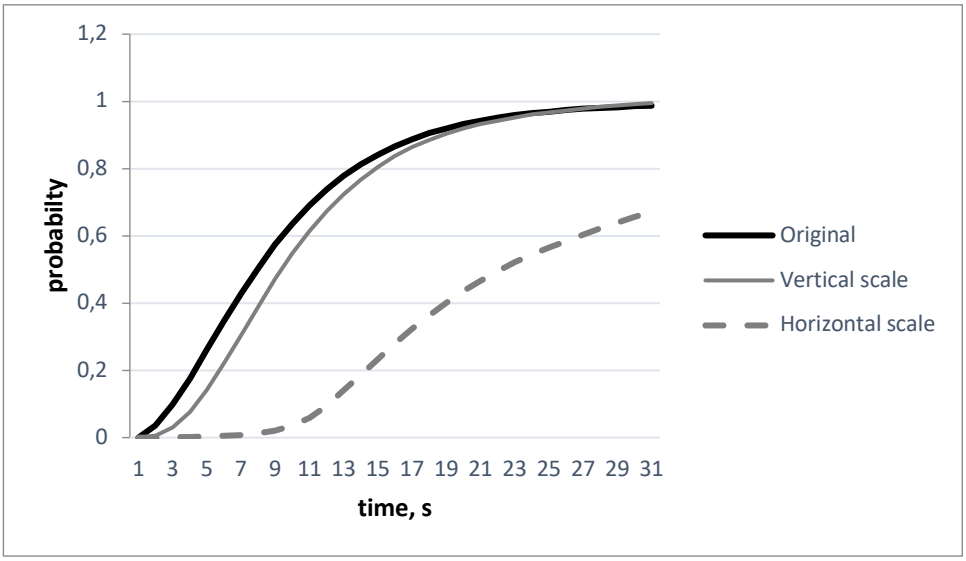


Рис. 4.9. Порівняльні графіки функцій розподілу процесу забезпечення ліцензійної безпеки програмного продукту при використанні GERT-мережі

Результати моделювання показали:

– при вертикальному масштабуванні, в зв’язку з використанням паралельного процесу час обробки даних практично не змінився. Це дає передумови використання даного типу масштабування при критичності часу виконання процесу, а також при необхідності істотного нарощування довжини ліцензійного ключа на слабких пристроях (наприклад, вбудованих пристроях і IoT);

– при горизонтальному масштабуванні, в зв’язку з використанням додаткових послідовних процесів обробки даних, час обробки значно збільшився (в 3.43 рази). Однак, введення додаткових послідовних дій в силу своєї природи збільшує час аналізу алгоритму поведінки. Це дає передумови використання даного типу масштабування при використанні прикладного програмного забезпечення, де час запуску програми не критичний.

Висновки за розділом 4

1. Розроблено модель безпечного переходу і кодування ліцензійних ідентифікаторів на основі математичного апарату GERT-мереж з парадигмою Гама-розподілу, що дозволило підвищити точність результатів моделювання. Дана логіка впроваджується в залежності від ідентифікаційного або серійного номера.

2. Розроблено методологію масштабування розробленої математичної моделі. Показана доцільність використання кожного типу масштабування з урахуванням критичності часу виконання здійснення перевірки безпеки програмного забезпечення на основі ліцензійних ідентифікаторів. Так, при вертикальному масштабуванні, в зв'язку з використанням паралельного процесу обробка даних, час обробки даних практично не змінилося. Це дає передумови використання даного типу масштабування при критичності часу виконання процесу, а також при необхідності істотного нарощування

довжини ліцензійного ключа на слабких пристроях (наприклад, вбудованих пристроях і IoT). При горизонтальному масштабуванні, в зв'язку з використанням додаткових послідовних процесів обробки даних, час обробки значно збільшилася (в 3.43 рази). Однак, введення додаткових послідовних дій збільшує час аналізу алгоритму поведінки. Це дає передумови використання даного типу масштабування при використанні прикладного програмного забезпечення, де час запуску програми не критично.

РОЗДІЛ 5. МЕТОД ФОРМУВАННЯ ЦИФРОВОГО ІДЕНТИФІКАТОРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЗАХИСТУ АВТОРСЬКИХ ПРАВ

5.1 Дослідження методів захисту авторських прав в програмному забезпеченні

Відповідно до ст. 433 Цивільного кодексу України та ст. 18 Закону України «Про авторське право і суміжні права» [93], комп'ютерні програми (програмне забезпечення) охороняються як літературні твори. Така охорона поширюється на комп'ютерні програми незалежно від способу чи форми їх вираження. Підставою для віднесення програмного забезпечення (ПЗ) до літературних творів служить певна спільнота відображення рядків літературного твору і комп'ютерної програми: і рядки літературного твору, і рядки комп'ютерної програми автор наповнює символами-літерами або символами-операторами. Тотожність творчого процесу щодо створення форм авторських творів літератури і комп'ютерних програм і стала визначаючою для вибору форми захисту ПЗ [1].

Охороняється ПЗ як об'єкт авторського права, тобто без виконання якихось особливих формальностей щодо нього і незалежно від його завершеності, призначення, цінності тощо, а також способу чи форми їх вираження.

Існують певні складності охорони ПЗ.

Перша складність закладена вже в самому об'єкті – це і літературний твір, і одночасно – набір інструкцій. Тобто це і якийсь набір символів (розповідь – теж літературний твір) і набір інструкцій для деяких дій (а це вже, швидше, алгоритм або спосіб дій).

Друга складність реєстрації авторського права на ПЗ закладена також і в самому обсязі програмного коду, який поданий на реєстрацію. Особливо з огляду на те, що він видається на реєстрацію в друкованому вигляді. Крім того, до матеріалів заявки на реєстрацію авторського права на комп'ютерну програму, крім звичайних документів, додається і керівництво по використанню програми [1, 90].

Третя складність закладена в тому, що сама реєстрація авторського права – це депонування програми, тобто – це збереження комп'ютерної програми з чітким визначенням терміну її здачі на зберігання і утримання зданого продукту. Авторське право на ПЗ не поширюється на закладені в ньому:

– ідеї;

– процеси;

– методи діяльності або математичні концепції як такі, на яких заснована комп'ютерна програма (в тому числі і на пару, тобто на ту частину програми, яка забезпечує діалог з користувачем і сумісність її з елементами апаратури);

– логіку роботи програми;

– алгоритми роботи програми;

– мови програмування.

Таким чином, авторським правом захищається текст (код) програми, а не функції, які вона виконує.

Аналіз літератури [81, 87, 94, 165] показав, що одним з механізмів захисту авторських прав на програмне забезпечення є поширення ліцензійної угоди на використання програм. Забезпечення відповідності використання програмних продуктів вимогам ліцензійних угод є проблемою, з якою не дуже просто впоратися. При цьому існує безліч різних рекомендацій технічного, соціального, психологічного та інших напрямків з ефективного використання зазначеного механізму захисту, таких як:

– незалежний аудит ПЗ;

– використання чітко сформульованої політики з відлагодженою системою сповіщення;

– інструменти для резервного копіювання;

– автоматизація процесу з використанням інструментів активного виявлення, які б знаходили як дозволені, так і недозволені встановлені програми.

Також останнім часом в науковій літературі [81, 87] все більше уваги приділяється захисту ПЗ за допомогою впроваджених цифрових водяних знаків, цифрового підпису та інших позначок, що підтверджують авторство і спрощують процес автоматичної ідентифікації та верифікації ПЗ.

Проведені дослідження показали, що одним з основних недоліків розробляюваних в даний час технічних систем ідентифікації та ліцензування ПЗ є нехтування формальними даними про комп'ютерні системи, на які ліцензійне ПЗ встановлюється. Це призводить до можливості тиражування цифрового ідентифікатора, тобто до порушення авторських прав.

Аналіз літератури [5, 18, 29, 40, 81, 94] показав, що одним з найбільш ефективних засобів захисту інтелектуальної власності на програмне забезпечення є ліцензійний ключ захисту застосунків (ЛКЗЗ). Зазвичай ключ застосовується під час встановлення. Програма-інсталятор застосовує алгебраїчні обчислення до введеного ключа для перевірки його на справжність. Наприклад, алгоритму необхідно визначити, що введений ключ повинен містити 5 чисел, сума яких дорівнює 25, і що ключ також повинен містити 3-5 літер так, що після переведення їх в числові еквіваленти отримаємо суму 42 [14, 193].

Проведені дослідження [5, 25, 30, 33, 81, 85, 94, 104, 220] показали, що в даний час існує ряд підходів до формування ЛКЗЗ. Їх основою є відомі криптографічні алгоритми, що дозволяють формувати послідовності різного рівня складності і стійкості. Слід зауважити, що у більшості фірм-розробників ПЗ ця інформація є конфіденційною. У той же час, аналіз

відкритих інтернет-ресурсів [14] показав масову пропозицію на програмне забезпечення, що дозволяє формувати так звані keygen (генератори ключів), які пишуться як окремими програмістами, так і хакерськими групами, наприклад, CORE, ORiON, ZWT, REVOLUTiON, XNTeam, Fight For Fun та ін., що спеціалізуються на зламі програмного забезпечення. Іноді такі групи заявляють про себе також тим, що включають свою назву в згенерований ключ у відкритому або зашифрованому вигляді [14].

Таким чином, в умовах повсюдного використання комп'ютерних, телекомунікаційних та інших комп'ютеризованих засобів, а також постійного вдосконалення і оновлення їх програмного забезпечення, досить актуальним завданням є захист інтелектуальної власності та авторських прав на програмні продукти (програмне забезпечення).

Особливо гострою ця проблема виглядає в Україні, де компанії-розробники ПЗ несуть фінансові втрати через несанкціоноване (незаконне) використання авторських прав на створені програмні продукти.

Аналіз літератури показав актуальність розробки генератора ЛКЗЗ, що реалізує сучасні принципи контролю дозволів виконання прикладного коду, який би дозволив мінімізувати ризик хакерської підробки, і тим самим підвищив рівень захисту авторських прав на інтелектуальну власність. Рішення поставленого завдання неможливо без розробки відповідного програмного комплексу.

5.2 Розробка методу формування цифрового ідентифікатора програмного забезпечення для захисту авторських прав

В рамках дослідження сформовано модель процесу формування цифрового ідентифікатора програмного забезпечення. Її структурна схема представлена на рис. 5.1.

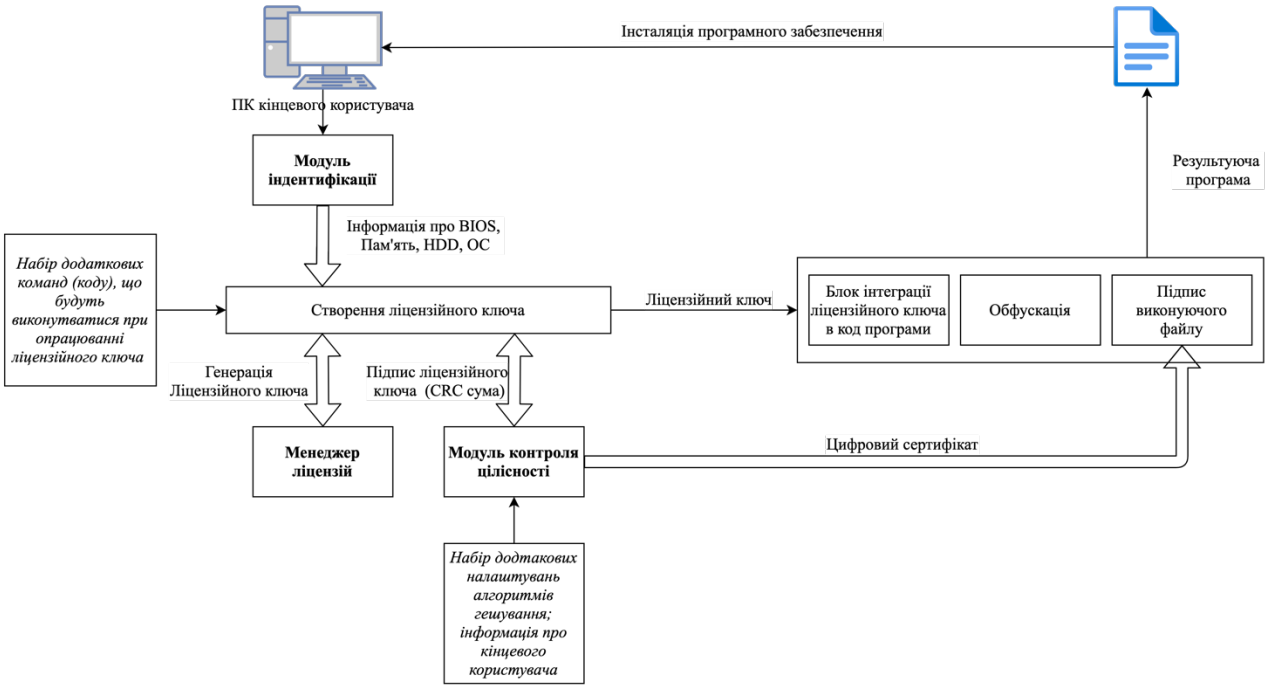


Рис. 5.1. Структурна схема процесу формування цифрового ідентифікатора програмного забезпечення

Як видно з рисунка, основними складовими розроблюваної моделі є:

– менеджер ліцензії. Цей модуль є розпорядником усіх інших компонентів моделі. У його завдання входить порівняння унікальних даних персонального комп’ютера, на якому запущена копія продукту, з тими, що закодовані всередині продукту; виклик модуля контролю цілісності; генерація ліцензійного ключа на основі даних про цільову комп’ютерну систему;

– модуль контролю цілісності. Цілі даного компонента: підпис ліцензійного ключа (CRC-сума); підпис програмного продукту цифровим сертифікатом (X.509) [175, 236] кінцевого користувача; періодична перевірка цілісності модулів системи безпеки на основі ліцензійного ключа в процесі експлуатації програмного продукту. Такі перевірки є додатковим бар’єром на шляху зловмисника, якщо він спробує модифікувати або фальсифікувати компоненти системи;

– модуль ідентифікації. У завдання цього компонента входить отримання даних, що однозначно ідентифікують персональний комп'ютер. В якості таких даних можуть виступати унікальні серійні номери апаратних комплектуючих ПК (наприклад, жорсткого диска, процесора, вбудованої камери, Bluetooth та WiFi модулі і т.д.), дати створення BIOS, ідентифікатор операційної системи (для ОС Windows - серійний номер);

– інтеграція ліцензійного ключа з програмним продуктом. Даний модуль отримує інформацію про можливий додатковий код, що виконується в режимі Runtime для індивідуального програмного продукту;

– обфускація вихідного коду виконуючого модуля.

Відповідно до процедур формування ліцензійного цифрового ідентифікатора необхідно виконати наступні кроки:

1. *Формування шаблону програмного коду, на основі якого генерується ліцензійний ключ*: запис в глобальну змінну MODE значення необхідного режиму роботи програми (список варіюється для індивідуального програмного продукту), від якого залежить функціональність (FULL – вся функціональність доступна; DEMO – доступна лише «базова» функціональність); порівняння типу продукту, версії, порівнянності поточного програмного продукту з встановленим типом операційної системи. У разі розбіжності програми можливі кілька варіантів розвитку ситуації залежно від вимог постачальника програмного забезпечення: виведення відповідного повідомлення користувачу і завершення програми; скидання функціональності до мінімально можливої (наприклад, демо-версія); скидання функціональності до лише-для-читання (без можливості збереження проміжних результатів).

2. *Отримання інформації про компоненти комп'ютерної системи кінцевого користувача*. Ця функція виконується шляхом створення програми-утиліти, яка запускається на системі кінцевого користувача і відправляє по захищеному каналу необхідну інформацію, щоб уникнути

перехоплення трафіку і його аналізу / зміни на сервер. Отримана інформація буде зберігатися для подальших дій. Програма-утиліта знаходиться на захищеному носії, доступ до якого є лише у авторизованого співробітника (даний варіант не виключає ймовірність проникнення зловмисників, але в контексті пропонованої реалізації, дії даного чинника ігноруються). Інформація, яка надходить на сервер про компоненти системи кінцевого користувача, має таку структуру:

– **накопичувачі**, наприклад: «WDC WD5000AAKX-001CA0; PCI\VEN_8086; DEV_1C02; SUBSYS_844D1043; REV_05\3». Отримання інформації про накопичувач для різних операційних систем:

- MacOS X: diskutil info disk0;
- Linux: smartctl -a /dev/yourdrive;
- Windows: wmic diskdrive get model,serialNumber,size.

– **процесори**, наприклад: «Intel(R) Pentium(R) CPU G620 @ 2.60GHz ACPI_HAL\PNP0C08\0». Отримання інформації про ЦПУ для різних операційних систем:

- MacOS X: sysctl -a | grep machdep.cpu;
- Linux: lscpu;
- Windows: wmic cpu get DeviceID, NumberOfCores, NumberOfLogicalProcessors.

– **ОС**, наприклад: «6.3.9600 N/A Build 9600; 00261-80443-28329-AA407». Для ОС Windows визначення даної інформації здійснюється за допомогою команди: wmic os get serialnumber;

– **BIOS**, наприклад: «American Megatrends Inc. 0409, 8/26/2011». Отримання інформації про BIOS для різних операційних систем:

- MacOS X: ioreg -l | grep IOPlatformSerialNumber;
- Linux: sudo dmidecode -t system | grep Serial;
- Windows: wmic bios get serialnumber.

3. *Формування результуючого ліцензійного ключа*. Для виконання даної функції в якості вхідних даних використовуються: інформація про систему кінцевого користувача; шаблон програмного коду, який буде виконуватися; розмір CRC-суми для валідації ліцензійного ключа.

5.3 Розробка алгоритму процесу формування цифрового ідентифікатора програмного забезпечення

Узагальнений алгоритм процесу формування цифрового ідентифікатора ПЗ має наступний вигляд:

1. Формування програмного коду (для .Net – файл машинного коду на асемблері (для ОС Windows), для JVM – java-файл), який буде виконуватися на системі кінцевого користувача з урахуванням отриманої інформації та наданого шаблону.

2. Отриманий код трансформується в відповідні машинні / байт-коди.

3. Для кожного байта отриманого коду виконуються наступні дії:

– пошук даного байта в текстовому поданні отриманої інформації;

– пошук даного байта в бінарному представленні отриманої інформації;

– якщо знайдений відповідний код в текстовому поданні, то в кінці формування ліцензійного ключа дописується 2 байта: «0» і «код пристрою», якщо в бінарному представленні - то 2 байта: «1» і «код пристрою», потім 2 байта, що відображають позицію знайденого символу. У разі, якщо код не був знайдений - 2 байта «00», а потім 2 байта, що відображають hex-представлення зазначеного байта (напр., Символ «A» має код «65», що в hex поданні - 41 - буде записано два байта «4» і «1»).

В кінці отриманого ліцензійного ключа додається 2 байта, що характеризують ступінь CRC-суми, наприклад «32». Далі йде CRC-сума в текстовому hex-поданні (для CRC32 – 8 символів). У процесі формування цифрового ідентифікатора ПЗ можна використовувати такі види пристроїв: 1

- накопичувачі; 2 - процесори; 3 - ОС; 4 - BIOS. У цьому випадку на виході отримуємо згенерований ліцензійний ключ, який буде вбудовуватися в програмний продукт. Вихідний програмний продукт проходить 3 фази перед надходженням до кінцевого користувача:

1. Отриманий ліцензійний ключ вбудовується в сегмент даних програми. При цьому сегмент даних повинен бути сформований таким чином, щоб ліцензійний ключ помістився в виділеному для цього блоці.

2. Обфускація, для зменшення ймовірності зламу алгоритму обробки ліцензійного ключа.

3. Підпис програмного продукту сертифікатом, щоб уникнути можливості редагування файлу з метою обійти алгоритм верифікації.

Далі отриманий програмний комплекс поставляється кінцевому користувачеві. При кожному запуску програма виконує наступні дії:

1. Отримує інформацію про компоненти поточної системи.

2. Верифікує ліцензійний ключ на основі CRC-суми.

3. Формує машинні коди для виконання на основі поточного ліцензійного ключа і параметрів системи.

4. Виконує сформовані машинні команди. У разі, якщо ліцензійний ключ виявився невалідним, або програма була запущена на іншому комп’ютері, то поточний програмний продукт аварійно завершить роботу.

Структура отриманого виконуючого .exe-файлу наведена на рис. 5.2.

Як видно з рисунка, пропонується ліцензійний ключ вбудовувати в блок даних. Весь виконуваний файл підписується електронним цифровим підписом з метою можливості перевірки цілісності та автентичності.

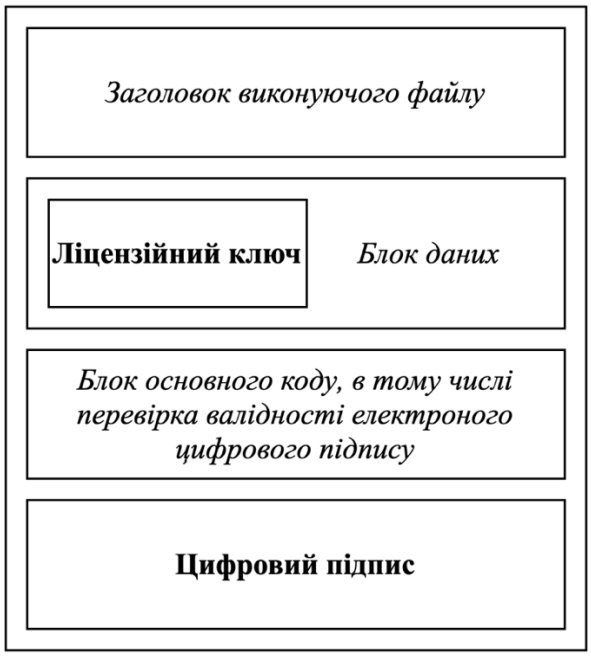


Рис. 5.2. Структура виконуючого .exe-файлу з вбудованим ліцензійним ключом

5.4 Розробка методу формування ліцензійного ключа

На основі аналізу літератури, переваг і недоліків існуючих механізмів формування та верифікації ліцензійного ключа, було зроблено спробу розроблення програмного комплексу, який зменшить ймовірність злочинного впливу на ліцензійний ключ. На основі аналізу переваг існуючих комплексів, в першу чергу, було зроблено спробу захистити програмний комплекс від злочинного впливу шляхом обфускації коду, покликаної збільшити час аналізу алгоритму програми зловмисником і використовувати підписи виконуваного файлу, щоб уникнути модифікування.

Базовими стратегіями захисту ліцензійного ключа від тиражування є наступні:

– використання інформації про складові обчислювальної системи кінцевого користувача;

– ядро ліцензійного ключа являє собою функцію (програмний код), яка буде виконуватися на системі кінцевого користувача.

Процес формування ліцензійного ключа:

– з боку кінцевого користувача передається інформація про систему на сервер по захищеному каналу зв’язку (наприклад, протокол https);

– сервер підбирає програмний код, який буде виконуватися на стороні клієнта, в залежності від переданих параметрів (ОС, тип ліцензії);

– сервер кодує обраний програмний код з додаванням CRC32-суми;

– сервер відправляє кінцевому користувачу ліцензійний ключ по захищеному каналу зв’язку;

– програма-клієнт кінцевого користувача зберігає закодований ключ з постійним його декодуванням при кожному запуску.

Процес кодування ліцензійного ключа представлений на рис. 5.3.

Опис алгоритму. Сервер на основі отриманої інформації генерує функцію, залежну від ОС клієнта, терміну дії ліцензії, яка буде виконуватися на стороні клієнта

Нехай в машинному коді дана функція буде представлена у вигляді набору байт: 0x60 0xA1 0x05 0x01 0xA3 0x61 0xC3.

Дані про характеристики комп’ютерної системи кінцевого користувача містять інформацію про процесор, ОС, вінчестер, ОЗУ, материнську плату. При цьому, кожен тип пристрою має свій ідентифікатор (0-6). Ідентифікатор «7» призначений для універсального пристрою, в тому випадку, якщо перші 7 не підійдуть. Приклад інформації про пристрої:

0 – Processor: BFEBFBF000202B7.

1 – OS: 00251-80443-12345-AA987.

2 – HDD: WD5000AAKX-001CA0.

3 – BIOS: 07/07/2002-VT8377-8233/5-JL7LVC0CC-00.

4 – CAMERA: EAB7A78FEC2B4487AADFD8A91C1CB782.

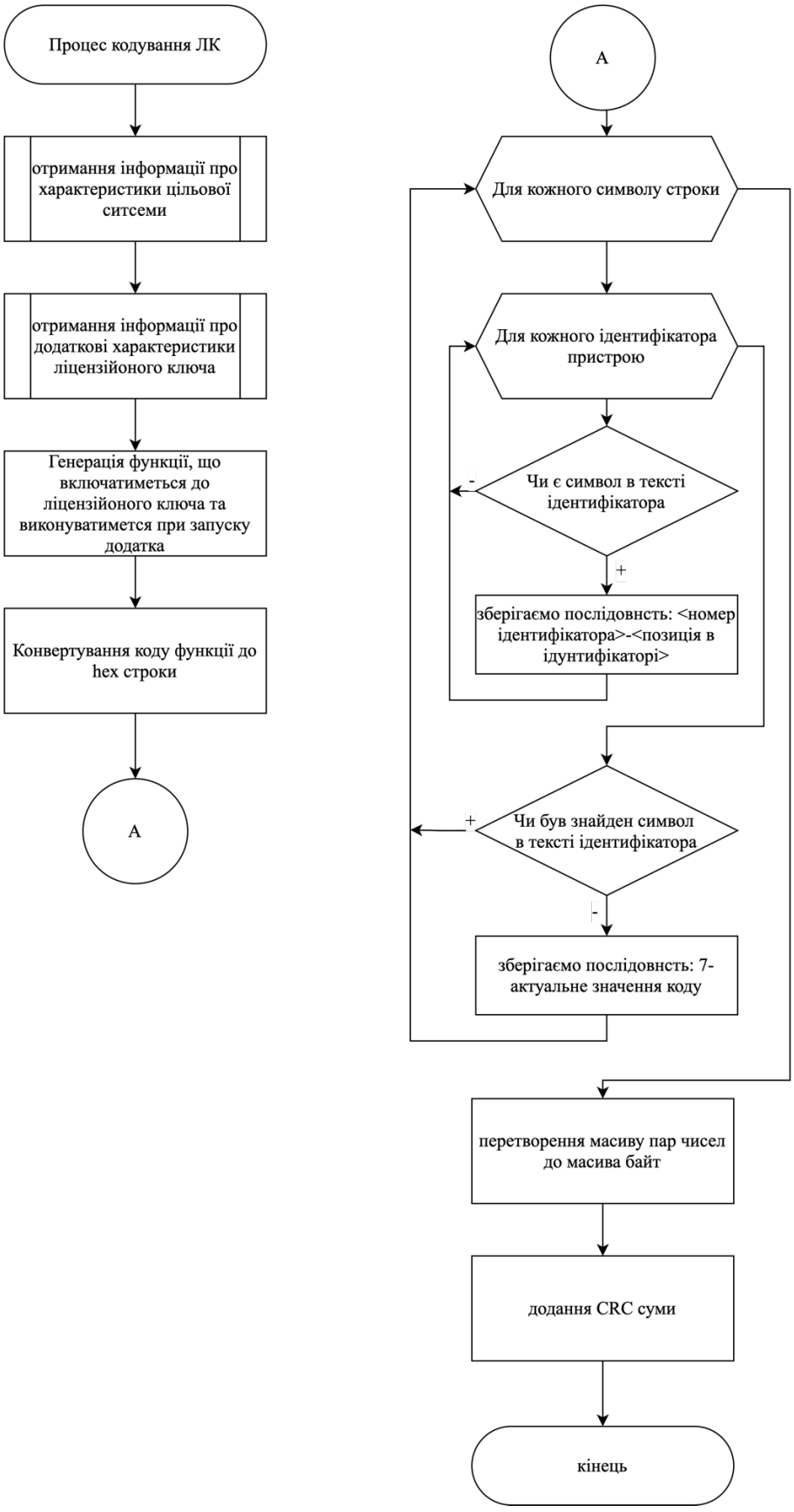


Рис. 5.3. Алгоритм процесу кодування ліцензійного ключа

Після того, як функція сформувалася, отримані байти перетворюються в рядок з відповідних hex-кодів: 60A10501A361C3. Для кожного символу з отриманого рядка:

– знаходимо відповідну позицію у вхідних даних. Якщо символ відсутній - вибираємо «7» (символ залишаємо без змін). Створюємо пари чисел «номер пристрою» - «позиція в номері пристрої». При цьому, позиція не повинна перевищувати значення 31 в зв’язку з обмеженням, використовуваним в наступному пункті, в 5 розрядів. Якщо заданий символ не знайдений в специфічній для кінцевого користувача інформації про пристрої, то пара формується за принципом «7» (код резервного пристрою) «-» «символ»:

7-6 0-8 1-18 1-4 0- 8 1-3 0-8 1-4 1-18 1-10 7-6 1-4 2-14 1-10;

– перетворюємо кожну пару чисел в один байт за принципом: старші 3 розряди - номер пристрою в вхідних байтах (0-7), решта - позиція (0-31) або саме число (0x0-0xF):

E6 08 32 24 08 23 08 24 32 2A E6 24 4E 2A.

В кінці відбувається додавання CRC32-суми. Даний тип гешу використовується лише для пересвідчення в тому, що передача по каналах зв’язку пройшла успішно. Захист від зламу на даному етапі не використовується, так як передбачається, що вміст ключа і алгоритм роботи декодувальника зловмиснику не відомий. В результаті отримуємо наступний ліцензійний ключ:

E6 08 32 24 08 23 08 24 32 2A E6 24 4E 2A AC 32 B7 5D.

Процес верифікації ліцензійного ключа:

– перевірка валідності виконуваного файлу (за допомогою утиліт signtool, certutil) на предмет його модифікації. Якщо валідність не підтвердилася - виведення відповідного повідомлення;

– перевірка CRC32 ліцензійного ключа для посвідчення в його цілісності. Якщо цілісність порушена - виведення відповідного повідомлення;

– декодування ліцензійного ключа;

– виконання коду, яким представлений ліцензійний ключ. У разі, якщо код не коректний, відбудеться аварійне завершення програми.

В рамках розробки програмної частини були висунуті наступні вимоги до кінцевого продукту - програмного комплексу формування цифрового ідентифікатора (ліцензійного ключа) для захисту авторських прав:

– відсутність модифікації реєстру. На поточній стадії розвитку злочинних технологій, знайти програмний продукт для моніторингу реєстру не представляється проблематичним. Це може спростити зловмиснику процес дослідження алгоритму декодування ліцензійного ключа;

– мінімізація кількості змін в файловій системі (створення / зміна / видалення файлів);

– мінімізація кількості повідомлень обміну з сервером;

– використання захищеного каналу зв’язку для обміну повідомленнями з сервером (протокол https);

– захищеність програмного комплексу від злочинного впливу. При поточній реалізації використовується підпис виконуваного файлу і обфускація коду для ускладнення аналізу алгоритму роботи і запобігання його модифікації;

– поліморфність алгоритму кодування / декодування, в тому числі порядок компонентів системи повинен мати можливість варіювання;

– кросплатформність.

З висунутих вимог виникають недоліки поточної реалізації модулів програмного комплексу, які будуть враховані і виправлені при подальшій розробці:

– ліцензійний ключ в закодованому вигляді зберігається у файловій системі кінцевого користувача. Це дає можливість зловмиснику отримати доступ до закодованого ключа і проводити його модифікацію з метою зламу;

– механізм поліморфності алгоритму кодування і декодування не реалізований;

– для збору інформації про характеристики комп'ютерної системи кінцевого користувача використовується служба Windows WMI, яка специфічна для ОС Windows.

5.5 Розробка клієнт-серверної архітектури для демонстрації розробленого методу

В ході розроблення програмного комплексу генерації ліцензійного ключа захисту ПЗ, з метою захисту програмного комплексу від тиражування, була розроблена клієнт-серверна архітектура, що дозволяє продемонструвати роботу розробленого методу.

Клієнтське ПЗ, яке має захист від копіювання, що використовує розроблений комплекс, містить:

– код основного програмного продукту;

– client-processor.jar – розроблена бібліотека, що підтверджує легальність ліцензійного ключа і реалізує обмін повідомленнями з сервером. Дана бібліотека обфускована, що зменшує ймовірність аналізу алгоритму зловмисниками. Бібліотека складається з:

1) сервісів доступу до бази даних на стороні клієнта;

2) client-systeminfo.jar - модуля, що відповідає за отримання інформації про комплектуючі клієнтської комп'ютерної системи;

3) client-decoder.jar - модуля, декодуючого переданий ліцензійний ключ. Виявляє в ліцензійному ключі закодований програмний код, запускає його;

4) common-license.jar - модуля, що відповідає за генерацію цифрового підпису до повідомлення, а також перевірку підпису повідомлення на основі тексту повідомлення, геш-суми і існуючого публічного ключа. Його дублікат знаходиться також на сервері;

5) common-api.jar - модуля, що представляє собою інтерфейс доступу до сервісів сервера. Містить інтерфейси функцій реалізованих можливостей для використання шляхом обміну REST-запитами. Його дублікат знаходиться також на сервері.

Серверна частина складається з сервісів доступу до бази даних сервера:

– server-encoder.jar – модуля, який на основі вхідного java-файлу, що містить код, який виконується на стороні клієнта, а також додатково тільних налаштувань - інформації про конкретний програмний продукт, створює class-файл (скомпільований java-файл), який надалі кодується розробленим алгоритмом;

– common-license.jar – модуля, що відповідає за генерацію цифрового підпису до повідомлення, а також перевірку підпису повідомлення на основі тексту повідомлення, геш-суми та існуючого публічного ключа. Його дублікат знаходиться також на сервері;

– common-api.jar – модуля, що представляє собою інтерфейс доступу до сервісів сервера. Містить інтерфейси функцій реалізованих можливостей для використання шляхом обміну REST-запитами. Його дублікат знаходиться також на сервері.

Всі описані нижче діаграми послідовностей складаються з чотирьох структур:

1. База даних клієнта. У зв’язку з тим, що передбачається зберігання даних в форматі «ключ-значення» і відсутністю пов’язаних об’єктів, була обрана база даних MapDB [145], призначена спеціально для зберігання пар «ключ-значення». Дана база даних дозволяє ввести систему автентифікації для захисту від несанкціонованого доступу. При цьому, в

якості пароля для бази даних використовується геш-сума рядка, що містить інформацію про комплектуючі даної комп'ютерної системи, яка описана роботі [81], і є практично унікальною для кожної комп'ютерної системи. Це зменшує ймовірність використання зловмисником бази даних іншого користувача з зареєстрованим програмним продуктом і дозволяє уникнути «тиражування» ліцензії на даний програмний продукт.

2. Клієнтське ПЗ. Складається з програмного забезпечення, яке має ліцензійний ключ, і надбудови сервісів програмного забезпечення по обробці ліцензій, призначених для реєстрації користувача, реєстрації програмного продукту, верифікації ліцензійного ключа. При запуску даного ПЗ:

– відбувається звернення до клієнтської бази даних з використанням пароля, який обчислюється «на льоту» (тобто пароль від бази даних ніде не зберігається. У разі, якщо змінилася конфігурація системи, то пароль вже підходити не буде);

– з бази даних береться закодований ліцензійний ключ для даного програмного продукту і відбувається його верифікація, зокрема виконання закодованого в ньому коду. Код представлений у вигляді блоку програми на мові програмування Java, що дає як можливість використання даного підходу ліцензування, так і можливість перехоплювання виконання неприпустимих команд, що призводять до аварійного завершення програми або доступу до «чужої» пам'яті.

3. Сервер. Знаходиться в центрі сертифікації компанії, що поставляє це програмне забезпечення. Представлений у вигляді REST API компонентів, які працюють під управлінням сервера застосунків Jboss Wildfly 8.2.0. При поточній реалізації системи генерації ліцензійних ключів використовується Java 1.7. На сервері застосунків налаштований SSL-доступ для підвищення захисту механізму обміну повідомленнями між сервером і клієнтом від злочинного впливу. Слід відзначити той факт, що версія Java і Wildfly обрані з урахуванням їх поширеності на поточний момент, а також стабільним

механізмом роботи з X.509 сертифікатами (що було сильно змінено в Java 1.8); версія Wildfly підбиралася під порівнянність з версією Java Virtual Machine.

4. База даних сервера. Згідно до бізнес-вимог було обрано базу даних MongoDB [146], що має більш високі показники продуктивності читання даних у порівнянні з іншими базами даних. Містить інформацію про всіх зареєстрованих клієнтів, придбаних ними ліцензіях, а також інформацію про кожну зареєстровану користувальницьку комп'ютерну систему для можливості відновлення пароля або організації можливості поновлення пароля / ліцензійного ключа, якщо кінцевий користувач узгоджено змінює комплектуючі системи.

Сервер надає наступні можливості:

– реєстрація нового користувача;

– авторизація користувача при втраті / пошкодженні клієнтської бази даних;

– реєстрація нової комп'ютерної системи.

5.5.1 Модуль реєстрації нових користувачів

Будь-яка покупка програмного забезпечення починається з того, що користувач вносить свою клієнтську інформацію в базу даних сервера, який володіє ліцензією на дане програмне забезпечення. Діаграма послідовностей даного процесу представлена на рис. 5.4. За допомогою надбудов сервісів програмного забезпечення по обробці ліцензій, користувач передає свої клієнтські дані, які збережуться на сервері і будуть відображати його в клієнтській базі.

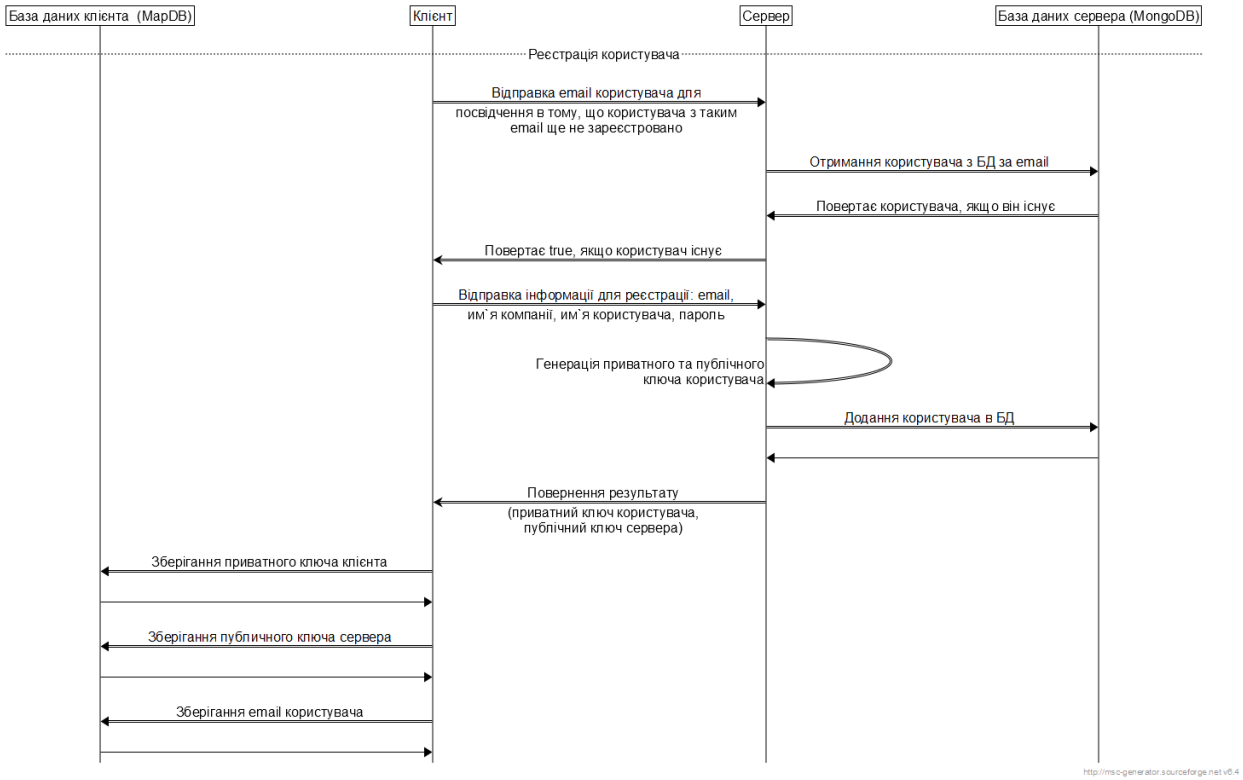


Рис. 5.4. Діаграма послідовностей процесу реєстрації користувача в системі

До таких даних відносяться:

– E-mail користувача;

– ім’я компанії, яку представляє даний користувач. У разі, якщо користувач - фізична особа, це поле може залишатися порожнім;

– ПІБ користувача, на ім’я якого відбувається реєстрація;

– пароль користувача. При цьому, e-mail і пароль користувача можуть виступати в процесі автентифікації користувача, в зв’язку з цим, e-mail повинен бути унікальним в системі, а пароль повинен бути безпечним згідно до політики паролів [193].

Після того, як система-сервер отримала інформацію про користувача, активізуються такі дії:

– перевіряється наявність даного користувача в своїй базі даних. У разі, якщо користувач з таким ідентифікатором-email вже існує, повертається відповідь з відповідною помилкою;

– генерується пара ключів, з урахуванням власного центру сертифікації, кінцевого користувача для можливості подальшого безпечного обміну повідомленнями з ним;

– зберігається в базі даних передана інформація про користувача, а також пара ключів. При цьому, з метою захисту даних, пароль користувача не зберігається у відкритому вигляді, а зберігається лише його геш-сума, створена за допомогою утиліти BCrypt [109], заснована на шифрі Blowfish;

– як відповідь, сервер повертає клієнту публічний ключ сервера і приватний ключ клієнта для можливості безпечного обміну повідомленнями. Дана відповідь не підлягає додатковим засобам захисту від зловмисника.

Отримавши успішну відповідь від сервера, клієнт зберігає отримані ключі, а також свій e-mail в клієнтській базі даних. Користувач зареєстрований в системі, і має можливість реєструвати конкретну комп'ютерну систему для даного програмного забезпечення.

5.5.2 Модуль автентифікації

Проведені дослідження показали, що дуже часто, в разі впливу злочинного програмного забезпечення або перевстановлення операційної системи, виникають ситуації втрати або пошкодження бази даних клієнтського програмного забезпечення. Нехтування цим істотно знижує практичну цінність розробки. Тому для врахування даного чинника був розроблений механізм відновлення зазначеної інформації при наявності збереженого e-mail і пароля зареєстрованого користувача. Механізм автентифікації користувача з відновленням його даних наведено на рис. 5.5.

Процес автентифікації відбувається наступним чином:

1. Користувач відправляє на сервер свої e-mail та пароль. З метою безпеки пароль відправляється у вигляді геш-суми, побудованої утилітою BCrypt. Сервер звіряє призначені для користувача e-mail і пароль. У разі помилки авторизації - повертається відповідна відповідь з помилкою. Якщо

призначені для користувача дані коректні, то з бази витягується вже наявна інформація про кінцевого користувача - його приватний ключ, і повертається відповіддю з сервера разом з публічним ключем сервера.

2. Отримавши успішну відповідь від сервера, клієнт зберігає отримані ключі, а також свій e-mail в клієнтській базі даних.

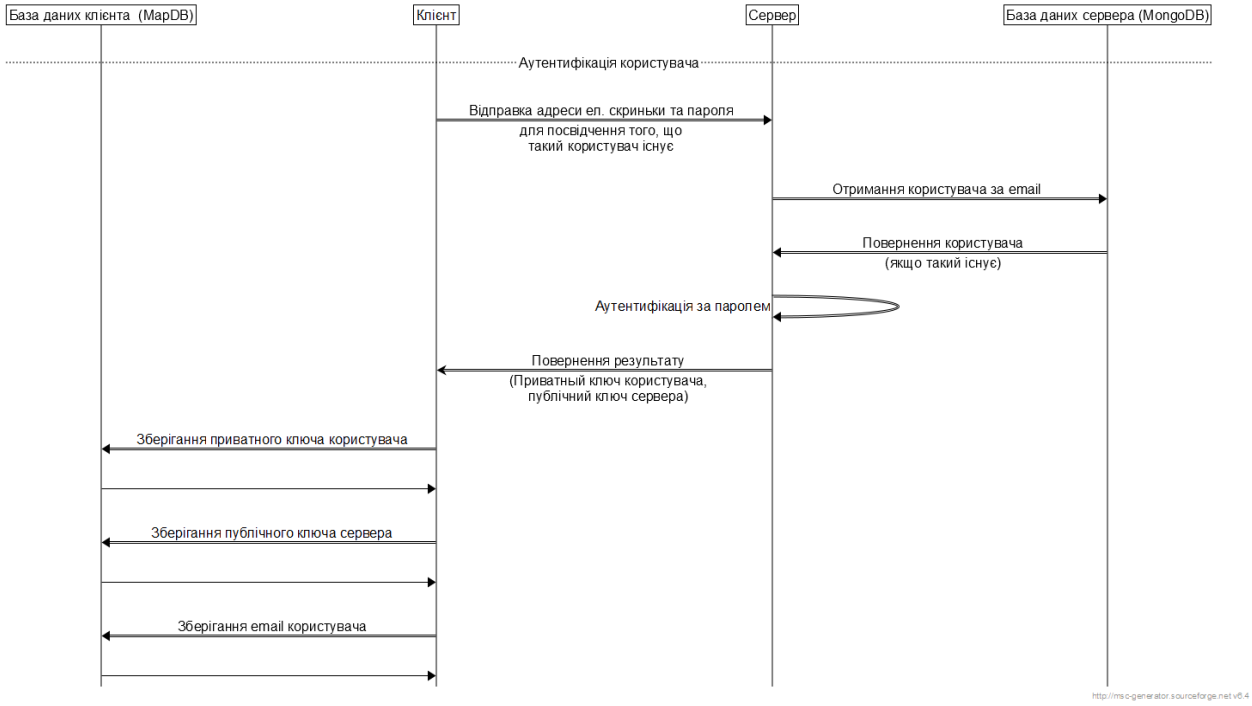


Рис. 5.5. Діаграма послідовностей процесу автентифікації користувача в системі

5.5.3 Модуль реєстрації комп'ютерної системи

Після того, як користувач виконав автентифікацію на сервері, він має можливість додавати / отримувати ліцензії на конкретну комп'ютерну систему. Процес додавання клієнтської комп'ютерної системи описаний на рис. 5.6, і складається з описаних нижче етапів.

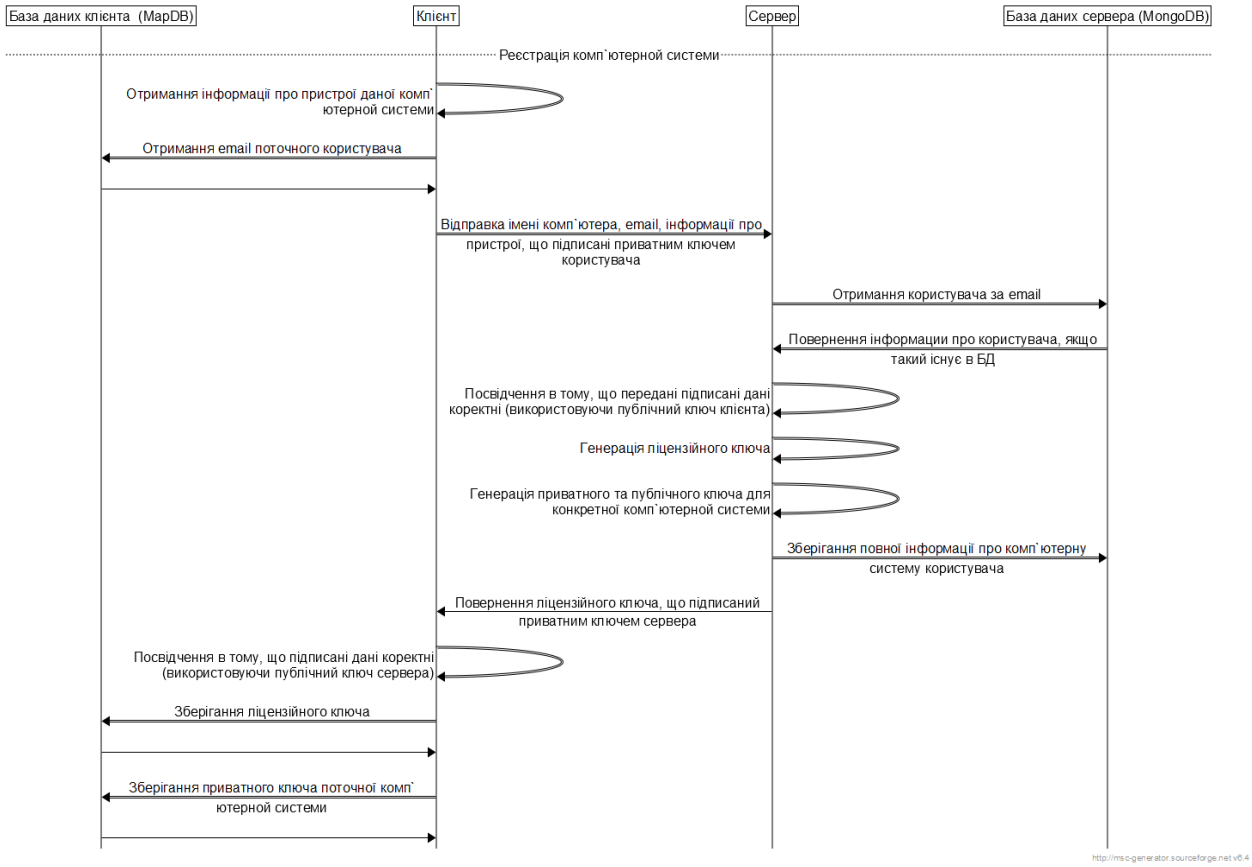


Рис. 5.6. Діаграма процесу реєстрації комп’ютерної системи

1. Ліцензійний модуль програмного забезпечення отримує інформацію про компоненти комп’ютерної системи.

2. На основі інформації, що зберігається в базі даних про користувача, відбувається відсилання запиту на сервер, що містить наступну інформацію: ім’я комп’ютерної системи (має бути унікальним для користувача, наприклад, «Мій ПК 1»), e-mail користувача, згенерована інформація про пристрій. Вся ця інформація підписується приватним ключем клієнта.

3. У зв’язку з тим, що будь-яка ліцензія вимагає оплати, користувач відправляється на сторінку оплати ліцензії через платіжну систему, наприклад, на paypal.com. Подальша реєстрація клієнтської комп’ютерної системи здійснюється лише при підтвердженні транзакції від платіжної системи.

4. Сервер отримує інформацію про зареєстрованого користувача на основі отриманих послідовностей клієнтської КС на сервері e-mail, перевіряє достовірність надісланого повідомлення на основі існуючого публічного ключа користувача.

5. Сервер генерує ліцензійний ключ, кодує його.

6. Копія ліцензійного ключа зберігається на сервері в закодованому вигляді з метою можливості його відновлення.

7. Сервер відправляє згенерований ліцензійний ключ. Повідомлення підписується приватним ключем сервера.

8. Ліцензійний модуль клієнтського програмного забезпечення перевіряє достовірність надісланого повідомлення на основі існуючого публічного ключа сервера і зберігає ліцензійний ключ в закодованому вигляді в базі даних, яка перебуває на стороні клієнта.

9. Ліцензійний ключ декодується і запускається, що, в разі успішної роботи, призводить до того, що програмне забезпечення буде зареєстрованим. Даний процес виконується кожен раз при запуску програмного забезпечення.

5.6 Дослідження розробленого методу формування цифрового ідентифікатора

В рамках досліджень було проведено експеримент.

Мета експерименту - дослідження розробленого методу на предмет складності і тривалості зламу ліцензійного ключа.

Аудиторія - студенти 4-5 курсів НТУ «ХПІ», а також IT-фахівці фірми NixSolutions. Загальна кількість учасників - 500 осіб.

Обмеження експерименту:

- аудиторія приділяла експерименту 4 години на день в робочі дні;
- тривалість експерименту – 5 тижнів.

Об’єкти дослідження. Для даного експерименту обрано програмні продукти різної цінової категорії. Відмінною особливістю обраних зовнішніх програмних продуктів є той факт, що вони комерційні (з закритим вихідним кодом); написані на мовах програмування, що використовують віртуальні машини (JVM, CLI); використовуються в IT-індустрії:

– програмний продукт, який використовує розроблену модель безпеки з використанням ліцензій:

– що використовує модуль ідентифікації і контролю цілісності;

– що використовує модуль ідентифікації і контролю цілісності, з використанням коду, що вбудовує функції в ліцензійний ключ;

– що використовує модуль ідентифікації і контролю цілісності, з використанням SSL-потоку передачі даних при клієнт-серверній взаємодії;

– DBeaver. Відмінною особливістю даного програмного продукту є використання методу клієнт-серверної взаємодії при верифікації ліцензійного ключа;

– SonarCube. Відмінною особливістю даного програмного продукту є використання методів обфускації коду і методу підпису / верифікації ліцензійного ключа за допомогою системи публічного / приватного ключів;

– HTTP Commander 3. Відмінною особливістю даного програмного продукту є відсутність прив’язки до персональних даних користувача (напр., ПІБ, організація, кінець періоду дії), використання статичних ліцензійних ключів в залежності від наданої функціональності;

– Ice Scrum. Відмінною особливістю даного програмного продукту є відсутність прив’язки до персональних даних користувача (напр., ПІБ, організація), використання алгоритмів гешування (MD2, MD5, SHA1) і методів заміщення бітів.

Результати експерименту наведені в табл. 5.1. для розробленої моделі.

Таблиця 5.1. Результати експерименту часу зламу програмного забезпечення з використанням розроблених методів безпеки на основі ліцензійних ключів

Номер учасника	Використання модуля ідентифікації і контролю цілісності, год.	Використання модуля ідентифікації і контролю цілісності, коду вбудовується функції в ліцензійний ключ, год.	Використання модуля ідентифікації і контролю цілісності, SSL потоку передачі даних при клієнт-серверному взаємодії, год.
1	66.8	33.9	27.8
2	45.9	49.4	30.7
3	61.3	50.6	33.0
4	76.5	45.4	36.8
5	60.2	53.4	35.9
6	44.6	49.8	21.9
7	75.6	49.1	38.7
8	78.0	38.7	32.5
9	63.2	33.0	21.3
10	71.0	41.3	31.8
11	52.1	31.8	26.9
12	83.6	52.5	22.9
13	84.4	48.4	40.4
14	69.7	34.2	35.1
15	48.5	56.3	39.9
16	77.5	32.2	35.2
17	81.3	47.3	31.6
18	52.6	41.5	23.7
19	74.6	48.0	21.5
20	54.8	41.1	26.5
…	…	…	…
500	69.6	57.8	26.8
Середній час	31.7	63.5	47.3

З таблиці видно, що найкращий результат показала розроблена модель, яка використовує модуль ідентифікації і контролю цілісності, з використанням коду, що вбудовує функції в ліцензійний ключ. Середній час зламу програмного забезпечення, що використовує дану модель безпеки, склав 63.5 години.

У табл. 5.2 наведено порівняльний аналіз середнього часу зламу програмного забезпечення. Результати експерименту показали підвищення складності зламу ліцензійного ключа розробленого методу в 1.29 раз.

Таблиця 5.2. Порівняльний аналіз середнього часу зламу програмного забезпечення

	Середній час, год.
Програмний продукт з використанням розробленого методу	63.5
Http Commander 3	5.6
IceScrum	16.7
SonarCube	49.2
DBeaver	25.1

Висновки за розділом 5

1. Розроблено модулі програмного комплексу, які демонструють можливість створення цифрового ідентифікатора - ліцензійного ключа з урахуванням індивідуальних даних кінцевого користувача, що запобігає можливості тиражування ліцензійного ключа зловмисниками. Сформований ліцензійний ключ являє собою програмний код, що виконується на стороні кінцевого користувача, що дає додатковий захист програмного продукту від злочинного впливу. Посилення захисту програмного продукту від злочинного впливу шляхом використання існуючих механізмів захисту: цифровий підпис виконуваного файлу, обфускація.

2. Розроблено узагальнену структуру процесу формування цифрового ідентифікатора ПЗ. Відмінною особливістю запропонованої структури є використання формальних даних про комп’ютерні системи, на які ліцензійне ПЗ встановлюється в процесі формування ліцензійного цифрового ідентифікатора. Запропоновано алгоритм функціонування системи і генерації ліцензійного ключа, адаптованого до вхідних даних і можливих умов верифікації ПЗ.

3. Розроблено структурно-функціональну модель формування цифрового ідентифікатора програмного забезпечення для захисту її авторських прав. Відмінною особливістю даної моделі є використанням індивідуальних даних комп’ютерної системи кінцевого користувача для однозначної ідентифікації приналежності. Також, модель враховує

можливість вбудовування довільного (заданого розробниками) коду в тіло ліцензійного ключа, який буде виконуватися при верифікації. Дані маніпуляції призвели до ускладнення аналізу і зламу ліцензійної складової програмного забезпечення. Це дозволило підвищити середній час зламу в 1.29 разів.

РОЗДІЛ 6. ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ РОЗРОБЛЕНИХ МОДЕЛЕЙ ТА МЕТОДІВ ПІДВИЩЕННЯ БЕЗПЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ОБҐРУНТУВАННЯ ПРАКТИЧНИХ РЕКОМЕНДАЦІЙ ЩОДО ЇХ ВИКОРИСТАННЯ

6.1 Розробка імітаційної моделі системи підвищення безпеки програмного забезпечення в умовах використання байт-код орієнтованих мов програмування

Одним з найважливіших етапів розробки методу підвищення безпеки програмного забезпечення в умовах використання байт-код орієнтованих мов програмування є імітаційне моделювання та експериментальні дослідження [3, 35, 83, 84]. При цьому, метою цих досліджень є:

– перевірка адекватності розроблених моделей та методів підвищення безпеки програмного забезпечення;

– аналіз достовірності отриманих результатів у ході поставлених оптимізаційних задач;

– обґрунтований вибір показників та оцінка ефективності розробленого методу завдяки ним;

– формування науково-практичних рекомендацій щодо використання розроблених методів обфускації коду програмного забезпечення та методів підвищення безпеки програмного забезпечення на основі ліцензійних ідентифікаторів в умовах використання байт-код орієнтованих мов програмування.

Для обґрунтування достовірності отриманих результатів та оцінки ефективності розроблених методів, було проведено імітаційне моделювання підсистем:

– захисту на основі обфускації програмного коду;

– захисту на основі формування ліцензійних ідентифікаторів;

– аналізу вимог до ймовірісно-часових показників програмного забезпечення.

У якості інструментарію для імітаційного моделювання використовувався пакет Mapple-2021 [3].

Узагальнена структурна схема розробленої імітаційної моделі системи підвищення безпеки програмного забезпечення наведена на рис. 6.1.

Як видно з рисунка, до складу програмного комплексу імітаційного моделювання входять дві системи, які виконують окремі функції:

– вироблення показників, обмежень і критеріїв оптимізації процесів обфускації коду та генерації ліцензійних ключів, а також вироблення відповідних керуючих команд;

– вироблення рішень про захист програмного забезпечення в процесі його експлуатації.

У процесі формування вимог до безпеки програмного забезпечення емулювалися та розглядалися різні показники і характеристики інформаційного обміну, типи системного програмного забезпечення, можливі варіанти архітектурної побудови програмного забезпечення. Зокрема розглядалися такі ситуації:

– кількість і типи використовуваних операційних систем (в даному випадку використовувалися наступні операційні системи: Windows 8, 10; Linux Ubuntu 16; MacOS X 11);

– кількість і типи використовуваних байт-код орієнтованих мов програмування (в даному випадку використовувалися наступні мови програмування: .Net 6, .Net Core 3, JVM 8, 11);

– зв’язність мережі, кількість функціональних вузлів і зв’язків між ними. Даний параметр задається в вигляді масиву вузлів, кожен з яких має список вузлів-«сусідів». Вузол представлений у вигляді набору наступних параметрів: назва вузла, тип ОС вузла, список вузлів-«сусідів». Даний

показник впливає на модуль формування ліцензійного ключа при взаємодії роботи клієнтського ПЗ та ліцензійного модулю сервера.

З урахуванням вищевказаних факторів, в програмному комплексі здійснюється вибір показників, обмежень і критеріїв оптимізації процесів забезпечення безпеки програмного забезпечення. При цьому формуються команди для підсистеми захисту на основі систем обфускації програмного коду та формування ліцензійного ідентифікатору.

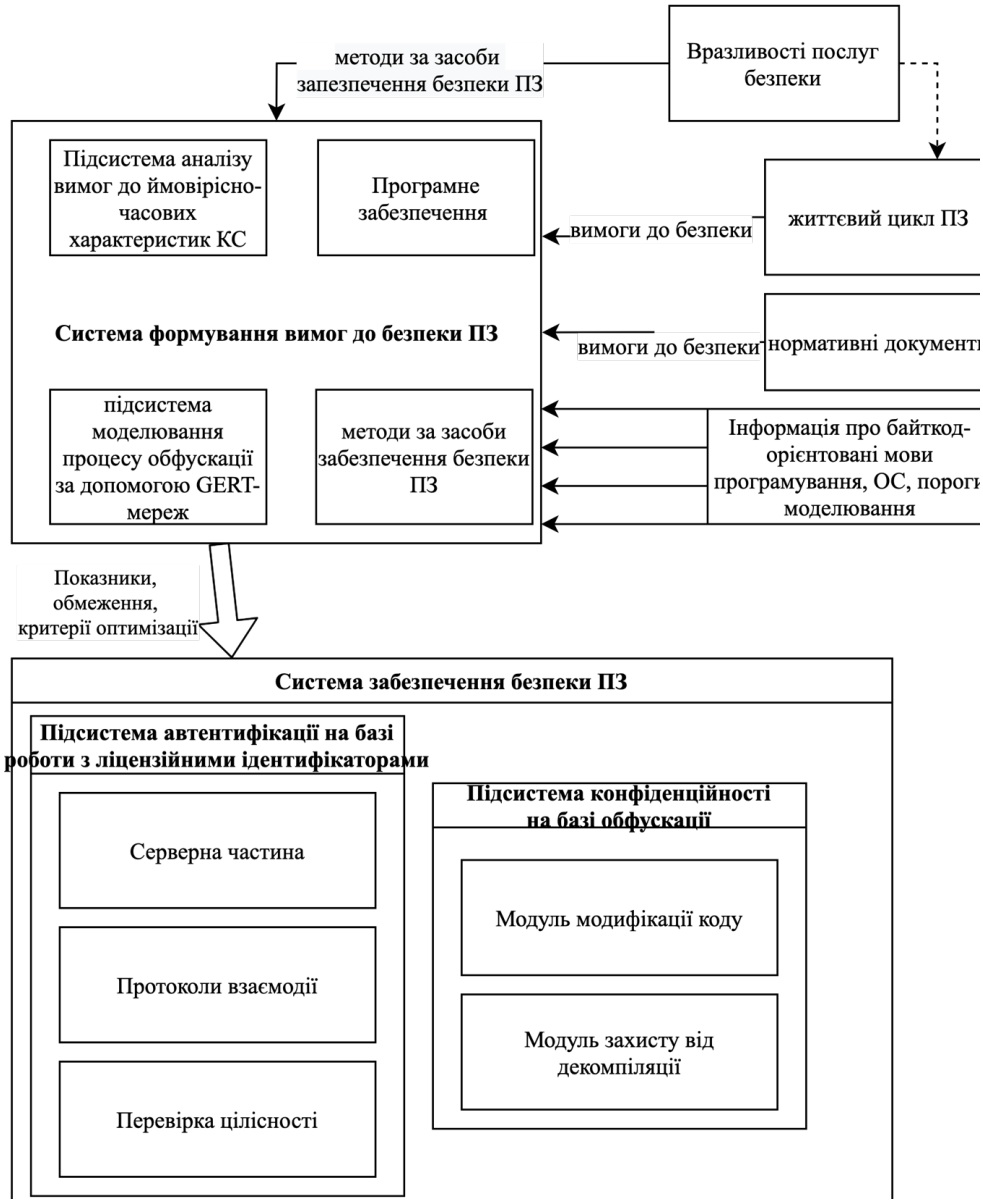


Рис. 6.1. Структурна схема імітаційної моделі систем формування вимог до безпеки та захисту програмного забезпечення

На рис. 6.2 наведено діаграму класів розробленого програмного комплексу формування ліцензійного ключа. Як видно з рисунку, програмний модуль формування ліцензійного ключа має об’єкти, що відповідають за інформацію про кінцевого користувача (UserBean) та його комп’ютерну систему (UserCompSystemBean), у тому числі інформацію про конфігурацію системи (DeviceInfo), дані якої є вхідними для формування ліцензійного ключа.

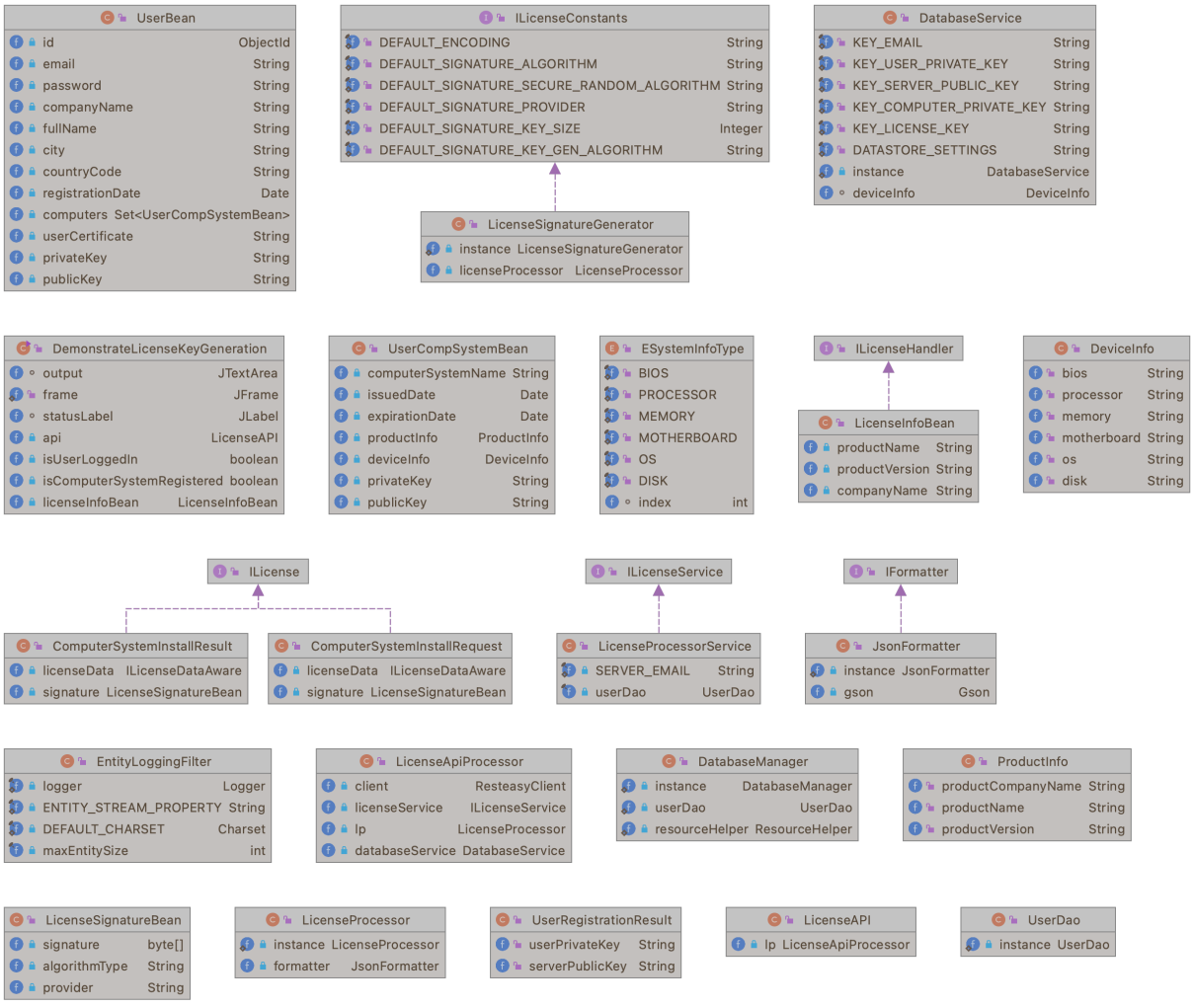


Рис. 6.2. Діаграма класів розробленого програмного забезпечення формування ліцензійного ключа

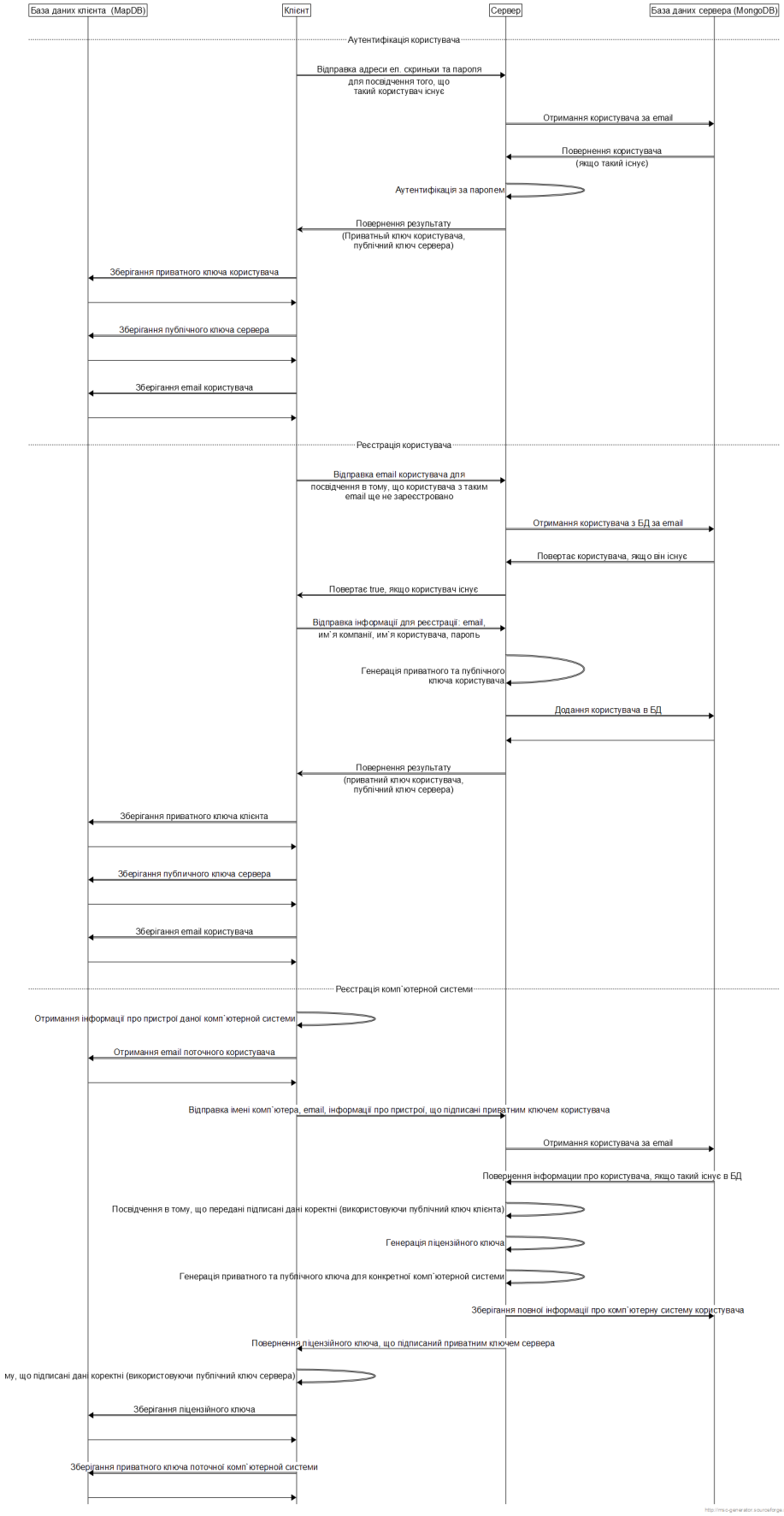


Рис. 6.3. Діаграма послідовностей алгоритму формування ліцензійного ключа

На рис. 6.3 зображено повну діаграму послідовності основних процесів при формуванні ліцензійного ключа. Зокрема, наведено процеси автентифікації та реєстрації.

Проведемо порівняльні дослідження з відомими методами, використовуючи методи та прийоми математичного та імітаційного моделювання.

Для підтвердження ефективності розробленого методу в порівнянні з існуючими, знайдемо співвідношення часу зламу ліцензійного ключа: t_{MI} / t_{Mi}.

6.2 Оцінка ефективності розроблених методів

У рамках дослідження час зламу ліцензійного ключа, що був створений завдяки розробленому методу підвищення безпеки програмного забезпечення, в умовах використання байт-код орієнтованих мов програмування (МПБПЗ) був порівняний з існуючими методами:

– метод клієнт-серверної взаємодії при верифікації ліцензійного ключа (КСВ);

– метод обфускації коду і методу підпису / верифікації ліцензійного ключа за допомогою системи публічного / приватного ключів (ОППК) [224];

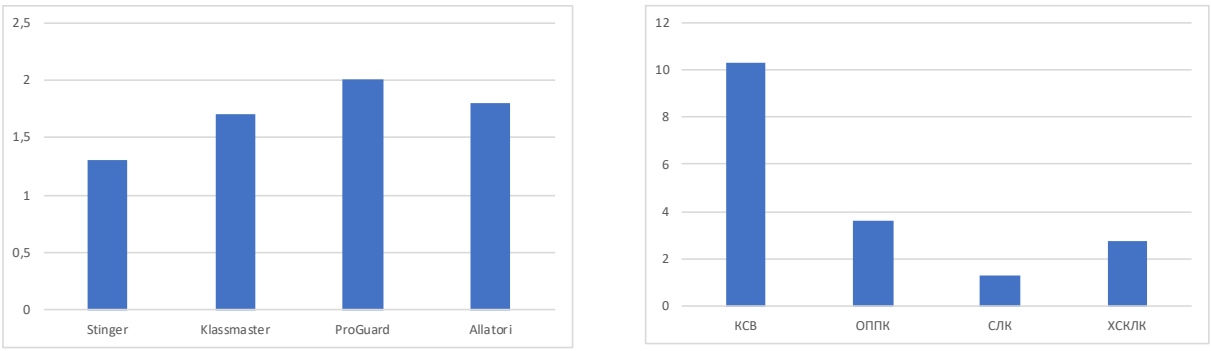
– метод статичних ліцензійних ключів в залежності від наданого функціоналу (СЛК);

– метод використання алгоритмів гешування (MD2, MD5, SHA1) [158] і симетричного кодування ліцензійного ключа (ХСКЛК).

Дані експерименту наведені у табл. 6.1 та рис. 6.4, 6.5. Як бачимо з рис. 6.5, мінімальна ефективність розробленого методу складає 1.4 рази (у порівнянні з існуючим методом обфускації коду і методу підпису / верифікації ліцензійного ключа за допомогою системи публічного / приватного ключів).

Таблиця 6.1. Результати експерименту часу зламу програмного забезпечення з використанням розроблених методів безпеки на основі ліцензійних ключів

Номер учасника	МПБПЗ, год.	КСВ, год	ОППК, год.	СЛК, год.	ХСКЛК, год.
1	53.03	5.19	13.90	33.9	27.8
2	56.24	5.50	16.75	49.4	30.7
3	86.99	5.20	11.41	50.6	33.0
4	81.36	5.87	20.38	45.4	36.8
5	56.52	5.99	16.08	53.4	35.9
6	88.73	6.17	13.29	49.8	21.9
7	97.79	7.29	19.38	49.1	38.7
8	56.65	4.29	11.16	38.7	32.5
9	84.11	5.12	18.89	33.0	21.3
10	59.24	6.09	11.33	41.3	31.8
11	74.27	7.42	11.49	31.8	26.9
12	89.25	7.09	21.25	52.5	22.9
13	77.60	5.98	17.37	48.4	40.4
14	92.11	5.39	16.41	34.2	35.1
15	93.01	5.37	12.94	56.3	39.9
16	104.19	5.55	20.23	32.2	35.2
17	83.24	6.33	14.67	47.3	31.6
18	53.26	6.66	14.50	41.5	23.7
19	71.50	3.91	14.92	48.0	21.5
20	82.65	4.43	15.57	41.1	26.5
…	…	…	…	…	…
500	52.54	4.69	11.43	32.1	31.8
Середній час	**69.5**	**5.6**	**16.7**	**47.3**	**25.1**



(а) (б)

Рис. 6.4. Графік співвідношення середнього часу зламу програмного забезпечення, що використовує розроблений метод обфускації (а) та генерації ліцензійного ключа (б) до програмних застосунків, що використовують існуючі методи

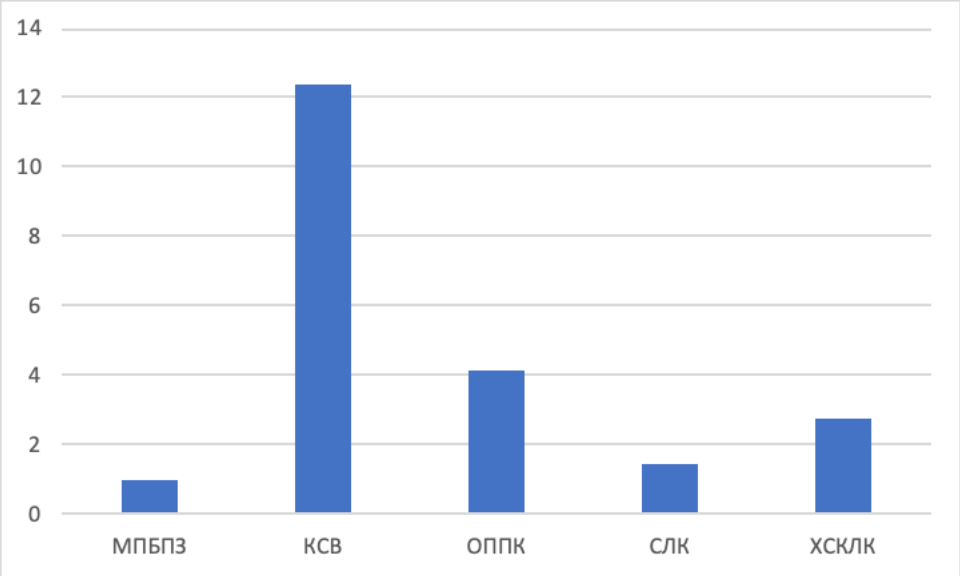


Рис. 6.5. Графік співвідношення середнього часу зламу програмного забезпечення розроблених методів до програмних забезпечень, що використовують існуючі методи

6.3. Обґрунтування достовірності одержаних результатів наукових досліджень

Для обґрунтування достовірності отриманих у попередніх розділах результатів проведено імітаційне моделювання процесу підвищення безпеки програмного забезпечення шляхом поліпшення методу обфускації програмних модулів і генерації ліцензійного ключа.

Висунута в дисертаційній роботі гіпотеза про нормальний розподіл випадкової величини часу зламу ліцензійного ключа була перевірена за критерієм згоди χ^2 Пірсона [9, 17]:

$$\chi^2 = N^* \sum_{i=1}^{k} (P_i^* - P_i)^2 / P_i ,$$

де k – кількість розрядів (інтервалів) статистичного ряду,

P_i^* і P_i – «статистична» і теоретична ймовірність «потрапляння» величини середнього часу зламу в і-й розряд.

Проведена перевірка довела правдоподібність гіпотези про те, що величина часу зламу розподілена за нормальним законом.

Отримані оцінки $w(\vec{\xi})^{(i)}$ математичного сподівання і $\widehat{D}_{w(\vec{\xi})^{(i)}}$ дисперсії ($\widehat{\sigma}_{w(\vec{\xi})^{(i)}}$ середнього відхилення) випадкової величини $w(\vec{\xi})^{(i)}$ [9, 17]:

$$\widehat{w}(\vec{\xi})^{(i)} = \frac{\sum_{i=1}^{k} \widehat{w}(\vec{\xi})^{(i)}}{\mathrm{N}^*}; \quad D_{w(\vec{\xi})^{(i)}} = \frac{\sum_{i=1}^{k} \left(w(\vec{\xi})^{(i)} - \widehat{w}(\vec{\xi})^{(i)} \right)^2}{N^* \text{-} 1}; \quad \widehat{\sigma}_{w(\vec{\xi})^{(i)}} = \sqrt{\widehat{D}_{w(\vec{\xi})^{(i)}}}.$$

Скориставшись виразом для розрахунку довірчої ймовірності відхилення відносної частоти від постійної ймовірності в незалежних випробуваннях, отримане в результаті експерименту значення прогнозованого рівня квантування «не відхилене» від математичного сподівання $\widehat{w}(\vec{\xi})^{(i)}$ більш ніж на 1:

$$P\left(\left| \widehat{w}(\vec{\xi})^{(i)} - w(\vec{\xi})^{(i)} \right| < 1 \right) = 2\Phi(1 \Big/ \widehat{w}(\vec{\xi})^{(i)}),$$

де Φ – функція Лапласа виду $\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x e^{-t^2/2} dt$ [9, 17].

Поведінка імітаційного моделювання показала, що для всіх досліджуваних видів даних довірча ймовірність того, що значення статистичної величини $w(\vec{\xi})$ «не відхилиться» від математичного сподівання $w(\vec{\xi})$ більш ніж на 1: $P \approx 0{,}98$.

Високий ступінь збігу результатів імітаційного та математичного моделювання підтверджує достовірність розробленого методу підвищення безпеки програмного забезпечення в умовах використання байт-код орієнтованих мов програмування.

6.4. Рекомендації щодо практичного застосування розроблених методів

Проведені дослідження [140, 191, 203] показали, що на стадіях процесу проектування архітектури програмних засобів та процесу детального проектування, а також, особливо, впровадження розроблених методів та засобів підвищення безпеки програмних засобів, слід використовувати ряд рекомендацій, які допоможуть уникнути звичайних проблем у кожній з досліджених областей захисту даних:

- автентифікація;
- авторизація;
- управління конфігурацією;
- управління винятками;
- протоколювання і інструментування;
- управління станом.

Проектування та розробка ефективної стратегії автентифікації грає суттєву роль у забезпеченні безпеки програмного забезпечення. Недотримання цих стратегій призводить до можливостей зловмисних дій шляхом:

- підробки пакетів;
- атаки перебором за словником;
- перехопленням сеансів (sniffing) та ін.

При проектуванні стратегії автентифікації необхідно керуватися такими рекомендаціями [186, 189, 190]:

1. Визначте межі довіри і проводьте автентифікацію користувачів і викликів на межах довіри:

– не дозволяйте входити в систему з конфіденційними обліковими записами (тобто обліковими записами, які можуть бути використані

всередині рішення, наприклад, для внутрішнього / середнього / бази даних) до будь-якого інтерфейсу користувача інтерфейсу;

– не використовуйте одне і те ж рішення для автентифікації (наприклад, Identity Provider (Azure, Google, Okta) / Active Directory (LDAP, Windows AD)), яке використовується всередині для незахищеного доступу (наприклад, відкритий доступ / демілітаризована зона (DMZ));

2. *Забезпечте використання надійних паролів або парольних фраз:*

– програмне забезпечення повинно декларувати мінімальну довжину паролів. Паролі менше 8 символів вважаються слабкими відповідно до NIST SP800-63B;

– впровадьте перевірку слабких паролів, наприклад, тестування нових або змінених паролів у списку 10 000 найгірших паролів.

– дозвольте використання всіх символів, включаючи unicode та пробіли. Не повинно бути правил складання паролів, що обмежують тип дозволених символів;

– забезпечте ротацію облікових даних під час зламу пароля або під час компрометації ідентифікації;

– включіть вимірювач міцності пароля, щоб допомогти користувачам створити більш складний пароль і заблокувати загальні та раніше порушені паролі. Для цього існує бібліотека zxcvbn від компанії Dropbox, та онлайн-сервіс з відкритим API: https://haveibeenpwned.com;

– не передавайте паролі по мережі і не зберігайте їх у базі даних або сховище даних у відкритому вигляді. Зберігайте геш пароля. Для цього існують бібліотеки, що реалізують алгоритм bcrypt;

3. *Використовуйте багатофакторну автентифікацію (MFA).* Аналіз Microsoft припускає, що він зупинив би 99,9% компрометацій облікових записів. Це запобігає автоматизованим заповненням облікових даних, грубою силою та викраденими атаками повторного використання.

4. *Використовуйте протоколи автентифікації, що не потребують пароля. Наприклад, OAuth, OpenID, SAML, FIDO.*

5. *Застосовуйте авторизацію на базі ресурсів для аудиту системи.* При авторизації на базі ресурсів права доступу визначаються в самому ресурсі. Наприклад, список керування доступом (Access control list) ресурсу Windows використовується для визначення прав доступу викликаючої функції до ресурсу.

6. *Використовуйте авторизацію на підставі тверджень, якщо потрібно підтримувати інтегровану авторизацію на базі поєднання даних, таких як посвідчення, роль, дозволи, права*.

Правильний вибір механізму управління конфігурацією має велике значення з точки зору забезпечення безпеки і надійності програми. Хоча управління конфігурацією може бути не найпростішою або першою функцією, яку ви впроваджуєте в управлінні своїм IT-середовищем, це може бути дуже потужним кроком у забезпеченні належного контролю ваших IT-ресурсів. Управління конфігурацією дозволяє зрозуміти ваші IT-ресурси та їх взаємозв'язок. Ця інформація дозволяє приймати обґрунтовані рішення щодо підтримання та вдосконалення IT-середовища. Впровадження програми управління конфігурацією є важливим кроком для оптимізації управління вашим IT-світом. При проектуванні стратегії управління конфігурацією керуйтеся такими рекомендаціями [184,199]:

– залишайтеся під контролем. Ви повинні контролювати елементи конфігурації та способи їх зміни. Якщо ви втрачаєте контроль, ваш процес ні до чого;

– використовуйте технології. Керування конфігурацією вручну може виявитись неможливим. Вам потрібно використовувати технологію, яка допоможе виявити, записати та зберегти інформацію про конфігурацію;

– управління конфігурацією призводить до появи інших речей. Впровадження управління конфігурацією саме по собі є марною тратою часу.

Ваша програма управління конфігурацією повинна включати програми управління змінами, інцидентами, проблемами та випусками, якщо ви хочете побачити повернення.

У рамках деталізації описаних правил можна виділити наступні рекомендації:

– *плануйте та виділяйте область застосування*. Реалізація програми управління конфігурацією може бути важким завданням. Тому надзвичайно важливо встановити реалістичні терміни проекту та почати з основ. Якщо ви спробуєте зробити занадто багато, ви ніколи не отримаєте програму з місця. Початок розумним, обмеженим чином дозволяє отримати контроль у певних сферах та розвивати досягнутий успіх у подальшому.

– *створіть сховище*. У багатьох постачальників є дуже потужні бази даних керування конфігурацією (БДКК), які можуть відповідати вашим потребам. Незалежно від обраного вами маршруту, ви повинні мати сховище для точної інформації про конфігурацію. Одним із результатів планування та розробки сфери має бути вимога, встановлена для сховища. Використовуйте ці вимоги, щоб допомогти вам обрати правильні елементи технології зберігання. Ви можете виявити, що окремий продукт БДКК може не відповідати вашим потребам. Додавайте інші компоненти, щоб створити правильну технологічну суміш для вашої організації.

– *визначайте правильні CI.* Існують тисячі *CI*, які Ви могли б відстежити, але спроба їх відстеження буде марною тратою Вашого часу та ресурсів. Необхідно переглянути кожен рівень вашої IT-інфраструктури (застосунки, операційні системи, обладнання та послуги), щоб визначити найбільш підходящий набір *CI* та відносини для відстеження та моніторингу. Визначивши *CI*, які потрібно відстежувати, певні постачальники можуть допомогти зібрати початкову інформацію про *CI*.

– *делегуйте керування конфігурацією іншим IT службам*. Функції, такі як управління інцидентами, управління змінами, управління випуском,

управління проблемами та управління активами можуть використовувати ключову інформацію, що відстежується управлінням конфігурацією. Оскільки інші служби використовують управління конфігурацією, вони зможуть реагувати ефективніше або швидше впроваджувати зміни. Ця інтеграція дозволяє оптимізувати IT в загальному обсязі її надання послуги для бізнесу.

– *федеруйте неоднозначну інформацію про конфігурацію*. Швидше за все, ви не будете реалізовувати один єдиний БДКК або сховище, а матимете кілька сховищ для інформації про конфігурацію (одне для мережі, одне для MVS, одне для операційної системи Solaris, одне для зберігання, тощо). Програми управління конфігурацією слід об'єднувати в незалежні домени. Це дасть більш вичерпний погляд на поточне IT-середовище.

– *використовуйте автоматизацію, що орієнтована на політику*. Автоматичне виявлення фактів та управління конфігурацією може суттєво допомогти зберегти стабільність вашої програми управління конфігурацією. Автоматизація потенційно може піти далі, ніж просто відкриття; наскільки ви можете базувати політику та правила на інформації про конфігурацію, у центрі обробки даних може відбуватися безліч інших засобів автоматизації. Це починає створення самосвідомої, самонаправляючої технологічної платформи.

Проектування ефективної стратегії управління виключеннями має велике значення з точки зору забезпечення безпеки і надійності програми.

Стратегія обробки виключень складається з усіх дій та домовленостей, необхідних для належної обробки виключень у вашій програмі [154, 188].

На рис. 6.6 наведено огляд різних частин стратегії обробки виключень – від виявлення помилки аж до місця обробки помилки.

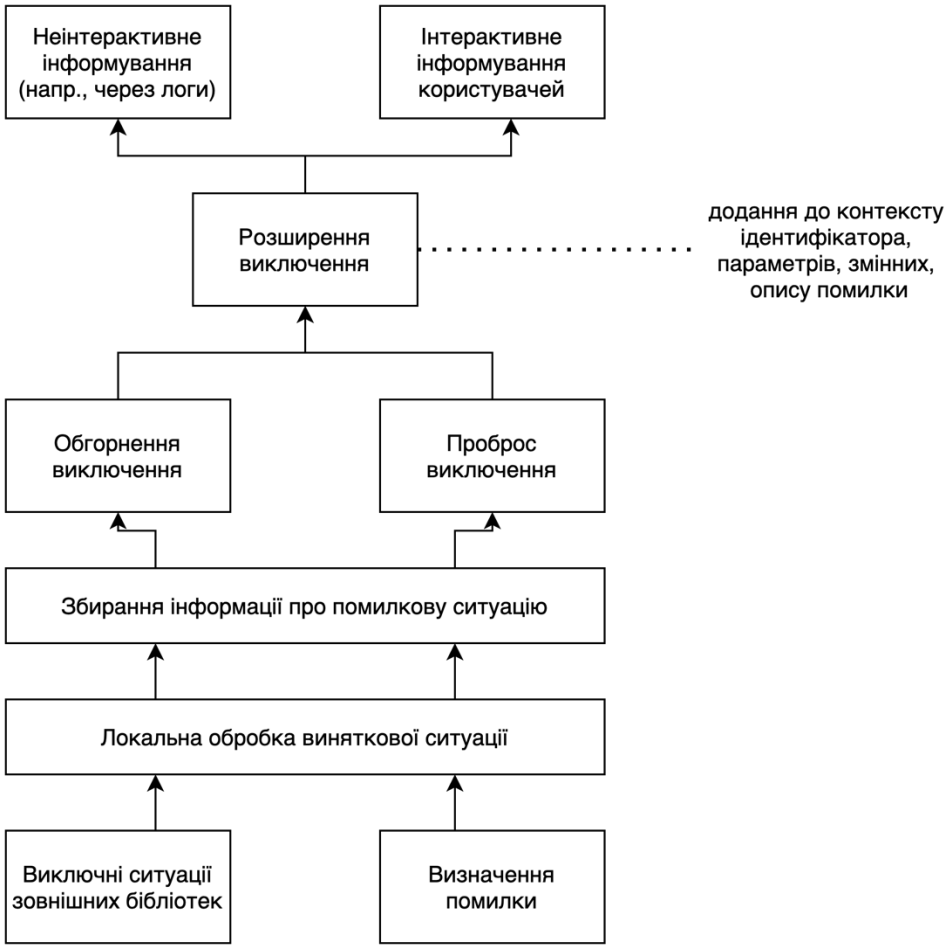


Рис. 6.6. Огляд стратегії обробки виключень

Внизу стратегії знаходиться виявлення помилок. Ви виявляєте помилку або ви самі, або ви виявляєте зовнішнє виключення.

На другому рівні ви пробуєте локальну обробку виключень, якщо це можливо. Тут ви, наприклад, надійшліть свій запит на сервер резервного копіювання або почекайте певну кількість секунд і спробуйте ще раз і т.д. Ви повинні робити все, що можете, щоб усунути помилку та продовжити будь-яку операцію, яку виконувала ваша програма.

На третьому рівні, коли локальна обробка помилок не працювала, ви збираєте всю інформацію, необхідну для діагностики, відтворення та повідомлення про помилку.

На четвертому рівні ви перетворюєте спіймане виключення або виявлену помилку у виключення для конкретної програми та кидаєте його.

Ви додаєте будь-яку інформацію, необхідну для належної обробки виключення.

На п’ятому рівні ви розповсюджуєте виключення, що стосується застосунку, до стеку викликів, закриваючи будь-які відкриті ресурси на шляху (наприклад, файли, мережеві з’єднання, звільнення виділених буферів тощо). Крім того, ви додаєте будь-яку контекстну інформацію, яка може бути корисною для визначення причини та серйозності помилки.

На шостому рівні ви ловите і реагуєте на виключення. На даний момент обробка локальних виключень вже неможлива, тому найчастіше єдине, що може зробити ваш застосунок – це повідомити про виключення користувачів та не користувачів. Якщо виключення дійсно дуже погане, ваша програма може навіть аварійно завершити роботу.

Неправильний вибір стратегії дуже ускладнить діагностування та вирішення проблем застосунку, зробить його вразливим для атак типу відмова в обслуговуванні (DoS), а також може призвести до розголошення конфіденційних і важливих відомостей. Формування та оброблення виключень є ресурсоємним процесом, тому важливо, щоб при проектуванні були також враховані питання продуктивності. Загальні рекомендації:

– не використовуйте виключення для управління логікою застосунку;

– перехоплюйте внутрішні виключення, тільки якщо можете обробити або повинні додати деякі дані;

– застосунок не повинен залишатися в нестабільному стані після збою;

– протоколюйте усі помилки. Проте, слід зазначити, що конфіденційні дані не повинні бути протокольовані.

Основою більшості пристроїв, що забезпечують захист IT-мереж, є можливість реєструвати події та вживати дії на основі цих подій. Ця програма та моніторинг системи надає подробиці як про те, що сталося з пристроєм, так і про те, що відбувається. Він забезпечує захист застосунків, попереджаючи вас про проблеми, тому захисні заходи можуть бути вжиті до

нанесення будь-якої реальної шкоди. Без моніторингу у вас мало шансів виявити, чи атакується програма, яка працює, чи вона порушена.

Критично важливі програми, процеси, що обробляють цінну або конфіденційну інформацію, раніше скомпрометовані або зловживані системи, а також системи, підключені до третіх сторін або Інтернету, вимагають активного моніторингу. Будь-яка серйозно підозріла поведінка або критичні події повинні створювати попередження, яке оцінюється та реагується на нього. Незважаючи на те, що вам потрібно буде провести оцінку ризику для кожної програми чи системи, щоб визначити, який рівень аудиту, перегляду журналу та моніторингу необхідний, вам доведеться реєструвати щонайменше наступне [115]:

– ідентифікатори користувачів;

– дата та час входу та виходу та інші ключові події;

– ідентифікація терміналу;

– успішні та невдалі спроби отримати доступ до систем, даних або застосунків;

– доступ до файлів та мереж;

– зміни в конфігураціях системи;

– використання системних утиліт;

– виключення та інші події, пов’язані з безпекою, такі як спрацьовані тривоги;

– активація систем захисту, таких як системи виявлення вторгнень та антивірусні програми.

Збір цих даних допоможе контролювати контроль доступу та може надати аудиторські сліди під час розслідування інциденту. Хоча більшість журналів охоплюються деякими формами регулювання в наші дні і повинні зберігатися до тих пір, поки цього вимагають вимоги, будь-які, що не є, повинні зберігатися мінімум протягом одного року, якщо вони потрібні для розслідування. Однак моніторинг повинен здійснюватися відповідно до

відповідного законодавства. Співробітники повинні бути проінформовані про вашу діяльність з моніторингу відповідно до політики прийнятного використання мережі.

Якими б обширними не були ваші журнали, файли журналів нічого не варті, якщо ви не можете довіряти їх цілісності. Файли журналів є чудовим джерелом інформації, лише якщо ви переглядаєте їх. Просто придбання та розгортання продукту управління журналами не забезпечить жодної додаткової безпеки. Вам доведеться регулярно використовувати зібрану інформацію та аналізувати її; для програми з високим ризиком це може означати автоматизовані огляди щогодини. Контроль ISO / IEC 27001 A.10.10.2 вимагає не тільки процедур моніторингу використання засобів обробки інформації, але вимагає регулярного перегляду результатів для виявлення можливих загроз безпеці та інцидентів.

Елементи керування ISO / IEC 27001 A.10.10.4 та A.10.10.5 охоплюють дві конкретні галузі реєстрації, важливість яких часто не цілком оцінюється: діяльність адміністратора та реєстрація несправностей. Адміністратори мають потужні права, і їх дії потрібно ретельно фіксувати та перевіряти. Оскільки події, такі як перезапуск системи для виправлення серйозних помилок, можуть не реєструватися в електронному вигляді, адміністратори повинні вести письмовий журнал своєї діяльності, реєструвати час початку та закінчення події, хто брав участь та які дії були вжиті. Слід також записати ім’я особи, яка робить запис у журналі, а також дату та час. Команда внутрішнього аудиту повинна вести ці журнали.

Є два типи помилок, які слід реєструвати: несправності, генеровані системою та запущеними на ній програмами, та помилки або помилки, про які повідомляють користувачі системи. Реєстрація несправностей та аналіз часто є єдиним способом з’ясувати, що не так із системою чи застосунком. Аналіз журналів несправностей може бути використаний для виявлення тенденцій, які можуть свідчити про більш глибоко вкорінені проблеми, такі

як несправне обладнання або недостатня компетентність або підготовка користувачів або системних адміністраторів.

При проектуванні стратегії протоколювання і інструментування керуйтеся такими рекомендаціями:

– проектуйте централізований механізм протоколювання і інструментування, що забезпечує перехоплення критично важливих для системи і бізнесу подій;

– створюйте політики безпечного управління файлами журналу;

– зробіть свої слухачі трасування, що настроюються, щоб забезпечити можливість їхньої зміни під час виконання відповідно до вимог інфраструктури розгортання. Серед популярних бібліотек можна також згадати NLog і log4net для C# та logj4 для Java.

Управління станом – це питання, пов’язані зі зберіганням даних, що належать до станів компонентів, операцій або етапів процесу. Для зберігання даних стану можуть використовуватися різні формати і сховища. Таким чином, при проектуванні стратегії управління станом керуйтеся такими рекомендаціями:

– мінімізуйте об’єм даних стану;

– правильно вибирайте сховище стану. Зберігання у файлу на диску має недолік, пов’язаний зі швидкістю роботи. З протилежної сторони існують in-memory бази даних, що зберігають дані у пам’яті. Це дає швидкість роботи, але не зберігає дані після зупинки. Якщо стан є критично важливим аспектом застосунку, або якщо ці стани повинні використовуватися спільно декількома комп’ютерами, зберігайте стан централізовано, наприклад, на виділеному SQL (наприклад MySQL, MSSQL, PostgreSQL) або NoSQL (neo4j, MongoDB) сервері.

Висновки за розділом 6

У даному розділі розроблено імітаційну модель системи підвищення безпеки байт-код орієнтованого програмного забезпечення в умовах використання кібератак. Ця модель дозволила провести порівняльні дослідження та отримати кількісні значення показника ефективності розроблених моделей та методів.

Проведена оцінка ефективності розроблених моделей та методів підвищення безпеки програмного забезпечення показала, що їх використання в дисертаційної роботі дозволить збільшити час зламу програмного забезпечення до 1.4 разів.

Отримані результати дисертаційної роботи дозволили обґрунтувати рекомендації з практичного використання розроблених методів та засобів підвищення безпеки байт-код орієнтованого програмного забезпечення в умовах використання кібератак.

ОСНОВНІ ВИСНОВКИ

В дисертаційній роботі вирішено науково-практичну проблему підвищення безпеки байт-код орієнтованого програмного забезпечення в умовах кібератак на основі розробки моделей та методів синтезу підсистем обфускації програмного коду та генерації ліцензійних ключів.

Проведено дослідження та порівняльний аналіз моделей та методів безпеки байт-код орієнтованого програмного забезпечення, який показав, що існуючі моделі і методи безпеки десктопного програмного забезпечення на системному рівні проектування в повній мірі не дозволяють усунути вплив засобів формування загроз відповідних загроз безпеки. Тому, існуючі моделі і методи захисту програмного забезпечення не відповідають вимогам якості, що регламентуються міжнародними стандартами та стандартами України щодо якості програмного забезпечення. Доказано, що підвищення показників захищеності програмного забезпечення впливає на інші показники якості програмного забезпечення, а саме переносимість, супроводжуваність, продуктивність.

Основні наукові та практичні результати дисертаційної роботи:

1. Отримав подальший розвиток метод перевірки логіко-смислової подібності програм складної логічної структури, що відрізняється від відомих розпаралелюванням процесів, що обчислюються при порівнянні подібних елементів програмного коду, а також формуванням та використанням графу спільних обчислень в процесі пошуку подібних елементів коду. Це дозволило розпаралелити дані процеси та досягти поліноміальної складності процесу верифікації, що в свою чергу зменшило час перевірки коду на логіко-смислову подібність для визначення впливу розробленого методу обфускації коду на його коректність.

2. Розроблено GERT-модель процесу обфускації програмних модулів, що реалізує парадигми використання математичного апарату гамма-

розподілу у якості ключового на всіх етапах моделювання процесу обфускації. Це дозволило досягти уніфікації моделі в умовах модифікації GERT-мережі. Результати дослідження показали, що для розробленої математичної моделі при додаванні ще одного процесу обфускації дисперсія часу виконання збільшується на 12%, а при його видаленні з системи – зменшується до 13%. Математичне сподівання часу виконання змінюється в геометричній прогресії - так, при видаленні вузла відбувається зменшення математичного сподівання на 9%, а при збільшенні на 1 вузол – збільшення математичного сподівання на 26%. Це показує незначність змін досліджуваних показників в умовах модифікації моделі і підтверджує гіпотезу про уніфікацію моделі в умовах використання математичного апарату гамма-розподілу як основного. Дані результати дають розробнику можливість спрогнозувати поведінку системи захисту програмних модулів з точки зору часу виконання. Це дозволяє зменшити час на прийняття рішення про доцільність використання процесу обфускації в умовах використання гнучких методологій.

3. Вдосконалено технологію обфускації програмних модулів, що відрізняється від відомих урахуванням варіативності типів лексем та ідентифікаторів. Це дозволило підвищити безпеку програмного забезпечення.

4. Розроблено критерій оцінки якості обфускації програмного забезпечення шляхом синтезу лінійної композиції часткових критеріїв метрик якості коду, що дозволило кількісно оцінити ступінь обфускованості програмних продуктів.

5. Розроблено модель безпечного переходу і кодування ліцензійних ідентифікаторів на основі математичного апарату GERT-мереж з парадигмою гамма-розподілу, що дозволило підвищити точність результатів моделювання. Дана логіка впроваджується в залежності від ідентифікаційного або серійного номера. Розроблено методологію масштабування розробленої математичної моделі шляхом її горизонтального

і вертикального масштабування. Показано доцільність використання кожного типу масштабування з урахуванням критичності часу виконання перевірки безпеки програмного забезпечення на основі ліцензійних ідентифікаторів. Так, при вертикальному масштабуванні, в зв'язку з використанням паралельного процесу обробки даних, час обробки даних практично не змінився. Це дає передумови використання даного типу масштабування при критичності часу виконання процесу, а також при необхідності істотного нарощування довжини ліцензійного ключа на слабких пристроях (наприклад, вбудованих пристроях і IoT). При горизонтальному масштабуванні, в зв'язку з використанням додаткових послідовних процесів обробки даних, час обробки значно збільшився (в 3.43 рази). Однак, введення додаткових послідовних дій збільшує час аналізу алгоритму поведінки. Це дає передумови використання даного типу масштабування при використанні прикладного програмного забезпечення, де час запуску програми не критичний.

6. Розроблено метод формування цифрового ідентифікатора програмного забезпечення, відмінною особливістю якого є використання індивідуальних даних про комп'ютерні системи кінцевого користувача для однозначної ідентифікації приналежності, на які ліцензійне програмне забезпечення встановлюється в процесі формування ліцензійного цифрового ідентифікатора. Це дає можливість підвищити безпеку програмного забезпечення шляхом захисту від неліцензійного копіювання. Запропоновано алгоритм функціонування системи і генерації ліцензійного ключа, адаптований до вхідних даних і можливих умов верифікації програмного забезпечення. Також, модель враховує можливість вбудовування довільного (заданого розробниками) коду в тіло ліцензійного ключа, який буде виконуватися при верифікації. Дані маніпуляції призвели до ускладнення аналізу і зламу ліцензійної складової програмного забезпечення. Це дозволило підвищити середній час зламу в 1.29 разів.

Проведено оцінку достовірності та ефективності запропонованих методів і моделей підвищення безпеки байт-код орієнтованого програмного забезпечення.

Практичне значення отриманих результатів підтверджено відповідними актами впровадження.

Результати дисертації впроваджені і використовуються у діяльності компаній «Line Up», «Нікс Солюшенс ЛТД», Державного підприємства «Південний державний проектно-конструкторський та науково-дослідний інститут авіаційної промисловості», Державного підприємства «Харківський науково-дослідний інститут технологій машинобудування», а також використано у навчальному процесі Національного технічного університету «Харківський політехнічний інститут».

Таким чином, сукупність отриманих у дисертаційній роботі наукових результатів і одержана в ході проведення експериментальних досліджень оцінка їх ефективності, дозволяють вважати сформульовану наукову проблему підвищення безпеки байт-код орієнтованого програмного забезпечення в умовах кібератак на основі розробки моделей та методів синтезу підсистем обфускації програмного коду та генерації ліцензійних ключів – вирішеною, а поставлену мету підвищення безпеки програмного забезпечення в умовах використання байт-код орієнтованих мов програмування – досягнутою.

СПИСОК ЛІТЕРАТУРИ

1. Авторское право на компьютерную программу. Режим доступу: http://copyright.ua/komp.php.

2. Агаджанов М. В сеть утекли данные 49 млн пользователей Instagram, утечкой заинтересовался «Роскомнадзор». 2019. Режим доступу: https://habr.com/ru/news/t/452918/.

3. Алексеев Е. Р., Чеснокова О. В. Решение задач вычислительной математики в пакетах Mathcad 12, MATLAB 7, Maple 9. Москва, НТ Пресс. 2006. 496 с.

4. Атаки на MySQL: теория, инструменты, примеры и практика, 2019. Режим доступу: https://tgraph.io/Ataki-na-MySQL-teoriya-instrumenty-primery-i-praktika-10-19.

5. Болтенков В. А., Еникеев Р. И. Практическое исследование современных систем электронной цифровой подписи. Одесса, ОНПУ. 2014. Том 4. No. 3. с. 201-209.

6. Бородакий Ю. В., Лободинский Ю. Г. Информационные технологии в военном деле. Основы теории и практические применения. Москва, Горячая линия – Телеком, 2008. 392 с.

7. Буда А. О., Иткин В. Э. Сводимость эквивалентности программ к термальной эквивалентности // Системное и теоретическое программирование. Сборник статей. Кишенев, 1974. Т. 1. с. 293–324.

8. Буй П. М. Защита информации в телекоммуникационных системах (конспект лекций по курсу). Режим доступу: https://studfile.net/preview/5443545/.

9. Венцель Е. С. Теория вероятностей (учебник, 2-е изд., перераб. и дополн). Москва, Гос. Изд. физ.-мат. лит. 1962. 564 с.

10. Вихорев С. В. Классификация угроз информационной безопасности. 2001. Режим доступу: https://elvis.ru/upload/iblock/f60/f602ee2337fcc7250c71c2a138fe9ecc.pdf.

11. Всесвітнє дослідження економічних злочинів та шахрайства 2018 року: результати опитування українських організацій. Режим доступу: https://www.pwc.com/ua/uk/survey/2018/pwc-gecs-2018-ukr.pdf.

12. Всесвітній огляд економічних злочинів. Режим доступу: https://www.pwc.com/ua/uk/press-room/assets/gecs_ukraine_ua.pdf.

13. Галатенко В. А. Основы информационной безопасности. ИНТУИТ.РУ. 2016. ISBN 5-9556-0052-3.

14. Генератор ключей. Режим доступу: https://ru.wikipedia.org/wiki/Генератор_ключей.

15. Глушков В. М. Теория автоматов и формальные преобразования микропрограмм // Кибернетика. 1965. Вып. 5. С. 1–9.

16. Глушков В. М., Летичевский А. А. Теория дискретных преобразователей // Избранные вопросы алгебры и логики. 1973. С. 5–39.

17. Гмурман В. Е. Теория вероятностей и математическая статистика. М.: Высшая школа, 2003. 480 с. Режим доступу: http://lib.maupfib.kg/wp-content/uploads/2015/12/Teoria_veroatnosty_mat_stat.pdf.

18. Давидов В. В. Моделі та методи підвищення безпеки програмного забезпечення (монографія). Харків, 2021. 146 с.

19. Давидов В. В., Бульба С. С., Кучук Г. А. Метод розподілу ресурсів між композитними застосунками // Системи управління, навігації та зв'язку. 2018. Том 4 (50). С. 99-104. DOI: https://doi.org/10.26906/SUNZ.2018.4.

20. Давидов В. В., Волошин Д. Г. Семенов С. Г. Про завдання позиціонування безпілотних літальних апаратів в умовах кібератак // Актуальні питання протидії кіберзлочинності та торгівлі людьми : Всеукраїнська наук.-практич. конф., збірник матеріалів. Харків, 2018. С. 311.

21. Давидов В. В., Гавриленко С. Ю., Прохорова Т. М. Дослідження методів побудови синтаксичних аналізаторів // Системи обробки інформації. 2015. № 11(136). С. 125-128.

22. Давидов В. В., Гавриленко С. Ю., Челак В. В. Разработка системы фиксации аномальных состояний компьютера // Вісник Національного технічного університету "ХПІ": Серія: Інформатика та моделювання. 2018. № 42 (1318). С. 109-121.

23. Давидов В. В., Гребенюк Д. С. Метод первинного виділення хмарних обчислювальних ресурсів на основі аналізу ієрархій // Системи управління, навігації та зв'язку. 2020. Том 3 (61). С. 80-85. DOI: https://doi.org/10.26906/SUNZ.2020.3.080.

24. Давидов В. В., Гребенюк Д. С. Метод прогнозування навантаження ресурсів хмарних обчислюваних систем // Проблеми інформатизації : міжнародна наук.-технічн. конф., тези доповідей. Черкаси. 2020. С. 73.

25. Давидов В.В., Мовчан А. В., Сидоренко И. И. Разработка системы формирования цифрового идентификатора программного обеспечения для защиты авторских прав // Системи обробки інформації. 2016. № 3. С. 15-17.

26. Давидов В. В., Можаєв М. А., Ліцзян Джан. Аналіз та порівняльні дослідження методів підвищення рівня безпеки програмного забезпечення // Сучасні інформаційні технології. 2020. Том 4. № 3. С. 124–132. DOI: https://doi.org/10.20998/2522-9052.2020.3.18.

27. Давидов В.В., Пасько Д.А., Молчанов Г.І. Управління необмеженою кількістю хмарних сховищ // Сучасні інформаційні технології. 2018. Том 2. №3. С. 49–53. DOI: https://doi.org/10.20998/2522-9052.2018.3.08.

28. Давидов В. В., Семенов С. Г., Зиков І. С. Дослідження ризиків моніторингу технічного стану об'єктів авіації // Системи озброєння і військова техніка. 2015. № 4(44). С. 108-110.

29. Давыдов В. В. Анализ методов обнаружения злоумышленного кода в Android приложениях // Інформаційні технології, наука, техніка, технологія,

освіта, здоров’я: міжнар. наук.-практ. конф., тези доповідей. Харків, 2015. С. 36.

30. Давыдов В. В., Гребенюк Д. С. Комплекс процедур генерации лицензионного ключа для защиты авторских прав интеллектуальной собственности на программное обеспечение // Системи управління, навігації та зв’язку. 2017. № 1(41). С. 11-15. Режим доступу: http://journals.nupp.edu.ua/sunz/article/view/621.

31. Давыдов В. В., Гребенюк Д. С. Особенности распределения ресурсов для многомашинных вычислительных комплексов // Проблеми інформатизації: міжнар. наук.-техн. конф., тези доповідей. Полтава, 2017. С. 21.

32. Давидов В. В. Дослідження методів управління та захисту даних в комп’ютеризованих інформаційно-вимірювальних та розподілених системах (науково-дослідна робота). ДР №0119U002603.

33. Давыдов В. В. Процедура генерации лицензионного ключа для защиты авторских прав // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління : міжнар. наук.-техн. конф., тези доповідей. Полтава, 2017. С. 52.

34. Давидов В. В. Створення завдань по мові програмування "С", тестування методологічної концепції, адаптації програми для інтеграції у систему вищої освіти України (науково-дослідна робота). ДР №0119U103871.

35. Девянин П.Н. Модели безопасности компьютерных систем. Москва, Издательский центр «Академия», 2005. 144 с.

36. Ендовицкий Д. А., Соболева В. Е. Анализ инвестиционной привлекательности компании-цели M&A // Корпоративный менеджмент, 2008. Режим доступу: https://www.cfin.ru/investor/m_and_a/motive.shtml.

37. Ершов А. П. Об операторных схемах Янова // Проблемы кибернетики. 1967. Вып. 20. С. 181—200.

38. Закон України «Про авторське право і суміжні права», 1993. (Верховна Рада України). Офіційний сайт Верховної Ради України. Режим доступу: https://zakon.rada.gov.ua/laws/show/3792-12.

39. Закон України «Про захист інформації в інформаційно-телекомунікаційних системах», 1994. (Верховна Рада України). Офіційний сайт Верховної Ради України. Режим доступу: https://zakon.rada.gov.ua/laws/show/80/94-вр.

40. Закон України «Про розповсюдження примірників аудіовізуальних творів, фонограм, відеограм, комп'ютерних програм, баз даних» від 13.01.2016. Режим доступу: http://zakon3.rada.gov.ua/laws/show/1587-14.

41. Захаров В. А. Быстрые алгоритмы разрешения эквивалентности пропозициональных операторных программ на упорядоченных полугрупповых шкалах // Вестник Московского университета. Серия 15. Вычислительная математика и кибернетика. 1999. Вып. 3. С. 29–35.

42. Захаров В. А. Проверка эквивалентности программ при помощи двухленточных автоматов // Кибернетика и системный анализ. 2010. Вып. 4. С. 39–48.

43. Захаров В. А., Подымов В. В. Об одной полугрупповой модели программ, определяемой двухленточным автоматом // Научные ведомости Белгородского государственного университета. Серия История, экономика, политология, информатика. 2010. Т. 14. Вып. 7. С. 94–101.

44. Захаров В., Кузюрин Н., Подловченко Р., Щербина В. Использование алгебраических моделей программ для обнаружения метаморфного вредоносного кода // Фундаментальная и прикладная математика. 2009. Т. 15. Вып. 5. С. 181–198.

45. Золотухіна О. А., Волошин Д. Г., Давидов В. В., Бречко В. О. Розробка імітаційної моделі процесу розрахунку і коригування безпечної польотної траєкторії безпілотного літального апарату // Телекомунікаційні та інформаційні технології. 2020. № 4 (69). С. 87–94.

46. Иткин В. Э. Логико-термальная эквивалентность схем программ // Кибернетика. 1972. Вып. 1. с. 5–27.

47. Казарин О. В. Безопасность программного обеспечения компьютерных систем. Москва, МГУЛ, 2003. 212 с. Режим доступу: http://citforum.ck.ua/security/articles/kazarin/.

48. Калинин М.О. Адаптивное управление безопасностью информационных систем на основе логического моделирования: дис. … доктора техн. наук: 05.13.19. Санкт-Петербург, 2010. 308 с.

49. Касперский Е. Компьютерное зловредство. Санкт-Петербург: Питер, 2007. 208 с.

50. Коваленко А. Технология тестирования DOM XSS уязвимости // Безпека інформації. 2017. Вып. 23(2). С. 73-79. DOI: http://doi.org/10.18372/2225-5036.23.11821.

51. Коваленко О. В., Хочкин Н. И. Решение системы уравнений марковского восстановления с помощью аппроксимации асимптотических рядов // Труды МФТИ. 2015. Вып. 7(2). С. 5-19. Режим доступу: https://mipt.ru/upload/medialibrary/26d/5-19.pdf.

52. Концепция безопасности и принципы создания систем физической защиты важных промышленных объектов. Режим доступу: http://www.oskord.ru/articles/koncepciya-bezopast-principi-sozdaniya-sistemi.html.

53. Котов В. Е., Сабельфельд В. К. Теория схем программ. Москва, Наука. 1991. 348 с.

54. Круль С. М. Злочини у сфері інформаційних технологій: національний та міжнародний аспекти // Актуальні проблеми вдосконалення чинного законодавства України. 2008. Вип. 20. - С. 200-204. Режим доступу: http://nbuv.gov.ua/UJRN/apvchzu_2008_20_32.

55. Кузнецов О. О., Семенов С. Г. Протоколи захисту інформації у комп’ютерних системах та мережах. Харків, ХНУРЕ, 2009. 184 с.

56. Кучук Н. Г., Давидов В. В. Моделі і методи захисту інформаційних структур для комп’ютерних систем на інтегрованих програмних платформах (монографія). Харків, 2021. 160 с.

57. Кучук Н. Г., Давидов В. В., Гребенюк Д. С. Аналіз методів розрахунку розмірності мінковського для фрактального трафіка мультисервісної мережі e-learning // Проблеми інформатизації : міжнародна наук.-техн. конф., тези доповідей. Черкаси, 2018. С. 34.

58. Летичевский А. А. Функциональная эквивалентность дискретных преобразователей I // Кибернетика. 1969. Вып. 2. С. 5–15.

59. Летичевский А. А. Функциональная эквивалентность дискретных преобразователей II // Кибернетика. 1970. Вып. 2. С. 14–28.

60. Летичевский А. А. Функциональная эквивалентность дискретных преобразователей III // Кибернетика. 1972. Вып. 1. С. 1–4.

61. Летичевский А. А. Эквивалентность автоматов относительно полугрупп с сокращением // Проблемы кибернетики. 1973. Вып. 27. с. 195–212.

62. Лисовик Л. П. Металинейные схемы с засылками констант // Программирование. 1985. Вып. 2. с. 29–38.

63. Лукацкий А. В. Обнаружение атак. Санкт-Петербург - BHV, 2003. 596 с.

64. Ляпунов А. А. О логических схемах программ // Проблемы кибернетики. 1958. Вып. 1. С. 46–74.

65. Ляпунов А. А., Янов Ю. И. О логических схемах программ // Труды конференции "Пути развития советского математического машиностроения и приборостроения". 1956. Часть 3. с. 5–8.

66. Подловченко Р. И. Иерархия моделей программ // Программирование. 1981. Вып. 2. С. 3–14.

67. Подловченко Р. И. Полугрупповые модели программ // Программирование. 1981. Вып. 4. С. 3–13.

68. Подловченко Р. И. Рекурсивные программы и иерархия их моделей // Программирование. 1991. Вып. 6. С. 44–51.

69. Подловченко Р. И. От схем Янова к теории моделей программ // Математические вопросы кибернетики. 1998. Вып. 7. С. 281–302.

70. Подловченко Р. Техника следов в разрешении проблемы эквивалентности в алгебраических моделях программ // Кибернетика и системный анализ. 2009.

71. Подловченко Р. И., Долгих Б. А. Двухступенчатое моделирование программ с процедурами // Математические вопросы кибернетики. 2004. Вып. 12. С. 47–56.

72. Подымов В. В. О проверке эквивалентности последовательных и рекурсивных программ на упорядоченных полугрупповых шкалах // Материалы X Международной конференции “Интеллектуальные системы и компьютерные науки”. 2011. С. 295–298.

73. Подымов В. В. Алгоритм проверки эквивалентности линейных унарных рекурсивных программ на упорядоченных полугрупповых шкалах // Вестник Московского университета. Серия 15. Вычислительная математика и кибернетика. 2012. Вып. 4. С. 37–43.

74. Подымов В. В. О проверке сильной эквивалентности металинейных унарных рекурсивных программ // Вестник Московского университета. Серия 15. Вычислительная математика и кибернетика. 2013. Вып. 1. С. 21–27.

75. Проснеков С. А. Выбор метрик оценки безопасности ERP систем // Международный научно-исследовательский журнал. 2017. № 07 (61). С. 68-71. DOI: https://doi.org/10.23670/IRJ.2017.61.007.

76. Раскин Л. Г., Пустовойтов П. Е., Са’ди Ахмад Абдельхамид Саед Ахмад. Марковская аппроксимация немарковских систем // Інформаційно-керуючі системи на залізничному транспорті. 2006. Вып. 1. с. 57-60. Режим доступу: http://repository.kpi.kharkov.ua/bitstream/KhPI-Press/6801/1/2006_Raskin_Markovskaya.pdf.

77. Рекомендации по организации комплексной централизованной охраны банковских устройств самообслуживания Р 78.36.035-2013. Режим доступу: http://www.ktso.ru/normdoc10/r78_36_035-2013/r78_36_035-2013_4-1.php.

78. Рудницький В. М., Мельник О. Г., Щербина В. П., Миронюк Т. В. Синтез елементарних функцій перестановок, керованих інформацією // Безпека інформації. 2014. Том 20. №3. С. 242-247. DOI: https://doi.org/10.18372/2225-5036.20.7550.

79. Рудницький В. М., Опірський І. Р., Мельник О. Г., Пустовіт М. О. Синтез групи операцій строгого стійкого криптографічного кодування для побудови потокових шифрів // Безпека інформації. 2018. Том 24. № 3. С. 195-200. DOI: https://doi.org/10.18372/2225-5036.24.13430.

80. Сабельфельд В. К. Полиномиальная оценка сложности распознавания логико-термальной эквивалентности // ДАН СССР. 1979. Т. 249, Вып. 4. с. 793–796.

81. Семенов С. Г., Гавриленко С. Ю., Мовчан А. В. Исследования технологий динамического анализа бинарного кода программного обеспечения // Матеріали Всеукраїнської НПК «Комп'ютерні системи і проектування технологічних процесів та обладнання». Чернівці, ЧФ НТУ «ХПІ». 2016. С. 152.

82. Семенов С. Г., Давидов В. В., Волошин Д. Г., Гребенюк Д. С. Метод захисту модуля програмного забезпечення на основі процедури обфускації // Телекомунікаційні та інформаційні технології. 2019. № 4 (65). С. 71–80. DOI: https://doi.org/10.31673/2412-4338.2019.047180.

83. Семенов С.Г., Давыдов В. В., Гавриленко С. Ю. Защита данных в компьютеризированных управляющих системах. LAP Lambert Academic Publishing GmbH & Co. KG. 2014. 236 с.

84. Семенов С. Г., Давыдов В. В. Имитационная модель и практические рекомендации использования метода динамического распределения доступа

и антивирусной защиты данных Государственной службы по чрезвычайным ситуациям // Системи управління, навігації та зв'язку. Полтава, ПНТУ ім. Ю Кондратюка. 2014. Вип. 2(30). С. 90-95.

85. Семенов С. Г., Давыдов В. В., Мовчан А. В. Система формирования цифрового идентификатора программного обеспечения для защиты авторских прав // Сучасні проблеми інформатики в управлінні, економіці, освіті та подоланні наслідків Чорнобильської катастрофи : міжнарод. наук. сем., тези доповідей. Київ, 2016. С. 110-116.

86. Семенов С. Г., Давидов В. В. Оценка рисков контроля и диагностики технического состояния объектов критического применения // Metrology and Metrology assurance 2016. 26th National Scientific Symposium with international participation. Sozopol, Bulgaria. 2016. P. 395-399.

87. Семенов С. Г. Исследование методов идентификации программного обеспечения и их характеристик // Системи обробки інформації. Харків, ХУПС. 2015. Вип. 12 (137). С. 148-150.

88. Система обеспечения информационной безопасности сети связи общего пользования: ГОСТ Р 53110-2008. В действии с 18 декабря 2008. Москва, Стандартинформ, 2009. Режим доступу: https://pdf.standartgost.ru/catalog/Data1/56/56923/.

89. Слукин А. М. Методы и средства защиты объектов (курс лекций) // Тольятти, Тольяттинский государственный университет. 2007. 196 с. Режим доступу: http://edu.tltsu.ru/sites/sites_content/site59/html/media3772/MSZO_lek_S.pdf.

90. Соблюдение лицензионного соглашения программного обеспечения. Режим доступу: http://www.pcwork.ru/soblyudenie_litsenzionnogo_soglasheniya_programmnogo_obespecheniya_chast_2_.htm.

91. Трещев И. А. О классификации угроз безопасности конфиденциальной информации предприятия // Мир науки. 2014. Вып. 3. 15KMN314. Режим доступу: https://mir-nauki.com/PDF/15KMN314.pdf.

92. Угрозы безопасности программного обеспечения. 2017. Режим доступу: https://studopedia.ru/19_24788_ugrozi-bezopasnosti-programmnogo-obespecheniya.html.

93. Цибульов П. М., Горнісевич А. М., Болєлий С. М. Законодавство України про інтелектуальну власність. Тематична збірка: у 3-х томах. Том 1: Законодавчі акти України про інтелектуальну власність // Київ, Ін-т. Інтел. власн. і права. 2005. 168 с.

94. Цифровые подписи в исполняемых файлах и обход этой защиты во вредоносных программах. Режим доступу: https://habrahabr.ru/post/112289/.

95. Шибанов А. П. Нахождение плотности распределения времени исполнения GERT-сети на основе эквивалентных упрощающих преобразований // Автоматика и телемеханика. 2003. Вып. 64(2). с. 117-126. DOI: https://doi.org/10.1023%2FA%3A1022267115444.

96. Янов Ю. И. О логических схемах алгоритмов // Проблемы кибернетики. 1958. Вып. 1. С. 75–127.

97. A Look At 5 of the Most Popular Programming Languages of 2019. Available at: https://stackify.com/popular-programming-languages-2018/.

98. Aho A. V., Lam M. S., Sethi R., Ullman J. D. Compilers: principles, techniques, and tools. Second edition. Addison-Wesley, 2007.

99. Aldea M., Gheorghică D. and Croitoru V. Software Vulnerabilities Integrated Management System // 2020 13th International Conference on Communications (COMM), Bucharest, Romania, pp. 97-102, DOI: https://doi.org/10.1109/COMM48946.2020.9141970.

100. Another Tool for Language Recognition. Available at: https://www.antlr.org/.

101. Ashcroft E., Manna Z., Pnueli A. Decidable properties of monadic functional schemes // Journal of the ACM. 1973. Vol. 20, No. 3. P. 489–499.

102. Baader F. Unification, weak unification, upper bound, lower bound, and generalization problems // Ronald V. Book, editor, RTA, Springer. 1991. V. 488 of Lecture Notes in Computer Science. p. 86–97.

103. Baader F., Snyder W. Unification theory // J. A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning. 2001. V. 1. p. 447–533.

104. Bahaa-Eldin A. M., Sobh M.A.A. A comprehensive Software Copy Protection and Digital Rights Management platform // Ain Shams Engineering Journal. 2014. Volume 5. Issue 3. P. 703-720.

105. Baker B. On finding duplication and near-duplication in large software systems // Proceedings of the 2nd Working Conference on Reverse Engineering. 1995. p. 86-–95.

106. Bannwart F., M¨uller P. Changing Programs Correctly: Refactoring with Specifications // Proceedings of the 14th International Symposium on Formal Methods. 2006. P. 492–507.

107. Barak B. On the (Im)possibility of Obfuscating Programs // Advances in Cryptology – CRYPTO 2001. Lecture Notes in Computer Science. 2001. Vol. 2139. DOI: https://doi.org/10.1007/3-540-44647-8_1.

108. Barak B., Goldreich O., Impagliazzo R., Rudich S., Sahai A., Vadhan S. and Yang K. On the (im) possibility of obfuscating programs // Journal of the ACM. 2001. Vol. 2. No. 69. DOI: https://doi.org/10.1145/2160158.2160159.

109. BCrypt. Режим доступу: https://en.wikipedia.org/wiki/Bcrypt.

110. Bird R. The equivalence problem for deterministic two-tape automata // J. Of Computer and System Science. 1973. V. 7, No. 4. p. 218–236.

111. Buch R., Shah S. Network sniffers and tools in cyber security // Journal of Emerging Technologies and Innovative Research (JETIR), November 2018. P. 685-695. Available at:

https://www.researchgate.net/publication/329016793_NETWORK_SNIFFERS_AND_TOOLS_IN_CYBER_SECURITY.

112. Burgelmana J., Vanhoucke M. Computing project makespan distributions: Markovian PERT networks revisited // Computers & Operations Research. 2019. No. 103. p. 123-133. DOI: https://doi.org/10.1016/j.cor.2018.10.017.

113. Christodorescu M., Jha S. Static Analysis of Executables to Detect Malicious Patterns // Proceedings of the 12th USENIX Security Symposium. 2003. P. 169–186.

114. Christodorescu M., Jha S., Seshia S., Song D., Bryant R. E. Semantics-Aware Malware Detection // Proceedings of the 2005 IEEE Symposium on Security and Privacy. 2005. P. 32–46.

115. Cobb M. Best practices for audit, log review for IT security investigations. 2011. Available at: https://www.computerweekly.com/tip/Best-practices-for-audit-log-review-for-IT-security-investigations.

116. Collberg C., Huntwork A., Carter E., Townsend G. Graph Theoretic Software Watermarks: Implementation, Analysis, and Attacks. Information Hiding. Lecture Notes in Computer Science. 2004. Vol. 3200. DOI: https://doi.org/10.1007/978-3-540-30114-1_14.

117. Collberg C., Thomborson C., Low D. A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, 1997. Available at: https://www.researchgate.net/publication/37987523_A_Taxonomy_of_Obfuscating_Transformations.

118. Collberg C., Thomborson C., Low D. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs // Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1998. P. 184–196.

119. CVE Details. Oracle Javafx: List of security vulnerabilities. Available at: https://www.cvedetails.com/vulnerability-list/vendor_id-93/product_id-21383/Oracle-Javafx.html.

120. Davydov V., Hrebeniuk D. Development of the methods for resource reallocation in cloud computing systems // Innovative Technologies and Scientific Solutions for Industries. 2020. № 3 (13), P. 25–33. DOI: https://doi.org/10.30837/ITSSI.2020.13.025.

121. Davydov V., Hrebeniuk D. Development the resources load variation forecasting method within cloud computing systems // Advanced Information Systems. 2020. Vol. 4. No. 4. P. 128–135. DOI: https://doi.org/10.20998/2522-9052.2020.3.18.

122. Davydov V., Zmiivska V., Shypova T., Lysytsia D. Analysis of fractal noise indicators in measuring systems of technical objects // Metrology and metrology assurance 2018 : 28th International Scientific Symposium, September 10-14, 2018 : Sozopol, Bulgaria. P. 44-47.

123. De Bakker J. W., Scott D. A. Theory of programs. Unpublished notes. Vienna: IBM Seminar. 1969.

124. De Souza R. On the Decidability of the Equivalence for k-Valued Transducers // Proceedings of the 12th International Conference on Developments in Language Theory. 2008. P. 252–263.

125. Deinum M., Cosmina I. Pro Spring MVC with WebFlux. Chapter Web Application Architecture. Apress, 2021. P. 55-72. Available at: https://doi.org/10.1007/978-1-4842-5666-4.

126. Denning P. J., Lewis T. G. Exponential laws of computing growth // Communications of the ACM. 2017. No. 60 (1), p. 54-65. DOI: http://doi.org/10.1145/2976758.

127. Ding N., Gu D. A Note on (Im)Possibilities of Obfuscating Programs of Zero-Knowledge Proofs of Knowledge // Cryptology and Network Security.

Lecture Notes in Computer Science. 2011. Vol. 7092. DOI: https://doi.org/10.1007/978-3-642-25513-7_20.

128. Distefanoab S., Longoa F., Scarpaa M. Marking dependency in non-Markovian stochastic Petri nets // Performance Evaluation. 2017. No. 110. p. 22-47. DOI: https://doi.org/10.1016/j.peva.2017.03.001.

129. Eder E. Properties of substitutions and unifications // Journal of Symbolic Computations. V. 1. 1985. p. 31–46.

130. El-Khalil R., Keromytis A. D. Hydan: Hiding Information in Program Binaries // Information and Communications Security. Lecture Notes in Computer Science. 2004. Vol 3269. DOI: https://doi.org/10.1007/978-3-540-30191-2_15.

131. Faruki P., Fereidooni H., Laxmi V., Conti M., Gaur M. Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions. arXiv:1611.10231. 2016. 37 p.

132. Fowler M., Beck K., Brant J., Opdyke W. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

133. Fowler M., Rice D., Foemmel M., Hieatt E., Mee D., Stafford R. Patterns of Enterprise Application Architecture. The Addison-Wesley Signature Series, 2002. Available at: https://martinfowler.com/books/eaa.html.

134. Friedman E. P. Equivalence problems for deterministic languages and monadic recursion schemes // Journal of Computer and System Sciences. 1977. Vol. 14, No. 3. P. 342–359.

135. Galatenko V.A. Fundamentals of Information Security. Moscow, National Open University "INTUIT": 267. Print. ISBN 5-9556-0052-3.

136. Garg S., Gentry C., Halevi S., Raykova M., Sahai A., and Waters B. Candidate indistinguishability obfuscation and functional encryption for all circuits // FOCS-2013. 2013.

137. Garland S. J., Luckham D. C. Program schemes, recursion schemes and formal languages // Journal of Computer and System Sciences. 1973. Vol. 7, No. 2. P. 119–160.

138. Goldwasser S., Rothblum G. N. On best-possible obfuscation // Journal of Cryptology. 2014. Vol. 27, No. 3. P. 480–505.

139. Google Vault API. Available at: https://developers.google.com/vault.

140. Grassi P., Fenton J., Newton E., Perlner R., Regenscheid A., Burr W., Richer J. NIST Special Publication 800-63B. Digital Identity Guidelines. U.S. Department of Commerce, National Institute of Standards and Technology. 2020. DOI: https://doi.org/10.6028/NIST.SP.800-63b.

141. Hachez G. A comparative study of software protection tools suited for e-commerce with contributions to software watermarking and smart cards // CiteCeerX. 2003.

142. Henk C. A., Tiborg V. Encyclopedia of cryptography and security // Eindhoven University of Technology, Springer, Boston, MA. 2007. 684 p. DOI: https://doi.org/10.1007/978-1-4419-5906-5.

143. Hickey M., Arcuri J. Hands on Hacking. John Wiley & Sons, 2020. 580 p. Available at: https://onlinelibrary.wiley.com/doi/book/10.1002/9781119561507.

144. Hua L., Liua Z., Hua W., Wangb Y., Tana J., Wuc F. (2020). Petri-net-based dynamic scheduling of flexible manufacturing system via deep reinforcement learning with graph convolutional network // Journal of Manufacturing Systems. 2020. No. 55. p. 1-14. https://doi.org/10.1016/j.jmsy.2020.02.004.

145. Introduction to MapDB. Режим доступу: https://www.gitbook.com/book/jankotek/mapdb/details.

146. Introduction to MongoDB. Режим доступу: https://docs.mongodb.com/manual/introduction/.

147. ISO/IEC/IEEE 29148 Systems and software engineering — Life cycle processes — Requirements engineering, available at: https://www.iso.org/ru/standard/72089.html.

148. ISO/IEC 9126-1 Software engineering — Product quality — Part 1: Quality model, available at: https://www.iso.org/ru/standard/22749.html.

149. ISO/IEC/IEEE 12207 Systems and software engineering — Software life cycle processes, available at: https://www.iso.org/ru/standard/63712.html.

150. ISO/IEC 25024 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of data quality, available at: https://www.iso.org/ru/standard/35749.html.

151. ISO/IEC 25010 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, available at: https://www.iso.org/ru/standard/35733.html.

152. ISO/IEC 25012 Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Data quality model, available at: https://www.iso.org/ru/standard/35736.html.

153. Itkin V. E., Zwinogrodski Z. On program schemata equivalence // Journal of Computer and System Science. 1972. V. 6, No. 1. p. 88–101.

154. Jenkov J. Exception Handling Strategy – Overview. 2014. Available at: http://tutorials.jenkov.com/exception-handling-strategies/overview.html.

155. Java 8 Stream filter. Available at: https://vertex-academy.com/tutorials/ru/java-8-stream-filter/.

156. Jiang S., & Yang S. An Improved Multiobjective Optimization Evolutionary Algorithm Based on Decomposition for Complex Pareto Fronts // IEEE Transactions on Cybernetics. 2016. No. 46(2). p. 421-437. DOI: https://doi.org/10.1109/TCYB.2015.2403131.

157. Kharchenko V., Dotsenko S., Illiashenko O., Kamenskyi S. Integrated Cyber Safety & Security Management System: Industry 4.0 Issue // 2019 10th International Conference on Dependable Systems, Services and Technologies (DESSERT). 2019. P. 197-201. DOI: http://doi.org/10.1109/DESSERT.2019.8770010.

158. Knebl H. Hashing. In: Algorithms and Data Structures // Springer, Cham. 2020. https://doi.org/10.1007/978-3-030-59758-0_3.

159. Kolisnyk M., Kharchenko V., Piskachova I. Research of the attacks spread model on the smart office's router // International Journal of Computing. 2020. P. 629-637. DOI: http://doi.org/10.47839/ijc.19.4.1998.

160. Kuchuk N., Cherneva G., Davydov V. Method of packet fragmentation in unstable data exchange in computer networks on transport // Mechanics Transport Communications : Academic journal. Vol. 19. Is. 1. 2021. P. X1-1 – X1-7, Article № 2064.

161. Kuchuk N., Shyman A., Hrebeniuk D., Davydov V. Mathematical model of the information system synthesis process // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління : міжнародна наук.-техн. конф., матеріали. – Баку: ВА ЗС АР; Харків: НТУ «ХПІ»; Київ: НАУ; Харків: ДП «ПДПРОНДІАВІАПРОМ»; Жиліна: Університет, 2021. С. 21.

162. Kumar P., Kumar V. R. Secure Cyber Network to Sharing Information through Cryptography & Stenography // Engineering technology open access journal. 2019. Vol. 2. Issue 5. P. 129-133. DOI: https://doi.org/10.19080/ETOAJ.2019.02.555598.

163. Kundu S., Tatlock Z., Lerner S. Proving Optimizations Correct Using Parameterized Program Equivalence // Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. 2009. P. 327–337.

164. Kuznetsov A., Lutsenko M., Kiian N., Makushenko T., Kuznetsova T. Code-based key encapsulation mechanisms for post-quantum standardization // 2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT). 2018. P. 276-281. DOI: http://doi.org/10.1109/DESSERT.2018.8409144.

165. Kuznetsov A., Pushkar'ov A., Kiyan N., Kuznetsova T. (2018). Code-based electronic digital signature // 2018 IEEE 9th International Conference on

Dependable Systems, Services and Technologies (DESSERT). 2018. P. 331-336. DOI: http://doi.org/10.1109/DESSERT.2018.8409154.

166. Kuznetsov A., Shekhanin K., Kolhatin A., Mikheev I., Belozertsev I. Hiding data in the structure of the FAT family file system // 2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT). 2018. P. 337-342. DOI: http://doi.org/10.1109/DESSERT.2018.8409155.

167. Kuznetsov A., Stefanovych O., Prokopovych-Tkachenko D., Kuznetsova K. (2019). 3D Steganography Information Hiding // Telecommunications and Radio Engineering. 2019. No. 78 (12). P. 1049-1061. DOI: http://doi.org/10.1615/TelecomRadEng.v78.i12.30.

168. Lacasa L., Mariño I. P., Miguez J., Nicosia V., Roldán É., Lisica A., Grill S. W., & Gómez-Gardeñes J. Multiplex Decomposition of Non-Markovian Dynamics and the Hidden Layer Reconstruction Problem // Physical Review X. 2018. No. 8(3). 031038. DOI: https://doi.org/10.1103/PhysRevX.8.031038.

169. Lague B., Proulx D., Mayrand J et al. Assessing the benefits of incorporating function clone detection in a development process // Proceedings of the International Conference on Software Maintenance. Washington, DC, USA: IEEE Computer Society. 1997. p. 314–321.

170. Lerner S., Millstein T. D., Chambers C. Automatically proving the correctness of compiler optimizations // Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. 2003. P. 220–231.

171. Liqiang Zhang, Weiling Cao, Davydov V., Brechko V. Analysis and comparative research of the main approaches to the mathematical formalization of the penetration testing process // Information Processing Systems. 2021. № 2 (64). С.73-77.

172. Luckham D. C., Park D. M., Paterson M. S. On formalized computer programs // Journal of Computer and System Science. 1970. V. 4, No. 3. p. 220–249.

173. Löfberg Martin, Molin Patrik. Web vs. Standalone Application - A maintenance application for Business Intelligence. Blekinge Institute of Technology, 2005. 36 p. Available at: https://www.diva-portal.org/smash/get/diva2:828988/FULLTEXT01.pdf.

174. Martelli A., Montanari U. An Effective Unification Algorithm // Transactions on Programming Languages and Systems (TOPLAS). 1982. V. 4, No. 2. p. 258–282.

175. Martinez R., Alejo A., Portugal P., Cuno A., Zapata F., Saavedra R. An artifact for X.509 digital certificates delivery. // 2019 38th International Conference of the Chilean Computer Science Society (SCCC). 2019. P. 1-8. DOI: https://doi.org/10.1109/SCCC49216.2019.8966448.

176. Milutin A. Metrics and software support. Available at: http://www.viva64.com/ru/a/0045/.

177. Mitsch S., Quesel J.-D., Platzer A. Refactoring, Refinement, and Reasoning - A Logical Characterization for Hybrid Systems // Proceedings of the 19th International Symposium on Formal Methods. 2014. P. 481–496.

178. Mohsen R., Pinto A. M. Algorithmic information theory for obfuscation security // Proceedings of 12th International Joint Conference on e-Business and Telecommunications. 2015. P. 76-87.

179. Myles G., Collberg C. Software Watermarking Through Register Allocation: Implementation, Analysis, and Attacks // Information Security and Cryptology - ICISC 2003. Lecture Notes in Computer Science. 2004. Vol. 2971. DOI: https://doi.org/10.1007/978-3-540-24691-6_21.

180. Nematollahi M. A., Vorakulpipat C., Rosales H. G. Software Watermarking // Digital Watermarking. Springer Topics in Signal Processing. 2017. Vol. 11. DOI: https://doi.org/10.1007/978-981-10-2095-7_9.

181. .NET Compiler Platform ("Roslyn"). Available at: https://github.com/dotnet/roslyn.

182. Nmap Network Scanning. Chapter 11. Defenses Against Nmap. Available at: https://nmap.org/book/defenses.html.

183. Novichkov A. Metrics and their practical implementation in IBM Rational ClearCase. Available at: http://www.viva64.com/go.php?url=241.

184. Orr C. What Is SCM (Security Configuration Management). 2020. Available at: https://www.tripwire.com/state-of-security/security-data-protection/security-controls/security-configuration-management/.

185. OWASP. XSS Filter Evasion Cheat Sheet. Available at: https://owasp.org/www-community/xss-filter-evasion-cheatsheet.

186. OWASP Cheat Sheet Series. Authentication Cheat Sheet. Available at: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html.

187. OWASP Cheat Sheet Series. Cross-Site Request Forgery Prevention Cheat Sheet. Available at: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

188. OWASP Cheat Sheet Series. Error Handling Cheat Sheet. Available at: https://cheatsheetseries.owasp.org/cheatsheets/Error_Handling_Cheat_Sheet.html.

189. OWASP Cheat Sheet Series. Multifactor Authentication Cheat Sheet. Available at: https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html.

190. OWASP Cheat Sheet Series. Password Storage Cheat Sheet. Available at: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html - maximum-password-lengths.

191. OWASP Top 10. Available at: https://owasp.org/www-project-top-ten/.

192. Palamidessi C. Algebraic properties of idempotent substitutions // Lecture Notes in Computer Science. 1990. V. 443. p. 386–399.

193. Password Policy. Режим доступу: https://en.wikipedia.org/wiki/Password_policy.

194. Paterson M. S. Program schemata // Machine Intelligence. 1968. Vol. 3. P. 19–31.

195. Paterson M. S., Hewitt C. T. Comparative Schematology // Proceedings of the ACM Conference on Concurrent Systems and Parallel Computation. 1970. P. 119–127.

196. Paterson M.S., Wegman M.N. Linear unification // The Journal of Computer and System Science. 1983. V. 16, No. 2. p. 158–167.

197. Pentest – Active Audit Agency. Available at: https://auditagency.com.ua/pentest/.

198. PhoenixNAP. 7 Tactics To Prevent DDoS Attacks & Keep Your Website Safe, 2018. Available at: https://phoenixnap.com/blog/prevent-ddos-attacks.

199. Plate H. Policy and Security Configuration Management // TrustBus 2012: Trust, Privacy and Security in Digital Business. 2012. P. 229-231. DOI: https://doi.org/10.1007/978-3-642-32287-7_26.

200. Potii O., Tsyplinskyi Yu., Illiashenko O., Kharchenko V. Criticality Assessment of Critical Information Infrastructure Objects: A Category Based Methodology and Ukrainian Experience // 10th International Conference on Multimedia Communications, Services and Security. At: Krakow, Poland. 2020. DOI: http://doi.org/10.1007/978-3-030-59000-0_7.

201. Programming Concepts: Compiled and Interpreted Languages. Available at: https://thecodeboss.dev/2015/07/programming-concepts-compiled-and-interpreted-languages/.

202. Rahli V., Bickford M., Anand A. Formal Program Optimization in Nuprl Using Computational Equivalence and Partial Types // Proceedings of the 4th Conference on Interactive Theorem Proving. 2013. P. 261–278.

203. Rashid A., Chivers H., Danezis G., Lupu E., Martin A. CyBOK: The Cyber Security Body of Knowledge. 2019. Available at: https://www.cybok.org/media/downloads/cybok_version_1.0.pdf?fbclid=IwAR09O-IC8l2xkkVILuX5LuAb3Mg5BnBPs4sOHHGBjp2gtOIeq_XTnmbARzc.

204. Replacing the conditional operator with the polymorphism. Available at: https://refactoring.guru/ru/replace-conditional-with-polymorphism.

205. Rice H. G. Classes of Recursively Enumerable Sets and Their Decision Problems // Transactions of the American Mathematical Society. 1953. Vol. 74. P. 358–366.

206. Robert H. Sloan, Richard Warner. Why Don't We Defend Better? Chapter 6: Software Vulnerabilities, available at: https://www.taylorfrancis.com/chapters/software-vulnerabilities-robert-sloan-richard-warner/10.1201/9781351127301-2.

207. Roger A. Grimes. Hacking the Hacker: Learn from the Experts Who Take Down Hackers, Chapter: Software Vulnerabilities, DOI: https://doi.org/10.1002/9781119396260.ch6.

208. Roy C. K., Cordy J. R. A survey on software clone detection research // Technical report TR 2007-541, School of Computing, Queen's University. 2007. V. 115.

209. Roy C. K., Cordy J. R. An empirical study of function clones in open source software systems // Proceedings of the 15th Working Conference on Reverse Engineering. 2008. p. 81-90.

210. Roy C.K., Zibran M.F., Koschke R. The vision of software clone management: past, present and future // Proc. CSMR- 18/WCRE-21 - SEW'14, Belgium. 2014. pp. 16.

211. Schafer M., Ekman T., de Moor O. Challenge proposal: verification of refactorings // Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification. 2009. P. 67–72.

212. Schrittwieser S., Katzenbeisser S., Kinder J., Merzdovnik G., Weippl E. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? // ACM Comput. Surv. 2016. No. 49, 1. Article 4 (April 2016). 40 p.

213. Semenov S., Bartosh M., Davydov V., Turuta O. Improvement of the Task Scheduler Model Taking Into Account the Heterogeneity of the Entities // Fifth International Scientific and Technical Conference "Computer and Information Systems and Technologies". 2021. P. 42-43. DOI: https://doi.org/10.30837/csitic52021232181.

214. Semenov S., Davydov V., Hrebeniuk D. Research of the software security model and requirements // Advanced Information Systems. 2021. Vol. 5. № 1. P. 87–92. DOI: https://doi.org/10.20998/2522-9052.2021.1.12.

215. Semenov S., Davydov V., Lipchanska O., Lipchanskyi M. Development of unified mathematical model of programming modules obfuscation process based on graphic evaluation and review method // Eastern-European Journal of Enterprise Technologies. 2020. № 3(2(105)). P. 6–16. DOI: https://doi.org/10.15587/1729-4061.2020.206232.

216. Semenov S., Davydov V., Semenova A., Voloshyn D., Lymarenko V. Method of UAVs Quasi-Autonomous Positioning in the External Cyber Attacks Conditions // 10th International Conference on Dependable Systems, Services and Technologies (DESSERT). 2019.

217. Semenov S., Davydov V., Voloshyn D. Data Protection Method of an Unmanned Aerial Vehicle based on Obfuscation Procedure // CEUR Workshop Proceedings, 2020, Volume 2654. P. 515-525.

218. Semenov S., Davydov V., Voloshyn D. Obfuscated Code Quality Measurement // Metrology and metrology assurance 2019 : 29th International Scientific Symposium. September 6-9, 2019. Sozopol, Bulgaria. P. 44-47.

219. Semenov S., Davydov V., Weilin C., Liqiang Z., Petrovskaya I. Enhanced Software vulnerability model // Інформаційні технології і безпека : міжнародна наук.-практ. конф., матеріали. Київ, 2020. С. 56-60.

220. Semenov S., Hrebeniuk D., Davydov V. Software copyright protection using identification key // Aviation in the XXI-st Century: the 7th World Congress, September 19-21, 2016: Kyiv, Ukraine. P. 1.10.20-1.10.23.

221. Semenov S., Liqiang Zhang, Weiling Cao, Davydov V. Development of protecting a software product methematical model from unlicensed copying based on the GERT method // Information Processing Systems. 2021. № 1 (164). P. 73-82. DOI: https://doi.org/10.30748/soi.2021.164.08.

222. Semenov S., Sira O., Kuchuk N. Development of graphicanalytical models for the software security testing algorithm // Eastern-European Journal of Enterprise Technologies. 2018. No. 2(94). p. 39-46. DOI: https://doi.org/10.15587/1729-4061.2018.127210.

223. Semyonov S. G., Gavrilenko S. Y., Chelak V. V. Information processing on the computer system state using probabilistic automata // Proceedings of 2nd International Ural Conference on Measurements. 2017. p. 11-14. DOI: https://doi.org/10.1109/URALCON.2017.8120680.

224. Sheikh A. Public Key Infrastructure // CompTIA Security+ Certification Study Guide, Network Security Essentials. 2020. P. 251-258. http://doi.org/10.1007/978-1-4842-6234-4_16.

225. Sheng Z., Hu Q., Liu J., & Yu D. Residual life prediction for complex systems with multi-phase degradation by ARMA-filtered hidden Markov model // Quality Technology & Quantitative Management. 2019. No. 16(1). p. 19-35. DOI: https://doi.org/10.1080/16843703.2017.1335496.

226. Siriwardena P. Advanced API Security. Chapter JWT, JWS, and JWE. Apress, 2014. P. 201-220. DOI: https://doi.org/10.1007/978-1-4302-6817-8_13.

227. Strong H. R. Translating recursive equations into flow-charts // Journal of Computer and System Science. 1971. Vol. 5, No. 3. P. 254–285.

228. Strzałka D., Dymora P., Mazurek M. Modified stretched exponential model of computer system resources management limitations —The case of cache

memory // Physica A: Statistical Mechanics and its Applications. 2018. No. 491. p. 490-497. DOI: https://doi.org/10.1016/j.physa.2017.09.012.

229. Swinhoe D. The 15 biggest data breaches of the 21st century. 2021. Available at: https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html.

230. Tetskyi A., Kharchenko V., Uzun D. Neural networks based choice of tools for penetration testing of web applications // 2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT). 2018. P. 402-405. DOI: http://doi.org/10.1109/DESSERT.2018.8409167.

231. Top Computer Languages. Available at: http://statisticstimes.com/tech/top-computer-languages.php.

232. Turchin V. F. The concept of a supercompiler // ACM Transactions on Programming Languages and Systems. 1986. V. 8. p. 292–325.

233. Vault Project. Available at: https://www.vaultproject.io/.

234. Venkatesan R., Vazirani V., Sinha S. A Graph Theoretic Approach to Software Watermarking // Information Hiding. Lecture Notes in Computer Science. 2001. Vol. 2137. DOI: https://doi.org/10.1007/3-540-45496-9_12.

235. Venkatesh S., Ertaul L. Novel Obfuscation Algorithms for Software Security // Proceedings of the 2005 International Conference on Software Engineering Research and Practice. 2005. No. 1. P. 209-215.

236. Wazan A., Laborde R., Chadwick D., Venant R., Benzekri A., Billoir E., Alfandi O. On the Validation of Web X.509 Certificates by TLS interception products // IEEE Transactions on Dependable and Secure Computing. 2020. P. 1-1. DOI: https://doi.org/10.1109/TDSC.2020.3000595.

237. Webster M., Malcolm G. Detection of metamorphic and virtualization-based malware using algebraic specification // Journal in Computer Virology. 2009. Vol. 5, No. 3. P. 221–245.

238. What's The Average Web App Development Cost? Available at: https://perfectial.com/blog/average-web- app-development-cost/.

239. Winandy M., Cremers A., Langweg H., Spalka A. Protecting Java component integrity against trojan horse programs // International Federation for Information Processing, 2003. P. 99-113. Available at: https://link.springer.com/content/pdf/10.1007%2F978-0-387-35693-8_6.pdf.

240. Wong W., Stamp M. Hunting for metamorphic engines // Journal in Computer Virology. 2006. Vol. 2, No. 3. P. 211–229.

241. Write LINQ queries in C#. Available at: https://docs.microsoft.com/en-us/dotnet/csharp/linq/write-linq-queries.

242. Xu H., Zhou Y., Ming J. Layered obfuscation: a taxonomy of software obfuscation techniques for layered security // Cybersecurity. 2020. No 3. Article 9. DOI: https://doi.org/10.1186/s42400-020-00049-3.

243. Young P. Optimization Among Provably Equivalent Programs // Journal of the ACM. 1977. Vol. 2, No. 4. P. 93–700.

ДОДАТКИ

Додаток А. Акти впровадження дисертаційних досліджень

Міністерство економічного
розвитку і торгівлі України

**ДЕРЖАВНЕ ПІДПРИЄМСТВО
"ХАРКІВСЬКИЙ НАУКОВО-
ДОСЛІДНИЙ ІНСТИТУТ ТЕХНОЛОГІЇ
МАШИНОБУДУВАННЯ"
(ДП "ХНДІТМ")**

Україна, 61016, м. Харків,
вул. Кривоконівська, 30
тел./факс: +38 (057) 372-40-50
e-mail: tehmash@ukr.net
www.tehmash.kharkov.ua

Ministry of Economic
Development and Trade of Ukraine

**STATE ENTERPRISE
"KHARKOV SCIENTIFIC-RESEARCH
INSTITUTE OF MECHANICAL
ENGINEERING TECHNOLOGY"**

30, Krivokonevskaya Str.,
Kharkiv, 61016, Ukraine
phone/fax: +38 (057) 372-40-50
e-mail: tehmash@ukr.net
www.tehmash.kharkov.ua

Вих. № 7 від "04" липня 2019 року

ЗАТВЕРДЖУЮ:
Директор Державного підприємства
«Харківський науково-дослідний
інститут технології машинобудування»,
д.т.н., доцент
_____ ***В.В. Косенко***

АКТ

впровадження результатів наукових досліджень
Давидова Вячеслава Вадимовича

Комісія Державного підприємства "Харківський науково-дослідний інститут технології машинобудування" у складі:

голова – Добротворський Сергій Семенович, вчений секретар, д.т.н., професор; члени комісії: Кобзєв Олександр Сергійович, начальник науково-технічного відділу, к.т.н., старший науковий співробітник; Свиридов Юрій Митрофанович, начальник відділу, склала даний акт про те, що в ході науково-дослідної роботи використано результати докторської дисертаційної роботи Давидова Вячеслава Вадимовича у вигляді:

- метод перевірки логіко-смислової подібності стандартних послідовних схем програм при верифікації обфускації байт-код орієнтованого програмного коду;
- критерій оцінки якості обфусцированості коду програмних засобів.

Застосування зазначених результатів дозволило зменшити витрати на експлуатацію системи захисту програмного забезпечення на основі методу обфускації до 10%.

Акт впровадження результатів дисертаційної роботи Давидова В.В. обговорено та ухвалено науково-технічною радою Державного підприємства «Харківський науково-дослідний інститут технології машинобудування», протокол № 7 від «_04_» ____07____ 2019 р.

Голова комісії: С.С. Добротворський

Члени комісії: О.С. Кобзев

Ю.М. Свиридов

МІНІСТЕРСТВО РОЗВИТКУ ЕКОНОМІКИ, ТОРГІВЛІ
ТА СІЛЬСЬКОГО ГОСПОДАРСТВА УКРАЇНИ

ДЕРЖАВНЕ ПІДПРИЄМСТВО
"ПІВДЕННИЙ ДЕРЖАВНИЙ ПРОЕКТНО-
КОНСТРУКТОРСЬКИЙ ТА НАУКОВО-ДОСЛІДНИЙ
ІНСТИТУТ АВІАЦІЙНОЇ ПРОМИСЛОВОСТІ"
(ДП "ПІВДЕНДІПРОНДІАВІАПРОМ")
вул. Сумська, 130а, м. Харків,
61023, УКРАЇНА
Тел: (057) 704-10-47
E-mail: yuzhgap@i.ua
www.yuzhgap.com.ua
код ЄДРПОУ 14307759

ЮЖГИПРОНИИАВИАПРОМ

MINISTRY OF ECONOMIC DEVELOPMENT
AND TRADE OF UKRAINE

STATE ENTERPRISE "SOUTHERN NATIONA
DESIGN & RESEARCH INSTITUTE
OF AEROSPACE INDUSTRIES"
(SE YUZHGIPRONIIAVIAPROM)

130a Sumska St. Kharkiv City
61023 UKRAINE
Phone: + 38 (057) 704-10-47
E-mail:yuzhgap@i.ua
www.yuzhgap.com.ua

_____ № _____
на № _____ від 05.07.2019

ЗАТВЕРДЖУЮ:
Директор Державного підприємства «Південний державний проектно-конструкторський та науково-дослідний інститут авіаційної промисловості»,
канд. техн. наук
_____ **Р.В. АРТЮХ**

УКРАЇНА ДЕРЖАВНЕ ПІДПРИЄМСТВО "ПІВДЕННИЙ ДЕРЖАВНИЙ ПРОЕКТНО-КОНСТРУКТОРСЬКИЙ ТА НАУКОВО-ДОСЛІДНИЙ ІНСТИТУТ АВІАЦІЙНОЇ ПРОМИСЛОВОСТІ" №14307759 МІНІСТЕРСТВО РОЗВИТКУ ЕКОНОМІКИ ТОРГІВЛІ

АКТ

впровадження результатів наукових досліджень Давидова Вячеслава Вадимовича, отриманих під час виконання дисертаційної роботи

Комісія Державного підприємства "Південний державний проектно-конструкторський та науково-дослідний інститут авіаційної промисловості" у складі:

Голова комісії: Чмихун Костянтин Євгенійович – головний інженер;

Члени комісії: Чорний Віктор Олексійович – заступник директора з виробництва; Топольницька Олена Олександрівна – головний архітектор проекту, констатує, що в ході виконання науково-дослідної роботи за темою «Проектування аеродромів, вертодромів, аеропортів і об'єктів їх інфраструктури» використано результати дисертаційних досліджень отриманих Давидовим В.В., а саме

- методу обфускації програмних модулів для підвищення безпеки байт-код орієнтованого програмного забезпечення на основі розробленої GERT-моделі, що дозволило підвищити час зламу коду до 9%;

- критерію якості обфускації програмних модулів; це дозволило підвищити точність оцінки безпеки програмного забезпечення на 15%.

В результаті впровадження наукових положень та результатів дисертації Давидова В.В. отримано науково-технічний ефект, пов'язаний з забезпеченням безпеки розробляємого програмного коду. Зокрема, запропонований у роботі Давидовим В.В. метод обфускації програмного забезпечення на основі розробленої GERT-моделі та критерій оцінки безпеки програмних засобів.

Акт складений для пред'явлення до спеціалізованої Вченої ради із захисту дисертацій і не є підставою для фінансових розрахунків.

Голова комісії: Чмихун К.Є.

Члени комісії: Чорний В.О.

Топольницька О.О.

ЗАТВЕРДЖУЮ
ГЕНЕРАЛЬНИЙ ДИРЕКТОР
КОМПАНІЇ “LINEUP”
КІБІТКІН Т.Р.
«18» липня 2022 р.

АКТ

впровадження результатів наукових досліджень дисертаційної роботи Давидова Вячеслава Вадимовича

Даним актом засвідчую, що результати наукових досліджень дисертаційної роботи Давидова В.В., поданої на здобуття наукового ступеню доктора технічних наук за темою «Моделі та методи підвищення безпеки байт-код орієнтованого програмного забезпечення в умовах кібератак», були використані в практичній діяльності фірми “LineUp”, а саме:

Математична модель безпечного переходу та кодування ліцензійних ідентифікаторів в GERT-мережах, що використовуються в якості графа керуючої логіки програмного продукту в залежності від ідентифікаційного або серійного номера комп’ютерної системи кінцевого користувача.

Практичне використання отриманих результатів дозволило спроектувати комплексну системи безпеки програмного забезпечення для захисту авторських прав.

Заступник генерального директора Водяницька А.О

товариство з обмеженою відповідальністю

«НІКС СОЛЮШЕНС ЛТД»
NIX Solutions Ltd

Україна, м. Харків, 61105, вул Каразіна, б.2,
Тел. (057) 784-06-00, факс (057)784-06-16

Код ЄДРПОУ 37365205

Вих. № _
від «7» квітня 2021 **р.**

АКТ

впровадження результатів наукових досліджень дисертаційної роботи

Давидова В'ячеслава Вадимовича

Даним актом засвідчую, що результати наукових досліджень дисертаційної роботи Давидова В.В., поданої на здобуття наукового ступеню доктора технічних наук, були використані в практичній діяльності фірми "Нікс Солюшенс ЛТД", а саме:

1. Математична модель безпечного переходу в GERT-мережах, що використовуються в якості графа керуючої логіки програмного продукту в залежності від ідентифікаційного або серійного номера комп'ютерної системи кінцевого користувача.
2. Метод формування цифрового ідентифікатора програмного забезпечення для захисту авторських прав шляхом введення та удосконалення модулів менеджерів ліцензій, контроля цілісності та ідентифікації.

Синтез методу генерації цифрового ідентифікатора дозволив реалізувати відповідні програмні засоби, що дозволяє підвищити захищеність розробляємих програмних засобів від неліцензованого копіювання.

Директор Шальнєв В.В.

ЗАТВЕРДЖУЮ
Проректор з науково-педагогічної роботи Національного технічного університету «ХПІ»
_____ Юрій ЗАЙЦЕВ
«_24_» _березня_ _2021_ р

АКТ

про впровадження результатів дисертаційної роботи доцента кафедри обчислювальної техніки та програмування НТУ «ХПІ» Давидова Вячеслава Вадимовича

Комісія у складі декана факультету «Комп'ютерні та інформаційні технології», к.е.н., проф. Главчева М.І. (голова комісії), професора кафедри обчислювальної техніки та програмування, к.т.н., проф. Заполовського М.Й. та професора кафедри обчислювальної техніки та програмування, д.т.н., проф. Кучука Г.А., розглянула стан використання матеріалів дисертаційної роботи Давидова В.В. при підготовці бакалаврів та магістрів за спеціальностями 123 «Комп'ютерна інженерія» та 125 «Кібербезпека».

Комісія прийшла до висновку:

1. Матеріали дисертаційної роботи Давидова В.В. використані при викладанні учбових дисциплін: «Програмування», «Сучасні технології безпечного програмування», «Основи безпечного програмування», для студентів спеціальностей 123 «Комп'ютерна інженерія» та 125 «Кібербезпека».

2. Розроблені автором математичні моделі та процедури обфускації і генерації ліцензійних ключів використовуються в курсових, дипломних та наукових роботах студентів спеціальностей 123 «Комп'ютерна інженерія» та 125 «Кібербезпека».

Голова комісії: декан факультету «КІТ», к.е.н., проф. Главчев М.І.

Члени комісії: проф. каф. «ОТП», к.т.н., проф. Заполовський М.Й.

проф. каф. «ОТП», д.т.н., проф. Кучук Г.А.

Додаток Б. Лістинг коду підпрограми розрахунку показників якості програмного забезпечення

ConditionalAnalysis.cs

```
public static class ConditionalAnalysis
{
public static int GetIfStatementCount(SyntaxNode tree)
{
    return tree.DescendantNodesAndSelf().Where(s => s.IsKind(SyntaxKind.IfStatement)).Count();
}

public static int GetElseStatementCount(SyntaxNode tree)
{
    return tree.DescendantNodesAndSelf().Where(s => s.IsKind(SyntaxKind.ElseClause)).Count();
}

public static int GetTernaryOperatorCount(SyntaxNode tree)
{
    return tree.DescendantNodesAndSelf().Where(s => s.IsKind(SyntaxKind.ConditionalExpression)).Count();
}

public static int GetSwitchOperatorCount(SyntaxNode tree)
{
```

```
            return tree.DescendantNodesAndSelf().Where(s => s.IsKind(SyntaxKind.SwitchStatement)).Count();
        }

        public static int GetGoToOperatorCount(SyntaxNode tree)
        {
            return tree.DescendantNodesAndSelf().Where(s => s.IsKind(SyntaxKind.GotoStatement)).Count();
        }

        public static int GetLabelsOperatorCount(SyntaxNode tree)
        {
            return tree.DescendantNodesAndSelf().Where(s => s.IsKind(SyntaxKind.LabeledStatement)).Count();
        }

        public static int GetWhereCount(SyntaxNode tree)
        {
            var linq = tree.DescendantNodesAndSelf().Where(s => s.IsKind(SyntaxKind.WhereClause)).Count();
            var extensionLinq = tree.DescendantNodesAndSelf().Where(s => s.IsKind(SyntaxKind.SimpleLambdaExpression) || s.IsKind(SyntaxKind.ParenthesizedLambdaExpression));
            foreach (var expression in extensionLinq)
            {
                InvocationExpressionSyntax parent = null;
                SyntaxNode node = expression;
                while (true)
                {
```

```
                    if (node.Parent == null) break;
                    node = node.Parent;
                    if (node is InvocationExpressionSyntax ies)
                    {
                        parent = ies;
                        break;
                    }
                }
                if (parent != null)
                {
                    if (parent.Expression.ToString().EndsWith(".Where"))
                        linq++;
                }

            }
            return linq;
        }
```

FitzpatrickAnalysis.cs

```
        public static class FitzpatrickAnalysis
        {
            public static Tuple<int, int, int> GetFitzpatrickMetrick(SyntaxNode tree)
            {
                var a = GetAssignmentCount(tree);
                var b = GetFunctionCallsCount(tree);
                var c = ConditionalAnalysis.GetIfStatementCount(tree) + ConditionalAnalysis.GetTernaryOperatorCount(tree);
```

```
            return new Tuple<int, int, int>(a, b, c);
        }

        private static int GetAssignmentCount(SyntaxNode tree)
        {
            return tree
                .DescendantNodesAndSelf()
                .Where(s =>
                    s.IsKind(SyntaxKind.AddAssignmentExpression)
                    || s.IsKind(SyntaxKind.AndAssignmentExpression)
                    ||
s.IsKind(SyntaxKind.CoalesceAssignmentExpression)
                    ||
s.IsKind(SyntaxKind.DivideAssignmentExpression)
                    ||
s.IsKind(SyntaxKind.ExclusiveOrAssignmentExpression)
                    ||
s.IsKind(SyntaxKind.LeftShiftAssignmentExpression)
                    ||
s.IsKind(SyntaxKind.ModuloAssignmentExpression)
                    ||
s.IsKind(SyntaxKind.MultiplyAssignmentExpression)
                    || s.IsKind(SyntaxKind.OrAssignmentExpression)
                    ||
s.IsKind(SyntaxKind.RightShiftAssignmentExpression)
                    ||
s.IsKind(SyntaxKind.SimpleAssignmentExpression)
                    ||
s.IsKind(SyntaxKind.SubtractAssignmentExpression)
```

```
                    )
                .Count();
        }

        private static int GetFunctionCallsCount(SyntaxNode tree)
        {
            return tree.DescendantNodesAndSelf().Where(s => s.IsKind(SyntaxKind.InvocationExpression)).Count();
        }
    }
```

FunctionAnalysis.cs

```
    class FunctionAnalysis
    {
        public static int GetParasiticCalls(SyntaxNode tree)
        {
            var constructors = tree.DescendantNodes().OfType<ObjectCreationExpressionSyntax>().Select(n=> n.Type.ToString());

            var invocationsGetAndSet = GetInvocationsNames(tree).Where(i=>i.StartsWith("get") || i.StartsWith("set"));

            return constructors.Count() + invocationsGetAndSet.Count();
        }
```

```
        public static int GetTotalFunctionCalls(List<Tuple<string, int, double>> tree)
        {
            return tree.Count();
        }

        public static int GetTotalFunctionDistinctCalls(List<Tuple<string, int, double>> tree)
        {
            return tree. Select(s=>s.Item1).Distinct().Count();
        }

        public static double GetMeanInvocationDegree(List<Tuple<string, int, double>> tree)
        {
            if (tree != null && tree.Any())
                return tree.Average(e => e.Item3);
            return 0;
        }

        public static double GetMedianInvocationDegree(List<Tuple<string, int, double>> tree)
        {
            var ordered = tree.OrderByDescending(e => e.Item2).ToArray();
            if (ordered.Length == 0)
                return 0;
            if (ordered.Length == 1)
                return ordered[0].Item3;
```

```
            if (ordered.Length % 2 == 1)
                return ordered[ordered.Length / 2].Item3;
            else
                return (ordered[ordered.Length / 2].Item3 +
ordered[ordered.Length / 2].Item3) / 2;
        }

        public static double[] GetModeInvocationDegree(List<Tuple<string,
int, double>> degree)
        {
            if (degree == null || !degree.Any()) return null;
            var degreeWithPopularity = degree.Select(e => new { item =
e.Item3, popularity = degree.Where(d => d.Item3 == e.Item3).Count() });
            var maxPopularity = degreeWithPopularity.Max(e =>
e.popularity);

            return degreeWithPopularity.Where(dp => dp.popularity ==
maxPopularity).Select(dp => dp.item).Distinct().ToArray();
        }

        public static List<string> GetInvocationsNames(SyntaxNode tree)
        {
            var invokes =
tree.DescendantNodes().OfType<InvocationExpressionSyntax>();
            return invokes.Select(e =>
                e.Expression is IdentifierNameSyntax i
                    ? i.Identifier.ValueText
                    : e.Expression is MemberBindingExpressionSyntax
```

```
                                    ? ((MemberBindingExpressionSyntax)
e.Expression).Name.Identifier.ValueText
                                    : e.Expression is
MemberAccessExpressionSyntax
                                        ? ((MemberAccessExpressionSyntax)
e.Expression).Name.Identifier.ValueText
                                        : ((ExpressionSyntax)
e.Expression).ToString()).ToList();
            }

            public static List<Tuple<string, int, double>>
GetFunctionsInvocationDegree(SyntaxNode tree)
            {
                var invocationNames = GetInvocationsNames(tree);
                double totalInvocations = invocationNames.Count;
                return invocationNames.GroupBy(g => g).Select(g => new
Tuple<string, int, double>(g.Key, g.Count(), g.Count() /
totalInvocations)).ToList();
            }
        }
```

HalsteadAnalysis.cs

```
        class HalsteadAnalysis
        {
            public static List<string> GetOperands(SyntaxNode node)
            {
                return node
                    .DescendantNodes()
```

```
                    .Where(s => s.GetType().GetProperty("OperatorToken")
!= null)
                    .Select(s => s.GetType().GetProperty("Operand") != null
? new[] { s.GetType().GetProperty("Operand").GetValue(s).ToString() } :
                        s.GetType().GetProperty("Left") != null &&
s.GetType().GetProperty("Right") != null ? new[] {
s.GetType().GetProperty("Left").GetValue(s).ToString(),
s.GetType().GetProperty("Right").GetValue(s).ToString() } :
                        new[] { string.Empty }).SelectMany(s =>
s.Select(z => z))
                    .Where(s => s != string.Empty)
                    .ToList();


        }

        public static List<string> GetOperators(SyntaxNode node)
        {
            return node
                .DescendantNodes().Where(s =>
s.GetType().GetProperty("OperatorToken") != null).Select(s =>
s.GetType().GetProperty("OperatorToken").GetValue(s).ToString()).ToList();
        }

        public static void GetHalsteadMetrics(double n1, double n2, double
N1, double N2, out double nProgramDict, out double N_Programlength, out
double Nt_ProgramLengthTheoretical, out double V_ProgramCapacity, out double
Lt_ProgrammingLevel, out double Ec_DifficultyUnderstandingProgram, out
```

```
double D_CodingComplexity, out double I_InformationContent, out double Et_EvaluationOfIntellectualEffortRequired)
        {
            nProgramDict = n1 + n2;

            N_Programlength = N1 + N2;
            Nt_ProgramLengthTheoretical = n1 * Math.Log(2, n1) + n2 * Math.Log(2, n2);

            V_ProgramCapacity = N_Programlength * Math.Log(2, nProgramDict);

            //var Vt =
            Lt_ProgrammingLevel = (2 * n2 / n1 * N2);
            Ec_DifficultyUnderstandingProgram = V_ProgramCapacity / (Lt_ProgrammingLevel * 2);

            D_CodingComplexity = 1 / Lt_ProgrammingLevel;
            //double I = VProgramCapacity/D

            I_InformationContent = Lt_ProgrammingLevel * V_ProgramCapacity;

            Et_EvaluationOfIntellectualEffortRequired = N_Programlength * Math.Log(2, nProgramDict / Lt_ProgrammingLevel);

        }
```

```
        public static void GetHalsteadVariables (SyntaxNode node, out double n1, out double n2, out double N1, out double N2)
        {
            var operators = GetOperators(node);
            var operands = GetOperands(node);

            n1 = operators.Distinct().Count();
            n2 = operands.Distinct().Count();
            N1 = operators.Count();
            N2 = operands.Count();
        }
    }
```

LinesOfCodeAnalysis.cs

```
    public static class LinesOfCodeAnalysis
    {
        public static int GetNumberOfPhysicalLines(string code, bool skipEmpty = true, bool removeComments = true)
        {
            var sso = skipEmpty ? StringSplitOptions.RemoveEmptyEntries : StringSplitOptions.None;
            return code.Split(new[] { "\r\n", "\r", "\n" }, sso).Length;
        }
    }
```

LoopAnalysis.cs

```
    public static class LoopAnalysis
```

```
    {
        public static int GetForLoopCount(SyntaxNode tree)
        {
            return tree.DescendantNodesAndSelf().Where(s =>
s.IsKind(SyntaxKind.ForStatement)).Count();
        }

        public static int GetWhileLoopCount(SyntaxNode tree)
        {
            return tree.DescendantNodesAndSelf().Where(s =>
s.IsKind(SyntaxKind.WhileStatement)).Count();
        }

        public static int GetDoWhileLoopCount(SyntaxNode tree)
        {
            return tree.DescendantNodesAndSelf().Where(s =>
s.IsKind(SyntaxKind.DoStatement)).Count();
        }

        public static int GetForEachLoopCount(SyntaxNode tree)
        {
            return tree.DescendantNodesAndSelf().Where(s =>
s.IsKind(SyntaxKind.ForEachStatement)).Count();
        }
    }
```

NestingAnalysis.cs

```
    public static class NestingAnalysis
```

```
    {
        public static int GetNesting(SyntaxNode node, int nesting = 0)
        {
            var nestingList = new List<int>();
            var childsWithChilds = node.ChildNodes().ToList().Where(sn
=> sn.ChildNodes().Any());
            if (childsWithChilds.Any())
            {
                if (node.IsKind(SyntaxKind.NamespaceDeclaration)
                    || node.IsKind(SyntaxKind.ClassDeclaration)
                    ||
node.IsKind(SyntaxKind.ParenthesizedLambdaExpression)
                    ||
node.IsKind(SyntaxKind.SimpleLambdaExpression)
                    || node.IsKind(SyntaxKind.Block)
                    )
                    nesting++;
                var tmp = nesting;
                foreach (var child in childsWithChilds)
                {
                    nesting = tmp;
                    var res = GetNesting(child, nesting);
                    nestingList.Add(res);
                }
            }
            return nestingList.Any() ? nestingList.Max() : nesting;
        }
    }
```

SpenVariablesAnalysis.cs

```
public class SpenVariablesAnalysis
{
    public static Dictionary<string, int> GetSpenMetric(SyntaxNode tree)
    {
        var dict = new Dictionary<string, int>();
        var variableIdentifiers = tree
            .DescendantNodesAndSelf()
            .Where(s => s.IsKind(SyntaxKind.VariableDeclaration))
            .SelectMany(
                w => ((VariableDeclarationSyntax)w)
                    .Variables
            ).ToList();
        var collectionOfNodesIds = tree.DescendantNodesAndSelf().Where(n => n.IsKind(SyntaxKind.IdentifierName)).Select(s => s.GetParentName("") + s.ToString()).ToArray();
        foreach (var variableNode in variableIdentifiers)
        {
            var variable = variableNode.GetParentName("") + variableNode.Identifier.ToString();
            if (!dict.ContainsKey(variable)) dict[variable] = 0;
            dict[variable] += collectionOfNodesIds.Count(nid => nid.ToString().Equals(variable));
        }

        return dict;
    }
```

```
        public static Dictionary<string, int> GetSpenMetric2(SyntaxNode tree)
        {
            var dict = new Dictionary<string, int>();

            //In Methods
            foreach(var m in tree.DescendantNodes().OfType<MethodDeclarationSyntax>())
            {
                var variableIdentifiers = m
                    .DescendantNodes()
                    .Where(s => s.IsKind(SyntaxKind.VariableDeclaration))
                    .SelectMany(
                        w => ((VariableDeclarationSyntax)w).Variables)
                    .Select(s=>s.GetParentName("") + s.Identifier.ToString())
                    .ToList();
                var collectionOfNodesIds = tree.DescendantNodesAndSelf().Where(n => n.IsKind(SyntaxKind.IdentifierName)).Select(s => s.GetParentName("") + s.ToString()).ToArray();
                foreach (var id in variableIdentifiers)
                {
                    var count = collectionOfNodesIds.Count(nid => nid.ToString().Equals(id));
                    //if (count != 0)
```

```
                {
                    if (!dict.ContainsKey(id)) dict[id] = 0;
                    dict[id] += collectionOfNodesIds.Count(nid
=> nid.ToString().Equals(id));
                }
            }
        }


        //In Classes
        foreach (var m in
tree.DescendantNodes().OfType<ClassDeclarationSyntax>())
        {
            var variableIdentifiers = m
                .ChildNodes()
                .OfType<FieldDeclarationSyntax>()
                .Where(f => f.Modifiers.Any(mod =>
mod.ValueText.Equals("private")))
                .SelectMany(w => w.Declaration.Variables)
                .Select(s=>s.Identifier)
                .ToList();
            var collectionOfNodesIds =
tree.DescendantNodesAndSelf().Where(n =>
n.IsKind(SyntaxKind.IdentifierName)).Select(s => s.GetParentName("") +
s.ToString()).ToArray();
            foreach (var id in collectionOfNodesIds)
            {
                var key = variableIdentifiers.FirstOrDefault(v =>
id.EndsWith($".{v.ToString()}")).ToString();
```

```
                        if (!string.IsNullOrEmpty(key))
                        {
                            if (!dict.ContainsKey(key)) dict[key] = 0;
                            dict[key]++;
                        }
                    }
                }

                return dict;
            }

            public static Dictionary<string,List<string>>
GetHierarchySctructureMetric(SyntaxNode node)
            {
                var list = string.Join("\r\n",
node.DescendantNodes().OfType<ClassDeclarationSyntax>().Select(c =>
c.GetParentName("") + c.GetNodeName()).ToList());
                return
node.DescendantNodes().OfType<ClassDeclarationSyntax>().ToDictionary(c =>
c.GetParentName("") + c.GetNodeName(), c => c.BaseList?.Types.Select(t =>
t.Type.ToString()).ToList());
            }

            public static Dictionary<string,List<SyntaxNode>>
GetAllVariables(SyntaxNode node)
            {
                return node.DescendantNodes()
                    .OfType<ClassDeclarationSyntax>()
                    .ToDictionary(
```

```
                                    s=>s.GetParentName("")+s.GetNodeName(),
                                    s=>s.DescendantNodes().Where(n =>
n.IsKind(SyntaxKind.IdentifierName))
                                .ToList());
            }

            public static Dictionary<string,List<string>>
GetClassDeclarations(SyntaxNode tree)
            {
                  return
tree.DescendantNodes().OfType<ClassDeclarationSyntax>().ToDictionary(c=>
c.GetParentName("") + c.GetNodeName(),
                                c=>c.ChildNodes()
                                      .OfType<FieldDeclarationSyntax>()

      .Where(s=>s.Modifiers.Any(m=>m.ValueText.Equals("public")||
m.ValueText.Equals("protected")))
                                      .SelectMany(w => w.Declaration.Variables)
                                      .Select(s => s.Identifier.ToString())
                                      .ToList());

                  }

            public static List<string> GetConstAndStaticVariables(SyntaxNode
tree)
            {
                  return
tree.DescendantNodes().OfType<ClassDeclarationSyntax>().SelectMany(
                                c => c.ChildNodes()
```

```
                .OfType<FieldDeclarationSyntax>()
                .Where(s => s.Modifiers.Any(m => m.ValueText.Equals("public")) &&
s.Modifiers.Any(m=>m.ValueText.Equals("const") || m.ValueText.Equals("static")))
                .SelectMany(w => w.Declaration.Variables)
                .Select(s => s.GetParentName("") + s.GetNodeName()))
                .ToList();

        }
    }
```

CsvUtility.cs

```
    public static class CsvUtility
    {
        public static string GetCsvHeader(string sep = "|")
        {
            return $"MetricIdentifier{sep}MetricType{sep}Path{sep}" +
                $"PhysicalLoc{sep}" +
                $"MaxNesting{sep}" +

$"HalsteadTotalDistinctOperators{sep}HalsteadTotalDistinctOperands{sep}HalsteadTotalAllOperators{sep}HalsteadTotalAllOperands{sep}ProgramDictionary{sep}ProgramLength{sep}TheoreticalProgramLength{sep}ProgramCapacity{sep}ProgrammingLevel{sep}DifficultyUnderstandingProgram{sep}CodingComplexity{sep}InformationContent{sep}EvaluationOfIntellectualEffortRequired{sep}" +
```

```
$"TotalIfCounter{sep}TotalElseCounter{sep}TotalTernaryOperatorCounter{sep}
TotalWhereCounter{sep}TotalSwitchCounter{sep}TotalGoToCounter{sep}TotalL
abelsCounter{sep}" +

$"TotalForCounter{sep}TotalForeachCounter{sep}TotalWhileCounter{sep}Total
DoWhileCounter{sep}" +

$"FitzpatrickCounters_A{sep}FitzpatrickCounters_B{sep}_FitzpatrickCounters_C
{sep}FitzpatrickCountersCalculated{sep}" +
                                $"SpenMetricAvg{sep}" +

$"TotalFunctionCalls{sep}TotalFunctionDistinctCalls{sep}TotalFunctionParasitic
CallsMetric{sep}InvocationDegreeMin{sep}InvocationDegreeMax{sep}MeanInv
ocationDegree{sep}MedianInvocationDegree{sep}ModeInvocationDegree_min{s
ep}ModeInvocationDegree_max{sep}ModeInvocationDegree_avg{Environment.
NewLine}";
                //  sb.Append($"{Environment.NewLine}");
            }
            public static string GetSourceCodeMetricsCsv(SourceCodeMetrics
scm, string sep = "|")
            {
                var sb = new StringBuilder();

                sb.Append($"{scm.Identifier}{sep}");
                sb.Append($"{scm.IdentifierType}{sep}");
                sb.Append($"{scm.Path}{sep}");

                sb.Append($"{scm.PhysicalLoc}{sep}");
```

```
                sb.Append($"{scm.NestingMetric}{sep}");


        HalsteadAnalysis.GetHalsteadMetrics(scm.HalsteadDistinctOperators,
scm.HalsteadDistinctOperands,
                        scm.HalsteadAllOperators, scm.HalsteadAllOperands,
out double nProgramDict, out double N_Programlength,
                        out double Nt_ProgramLengthTheoretical, out double
V_ProgramCapacity, out double Lt_ProgrammingLevel,
                        out double Ec_DifficultyUnderstandingProgram, out
double D_CodingComplexity,
                        out double I_InformationContent, out double
Et_EvaluationOfIntellectualEffortRequired);

                sb.Append($"{scm.HalsteadDistinctOperators}{sep}");
                sb.Append($"{scm.HalsteadDistinctOperands}{sep}");
                sb.Append($"{scm.HalsteadAllOperators}{sep}");
                sb.Append($"{scm.HalsteadAllOperands}{sep}");
                sb.Append($"{nProgramDict}{sep}");
                sb.Append($"{N_Programlength}{sep}");
                sb.Append($"{Nt_ProgramLengthTheoretical}{sep}");
                sb.Append($"{V_ProgramCapacity}{sep}");
                sb.Append($"{Lt_ProgrammingLevel}{sep}");
                sb.Append($"{Ec_DifficultyUnderstandingProgram}{sep}");
                sb.Append($"{D_CodingComplexity}{sep}");
                sb.Append($"{I_InformationContent}{sep}");

        sb.Append($"{Et_EvaluationOfIntellectualEffortRequired}{sep}");
```

```
                sb.Append($"{scm.IfCounter}{sep}");
                sb.Append($"{scm.ElseCounter}{sep}");
                sb.Append($"{scm.TernaryOperatorCounter}{sep}");
                sb.Append($"{scm.WhereCount}{sep}");
                sb.Append($"{scm.SwitchCount}{sep}");
                sb.Append($"{scm.GoToCount}{sep}");
                sb.Append($"{scm.LabelsCount}{sep}");

                sb.Append($"{scm.ForCounter}{sep}");
                sb.Append($"{scm.ForeachCounter}{sep}");
                sb.Append($"{scm.WhileCounter}{sep}");
                sb.Append($"{scm.DoWhileCounter}{sep}");


        sb.Append($"{scm.FitzpatrickCounters.Item1}{sep}{scm.FitzpatrickCounte
rs.Item2}{sep}{scm.FitzpatrickCounters.Item3}{sep}");
                sb.Append($"{scm.FitzpatrickCounterCalculated}{sep}");

                var spen = scm.SpenMetric != null && scm.SpenMetric.Any()
? scm.SpenMetric.Average(c => c.Value) : 0;
                sb.Append($"{spen}{sep}");

                sb.Append($"{scm.TotalFunctionCalls}{sep}");
                sb.Append($"{scm.TotalFunctionDistinctCalls}{sep}");
                sb.Append($"{scm.FunctionParasiticCallsMetric}{sep}");
                sb.Append($"{scm.InvocationDegreeMin}{sep}");
                sb.Append($"{scm.InvocationDegreeMax}{sep}");
                sb.Append($"{scm.MeanInvocationDegree}{sep}");
```

```
            sb.Append($"{scm.MedianInvocationDegree}{sep}");
            sb.Append($"{scm.ModeInvocationDegree?.Min()}{sep}");
            sb.Append($"{scm.ModeInvocationDegree?.Max()}{sep}");
            sb.Append($"{scm.ModeInvocationDegree?.Average()}");
            sb.Append($"{Environment.NewLine}");
            return sb.ToString();
        }

        public static string GetCsv(this IEnumerable<SourceCodeMetrics>
metrics)
        {
            var sb = new StringBuilder();
            foreach(var metric in metrics)
            {
                sb.Append(GetSourceCodeMetricsCsv(metric));
            }
            return sb.ToString();
        }
    }
```

SourceCodeMetrics.cs

```
    public class SourceCodeMetrics
    {
        public string Identifier { get; set; }
        public string Path { get; set; }
        public IdentifierTypeEnum IdentifierType { get; set; }
        public int PhysicalLoc { get; set; }
        public int LogicalLoc { get; set; }
```

```
public int NestingMetric { get; set; }
public double HalsteadAllOperators { get; set; }
public double HalsteadAllOperands { get; set; }
public double HalsteadDistinctOperators { get; set; }
public double HalsteadDistinctOperands { get; set; }
public double ProgramDictionary { get; set; }
public double ProgramLength { get; set; }
public double ProgramLengthTheoretical { get; set; }
public double ProgramCapacity { get; set; }
public double ProgrammingLevel { get; set; }
public double DifficultyUnderstandingProgram { get; set; }
public double CodingComplexity { get; set; }
public double InformationContent { get; set; }
public double EvaluationOfIntellectualEffortRequired { get; set; }
public int IfCounter { get; set; }
public int ElseCounter { get; set; }
public int TernaryOperatorCounter { get; set; }
public int WhereCount { get; set; }
public int SwitchCount { get; set; }
public int GoToCount { get; set; }
public int LabelsCount { get; set; }
public int ForCounter { get; set; }
public int ForeachCounter { get; set; }
public int WhileCounter { get; set; }
public int DoWhileCounter { get; set; }
public Tuple<int, int, int> FitzpatrickCounters { get; set; }
public int FitzpatrickCounterCalculated
{
        get
```

```
                {
                    return Math.Abs(FitzpatrickCounters.Item1 *
FitzpatrickCounters.Item1
                                    + FitzpatrickCounters.Item2 *
FitzpatrickCounters.Item2
                                    + FitzpatrickCounters.Item3 *
FitzpatrickCounters.Item3);
                }
            }

            public Dictionary<string, int> SpenMetric { get; set; }
            public Dictionary<string, List<string>> HierarchySctructureMetric {
get; set; }
            public Dictionary<string, List<string>> ClassAndDeclarations { get;
set; }
            public Dictionary<string, List<SyntaxNode>> AllVariables { get; set;
}
            public List<string> ConstAndStaticDeclarations { get; set; }
            public List<Tuple<string, int, double>> FunctionCallsMetric { get;
set; }
            public int FunctionParasiticCallsMetric { get; set; }
            public int TotalFunctionCalls
            {
                get { return
FunctionAnalysis.GetTotalFunctionCalls(FunctionCallsMetric); }
            }

            public int TotalFunctionDistinctCalls
            {
```

```
            get { return
FunctionAnalysis.GetTotalFunctionDistinctCalls(FunctionCallsMetric); }
        }

        public double InvocationDegreeMin
        {
            get
            {
                return FunctionCallsMetric != null &&
FunctionCallsMetric.Any()
                    ? FunctionCallsMetric.Select(mc =>
mc.Item3).Min()
                    : 0;
            }
        }

        public double InvocationDegreeMax
        {
            get
            {
                return FunctionCallsMetric != null &&
FunctionCallsMetric.Any()
                    ? FunctionCallsMetric.Select(mc =>
mc.Item3).Max()
                    : 0;
            }
        }

        public double MeanInvocationDegree
```

```
            {
                get { return FunctionAnalysis.GetMeanInvocationDegree(FunctionCallsMetric); }
            }

            public double MedianInvocationDegree
            {
                get { return FunctionAnalysis.GetMedianInvocationDegree(FunctionCallsMetric); }
            }

            public double[] ModeInvocationDegree
            {
                get { return FunctionAnalysis.GetModeInvocationDegree(FunctionCallsMetric); }
            }
        }
```

SyntaxNodesHelpers.cs

```
        static class SyntaxNodesHelpers
        {
            public static string GetParentName(this SyntaxNode node, string parentName)
            {
                if (node.Parent == null)
                    return null;
                if (node.Parent.IsKind(SyntaxKind.CompilationUnit))
```

```
                return parentName;
            else
            {
                if (node.Parent.IsKind(SyntaxKind.MethodDeclaration) ||
node.Parent.IsKind(SyntaxKind.ClassDeclaration)                    ||
node.Parent.IsKind(SyntaxKind.NamespaceDeclaration))
                {
                    var parent = GetNodeName(node.Parent);
                    parentName = parent + "." + parentName;
                }
                parentName          =          GetParentName(node.Parent,
parentName);
            }
            return parentName;
        }


        public static string GetNodeName(this SyntaxNode node)
        {
            var      name      =      node.ChildTokens().FirstOrDefault(t      =>
t.IsKind(SyntaxKind.IdentifierToken)).ToString();
            if (string.IsNullOrWhiteSpace(name))
                name = node.ChildNodes().FirstOrDefault()?.ToString();
            return name;
        }

        public static SyntaxNode GetChildNodeByName(this SyntaxNode
node, string nodeFullName, string nodePartName)
        {
```

```
                if (string.IsNullOrEmpty(nodePartName))
                {
                        if (string.IsNullOrWhiteSpace(nodeFullName))
                        {
                                return node;
                        }
                        else
                        {
                                var dotIndex = nodeFullName.IndexOf('.');
                                if (dotIndex != -1)
                                {
                                        nodePartName = nodeFullName.Substring(0,
dotIndex);
                                        nodeFullName                                =
nodeFullName.Substring(dotIndex + 1);
                                }
                                else
                                {
                                        nodePartName = nodeFullName;
                                        nodeFullName = "";

                                        if (node is VariableDeclarationSyntax vds
&& vds.Variables.Any(v => v.Identifier.Equals(nodePartName)))
                                        {
                                                return   vds.Variables.FirstOrDefault(v
=> v.Identifier.Equals(nodePartName));
                                        }
                                }
                        }
```

```
            }
            var nodes = node.ChildNodes().ToList();
            nodes.Add(node);
            var child = nodes.FirstOrDefault(cn =>
                    GetNodeName(cn).Equals(nodePartName)
                );
            if (child != null)
            {
                return GetChildNodeByName(child, nodeFullName, "");
            }
            else
            {

                foreach (var cNode in node.ChildNodes())
                {
                    child = GetChildNodeByName(cNode, nodeFullName, nodePartName);
                    if (child != null) return child;
                }
            }
            return child;
        }

        public static SourceCodeMetrics AggreagateSourceCodeMetrics(this IEnumerable<SourceCodeMetrics> metrics)
        {
            var metric = metrics.FirstOrDefault();

            var classHierarchy = metric.HierarchySctructureMetric;
```

```
var classesWithDeclaration = metric.ClassAndDeclarations;
var classesWithVariables = metric.AllVariables;
var allPublicStaticConstDeclarations = metrics.SelectMany(m => m.ConstAndStaticDeclarations);

foreach (var m in metrics.Skip(1))
{
    foreach (var hierarhyElement in m.HierarchySctructureMetric)
    {
        if (hierarhyElement.Key != null && hierarhyElement.Value != null)
        {
            if (metric.HierarchySctructureMetric.ContainsKey(hierarhyElement.Key))
            {

metric.HierarchySctructureMetric[hierarhyElement.Key].AddRange(hierarhyElement.Value);
            }
            else { metric.HierarchySctructureMetric.Add(hierarhyElement.Key, hierarhyElement.Value); }
        }
    }
    foreach (var classWithDeclaration in m.ClassAndDeclarations)
    {
```

```
                    if (classWithDeclaration.Key != null &&
classWithDeclaration.Value != null)
                    {
                        if
(metric.ClassAndDeclarations.ContainsKey(classWithDeclaration.Key))
                        {

    metric.ClassAndDeclarations[classWithDeclaration.Key].AddRange(classW
ithDeclaration.Value);
                        }
                        else                                    {
metric.ClassAndDeclarations.Add(classWithDeclaration.Key,
classWithDeclaration.Value); }
                    }
                }
                foreach (var classWithVariables in m.AllVariables)
                {
                    if (classWithVariables.Key != null &&
classWithVariables.Value != null)
                    {
                        if
(metric.AllVariables.ContainsKey(classWithVariables.Key))
                        {

    metric.AllVariables[classWithVariables.Key].AddRange(classWithVariable
s.Value);
                        }
                        else                                    {
metric.AllVariables.Add(classWithVariables.Key, classWithVariables.Value); }
```

```
                }
            }
        }

        classHierarchy = classHierarchy
            .ToDictionary(
                c => c.Key.Contains("<") ? c.Key.Substring(0, c.Key.IndexOf("<")) : c.Key,
                c => c.Value?.Distinct().Select(s => s.Contains("<") ? s.Substring(0, s.IndexOf("<")) : s)
                .ToList());



        //public Dictionary<string, List<string>> ClassAndDeclarations { get; set; }
        //public Dictionary<string, List<SyntaxNode>> AllVariables { get; set; }

        var allClasses = classesWithDeclaration.Keys.ToList();


        Dictionary<string, List<string>> classesAndParents = new Dictionary<string, List<string>>();
        var classesWithParents = new List<ClassWithParents>();
        foreach (var longClassName in allClasses)
        {

            var cl = GetAllChildren(classHierarchy, longClassName);
```

```
                    var classWithParents = new ClassWithParents() { ClassName = longClassName };
                    GetClassWithParents(classWithParents, classHierarchy);
                    classesWithParents.Add(classWithParents);

                    var classWithDeclarations = classWithParents.GetDeclarationsFromClassWithParents(classesWithDeclaration);

                    var list = new List<string>();
                    classesAndParents.Add(longClassName, list);

                    var variablesNames = classesWithVariables[longClassName]
                        .Select(v => v.Parent.IsKind(SyntaxKind.SimpleMemberAccessExpression) || v.Parent.IsKind(SyntaxKind.PointerMemberAccessExpression) ? (v.Parent as MemberAccessExpressionSyntax).Expression + "." + v.GetNodeName() : v.GetNodeName())
                        ;
                    var variablesUsage = variablesNames
                        .Where(i => classWithDeclarations.Any(d => d.EndsWith($".{i}"))
                            // || allPublicStaticConstDeclarations!=null && allPublicStaticConstDeclarations.Any(d => !string.IsNullOrEmpty(d) && d.EndsWith(i)))
                        ).Select(i => classWithDeclarations.FirstOrDefault(d => d.EndsWith($".{i}"))!=null
```

```
                                        ? classWithDeclarations.FirstOrDefault(d =>
d.EndsWith($".{i}"))
                                        :
allPublicStaticConstDeclarations.FirstOrDefault(d => d.EndsWith(i)))
                                    .ToList();
                    //              var              variablesPublicUsage              =
variablesNames.Where(i=>allPublicStaticConstDeclarations.Any(d              =>
d.EndsWith(i)))
                    //                                                  .Select(i          =>
allPublicStaticConstDeclarations.FirstOrDefault(d => d.EndsWith(i)))
                    //    .ToList();
                        //.Select(i=>new          {          baseName          =
classWithDeclarations.FirstOrDefault(d => d.EndsWith($".{i.GetNodeName()}")),
originalName = i.GetNodeName() });

                        var              spenMetricForClassUsage              =
variablesUsage.GroupBy(v => v).ToDictionary(v=>v.Key, v => v.Count());
                    //          var          spenMetricForClassPublicUsage          =
variablesPublicUsage.GroupBy(v  =>  v).ToDictionary(v  =>  v.Key,  v  =>
v.Count());

                        var pureClassNameParts = longClassName.Split('.');
                        var shortClassName = pureClassNameParts.Last();
                        var children = metric.HierarchySctructureMetric.Where(c
=> c.Value != null && c.Value.Any() && (c.Value.Contains(longClassName) ||
c.Value.Contains(shortClassName))).ToList();
                    }
```

```
                foreach (var m in metrics.Skip(1))
                {
                    //metric.Path = $"{metric.Path};{m.Path}";

                    metric.PhysicalLoc += m.PhysicalLoc;
                    metric.LogicalLoc += m.LogicalLoc;

                    metric.NestingMetric = metric.NestingMetric > m.NestingMetric ? metric.NestingMetric : m.NestingMetric;

                    metric.HalsteadAllOperators += m.HalsteadAllOperators;
                    metric.HalsteadAllOperands += m.HalsteadAllOperands;
                    metric.HalsteadDistinctOperators += m.HalsteadDistinctOperators;
                    metric.HalsteadDistinctOperands += m.HalsteadDistinctOperands;

                    metric.IfCounter += m.IfCounter;
                    metric.ElseCounter += m.ElseCounter;
                    metric.TernaryOperatorCounter += m.TernaryOperatorCounter;
                    metric.WhereCount += m.WhereCount;
                    metric.SwitchCount += m.SwitchCount;
                    metric.GoToCount += m.GoToCount;
                    metric.LabelsCount += m.LabelsCount;

                    metric.ForCounter += m.ForCounter;
                    metric.ForeachCounter += m.ForeachCounter;
```

```
                    metric.WhileCounter += m.WhileCounter;
                    metric.DoWhileCounter += m.DoWhileCounter;

                    metric.FitzpatrickCounters = new System.Tuple<int, int,
int>(metric.FitzpatrickCounters.Item1 + m.FitzpatrickCounters.Item1,
                        metric.FitzpatrickCounters.Item2 +
m.FitzpatrickCounters.Item2,
                        metric.FitzpatrickCounters.Item3 +
m.FitzpatrickCounters.Item3);

                    foreach (var spen in m.SpenMetric)
                    {
                        if (metric.SpenMetric.ContainsKey(spen.Key))
                        {
                            metric.SpenMetric[spen.Key] +=
spen.Value;
                        }
                        else { metric.SpenMetric.Add(spen.Key,
spen.Value); }
                    }
                    foreach (var hierarhyElement in
m.HierarchySctructureMetric)
                    {
                        if (hierarhyElement.Value != null)
                        {
                            if
(metric.HierarchySctructureMetric.ContainsKey(hierarhyElement.Key))
                            {
```

```
metric.HierarchySctructureMetric[hierarhyElement.Key].AddRange(hierarhyElement.Value);
                                        }
                                        else                                        {
metric.HierarchySctructureMetric.Add(hierarhyElement.Key,
hierarhyElement.Value); }
                                    }
                                }


        metric.FunctionCallsMetric.AddRange(m.FunctionCallsMetric);
                                metric.FunctionParasiticCallsMetric                 +=
m.FunctionParasiticCallsMetric;
                        }

                        double totalInvocations = metric.FunctionCallsMetric.Sum(s =>
s.Item2);
                        metric.FunctionCallsMetric                                  =
metric.FunctionCallsMetric.GroupBy(g => g.Item1).Select(g => new
Tuple<string, int, double>(g.Key, g.Sum(s => s.Item2), g.Sum(s => s.Item2) /
totalInvocations)).ToList();

                        return metric;
                }

                public static void GetClassWithParents(ClassWithParents
classWithParent, Dictionary<string,List<string>> classHierarhy)
                {
```

```
                var alternateKey = classHierarhy.Keys.FirstOrDefault(k =>
k.EndsWith($".{classWithParent.ClassName}"));
                string key = string.Empty;
                if (classHierarhy.ContainsKey(classWithParent.ClassName))
key = classWithParent.ClassName;
                else if (!string.IsNullOrEmpty(alternateKey) &&
classHierarhy.ContainsKey(alternateKey)) key = alternateKey;
                if (key == string.Empty || classHierarhy[key] == null ||
!classHierarhy[key].Any()) return;
                var classParents = classHierarhy[key];
                foreach(var classParent in classParents)
                {
                    if(classWithParent.Parents == null)
                    {
                        classWithParent.Parents = new
List<ClassWithParents>();
                    }
                    var parent = new ClassWithParents() { ClassName =
classParent };
                    classWithParent.Parents.Add(parent);
                    GetClassWithParents(parent, classHierarhy);

                }
            }

            public class ClassWithParents
            {
                public string ClassName { get; set; }
```

```
        public List<ClassWithParents> Parents { get; set; }
    }

    public static List<string> GetDeclarationsFromClassWithParents(this
ClassWithParents classWithParents, Dictionary<string,List<string>>
classesWithDeclaration)
    {
        var list = new List<string>();

        string key = string.Empty;
        var alternateKey =
classesWithDeclaration.Keys.FirstOrDefault(k =>
k.EndsWith($".{classWithParents.ClassName}"));
        if
(classesWithDeclaration.ContainsKey(classWithParents.ClassName)) key =
classWithParents.ClassName;
        else if (!string.IsNullOrEmpty(alternateKey) &&
classesWithDeclaration.ContainsKey(alternateKey)) key = alternateKey;

        if (string.IsNullOrEmpty(key)) return list;
        list.AddRange(classesWithDeclaration[key].Select(s=> key +
"."+s));
        if(classWithParents.Parents == null ||
!classWithParents.Parents.Any()) return list;
        foreach(var parent in classWithParents.Parents)
        {

    //list.AddRange(classesWithDeclaration[parent.ClassName]);
```

```
        list.AddRange(parent.GetDeclarationsFromClassWithParents(classesWithDeclaration));
                }
                return list;

            }

            public static List<string> GetAllChildren(Dictionary<string, List<string>> hierarhy, string className)
            {
                var partOfClassName = className.Split('.');
                var cNameLastPart = partOfClassName[partOfClassName.Length - 1];
                var children = hierarhy.Where(c => c.Value != null && (c.Value.Contains(className) || c.Value.Contains(cNameLastPart))).Select(c=>c.Key).ToList();
                if (!children.Any()) return null;
                var additionalChildren = new List<string>();
                foreach (var child in children)
                {
                    var tmp = GetAllChildren(hierarhy, child);
                    if(tmp!=null && tmp.Any())
                        additionalChildren.AddRange(tmp);
                }
                children.AddRange(additionalChildren);
                return children;
            }
        }
```

Program.cs

```
        private static void GetCalculatedMetricsAndSaveResult(string path, bool extended, bool splitFiles, string fName = "")
        {
            var localFiles = new List<string>();
            if (Directory.Exists(path))
                GetAllCsFilesUnderDirectory(path, 0, ref localFiles);
            else
                localFiles.Add(path);
            //files.AddRange(localFiles);
            var metrics = new List<SourceCodeMetrics>();
            //var tempFileName = $"temp{new DirectoryInfo(path).Name}.cs";
            //System.IO.File.WriteAllText(tempFileName,null);

            foreach (var file in localFiles)
            {
                metrics.AddRange(GetAllMetrics(file, extended));
            }
            //var compilationUnitMetrics = metrics.Where(mc => mc.IdentifierType == IdentifierTypeEnum.File);
            if (splitFiles)
            {
                var tempMetrics = metrics.Where(mc => mc.IdentifierType != IdentifierTypeEnum.File).GroupBy(mc => mc.Identifier).Select(scm => scm.AggreagateSourceCodeMetrics()).ToList();
```

```
                tempMetrics.AddRange(metrics.Where(mc => mc.IdentifierType == IdentifierTypeEnum.File));
                metrics = tempMetrics.OrderBy(mc => mc.Identifier).ToList();
            }
            else
            {
                var tempMetrics = metrics.GroupBy(mc => mc.Identifier).Select(scm => scm.AggreagateSourceCodeMetrics()).OrderBy(i => i.Identifier).ToList();
                metrics = tempMetrics;
            }
            var name = Directory.Exists(path) ? new DirectoryInfo(path).Name : new FileInfo(path).Name;
            var fileName = string.IsNullOrWhiteSpace(fName) ? $"{name}_metrics" : fName;
            //var serialize = JsonConvert.SerializeObject(metrics);
            //if()
            var csv = $"{fileName}.csv";
            var json = $"{fileName}.json";
            //if (!File.Exists(json)) File.Create(json);
            if (!File.Exists(csv))
            {
                File.WriteAllText(csv, "");
                //File.Create(csv);
                System.Threading.Thread.Sleep(5000);
                File.AppendAllText(csv, $"sep=|{Environment.NewLine}");
                File.AppendAllText(csv, CsvUtility.GetCsvHeader());
```

```
            }
            System.Threading.Thread.Sleep(5000);
            // File.AppendAllText(json, serialize);
            File.AppendAllText(csv, metrics.GetCsv());

        }

        public static void Main(string[] args)
        {
            if (args.Contains("-decimalcomma"))

                Thread.CurrentThread.CurrentCulture =
        new CultureInfo("ru-RU", false);
            else
                Thread.CurrentThread.CurrentCulture =
    new CultureInfo("en-US", false);

            var code =
        @"  using System;
            using System.Collections.Generic;
            using System.Text;

        namespace HelloWorld
        {
            public class a{
                private const int cIntPrivate = 8;
                public static int cIntPublicStatic = 18;
                protected static int cIntProtectedStatic = 11;
                public const int cInt = 13;
```

```
        public a()
        {
            Console.WriteLine(cInt);

            Console.WriteLine(cIntPrivate);
        }
    }

    public class b : a{
        public b()
        {
            Console.WriteLine(cInt);
            Console.WriteLine(""b created"");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(a.cIntPublicStatic);
            Console.WriteLine(a.cInt);
            Console.WriteLine(""Hello, World!"");
            var str = new string('c',3);

            var l = new b();
        }
    }
}";
```

```
                SyntaxTree tree = CSharpSyntaxTree.ParseText(code);
                var root = tree.GetRoot();

                var metrics = GetAllMetrics(root);

                new List<SourceCodeMetrics>() { metrics }.AggreagateSourceCodeMetrics();


                Console.WriteLine($"Welcome, {Environment.MachineName} {Environment.UserName}!");

                bool extended = args.Contains("-ex");
                bool splitFiles = args.Contains("-sp");
                if (args.Contains("-pf"))
                {
                    string path = null;
                    var indexOfSource = Array.IndexOf(args, "-pf") + 1;
                    if (indexOfSource < args.Length && !string.IsNullOrWhiteSpace(args[indexOfSource]) && File.Exists(args[indexOfSource]))
                    {
                        path = args[indexOfSource];

                        var dirsToAnalyze = System.IO.File.ReadAllLines(path);

                        var files = new List<string>();
```

```
                    var fileName = "result";

                    foreach (var f in
Directory.EnumerateFiles(".").Where(fd => fd.EndsWith(".csv")))
                        File.Delete(f);
                    foreach (var dir in
dirsToAnalyze.Where(d=>!string.IsNullOrEmpty(d)))
                    {
                        GetCalculatedMetricsAndSaveResult(dir,
extended, splitFiles, fileName);
                        Console.WriteLine($"{dir} - done");
                    }
                }
                else
                {
                    Console.WriteLine("Wrong path of the file with
projects' pathes! Try again!");
                }
            }
            else if (args.Contains("-d"))
            {
                string path = null;
                var indexOfSource = Array.IndexOf(args, "-d") + 1;
                if (indexOfSource < args.Length &&
!string.IsNullOrWhiteSpace(args[indexOfSource]) &&
Directory.Exists(args[indexOfSource]))
                {
                    path = args[indexOfSource];
```

```
                    GetCalculatedMetricsAndSaveResult(path,
extended, splitFiles);
                }
                else
                {
                    Console.WriteLine("Wrong dir path! Try again!");
                }
            }
            else if (args.Contains("-f"))
            {
                string path = null;
                var indexOfSource = Array.IndexOf(args, "-f") + 1;
                if (indexOfSource < args.Length &&
!string.IsNullOrWhiteSpace(args[indexOfSource]) &&
args[indexOfSource].EndsWith(".cs") && File.Exists(args[indexOfSource]))
                {
                    path = args[indexOfSource];
                    GetCalculatedMetricsAndSaveResult(path,
extended, splitFiles);
                }
                else
                {
                    Console.WriteLine("Wrong file name! Try
again!");
                }
            }
            else
            {
                Console.WriteLine($"Press any key to continue...");
```

```
                    Console.ReadKey();

                    var answer = "none";
                    while (answer != "0")
                    {
                        List<string> files = new List<string>();
                        Console.WriteLine("Please, choose what you want to do and enter the <#> of command:");
                        Console.WriteLine("1. Analyze specific file.");
                        Console.WriteLine("2. Analyze files in directory and all its subdirectories recursively.");
                        Console.WriteLine("0. Exit");
                        answer = Console.ReadLine();
                        switch (answer)
                        {
                            case "1":
                                Console.WriteLine("Enter the file name: ");
                                var fName = Console.ReadLine();
                                if (!string.IsNullOrWhiteSpace(fName))

    GetCalculatedMetricsAndSaveResult(fName, extended, splitFiles);
                                break;
                            case "2":
                                Console.WriteLine("Enter the directory name: ");
                                var dName1 = Console.ReadLine();
```

```
                                        if
(!string.IsNullOrWhiteSpace(dName1))

        GetCalculatedMetricsAndSaveResult(dName1, extended, splitFiles);
                                        break;
                                    case "0":
                                        Console.WriteLine("Goodbye!");
                                        break;
                                    default:
                                        Console.WriteLine("Wrong
command! Try again...");
                                        break;
                                }

                                Console.WriteLine($"Press any key to
continue...");
                                Console.ReadKey();
                            }
                        }
                    }

                    private static SourceCodeMetrics GetAllMetrics(SyntaxNode node)
                    {
                        var metric = new SourceCodeMetrics();
                        var code = node.ToString();

                        // 1. LOC
                        metric.PhysicalLoc                                    =
LinesOfCodeAnalysis.GetNumberOfPhysicalLines(code);
```

```
                //var LOCwithEmptyLines = GetNumberOfPhysicalLines(code,
false);

                // 2. Nesting
                metric.NestingMetric = NestingAnalysis.GetNesting(node);

                // 3. Halstead
                HalsteadAnalysis.GetHalsteadVariables(node, out double n1,
out double n2, out double N1, out double N2);
                metric.HalsteadAllOperators = n1;
                metric.HalsteadAllOperands = n2;
                metric.HalsteadDistinctOperators = N1;
                metric.HalsteadDistinctOperands = N2;
                //(node,out double n, out double N, out double Nt, out double V,
out double Lt, out double Ec, out double D, out double I, out double Et)

                // 4. Conditions
                metric.IfCounter                                    =
ConditionalAnalysis.GetIfStatementCount(node);
                metric.ElseCounter                                  =
ConditionalAnalysis.GetElseStatementCount(node);
                metric.TernaryOperatorCounter                       =
ConditionalAnalysis.GetTernaryOperatorCount(node);
                metric.WhereCount                                   =
ConditionalAnalysis.GetWhereCount(node);
                metric.SwitchCount                                  =
ConditionalAnalysis.GetSwitchOperatorCount(node);
                metric.GoToCount                                    =
ConditionalAnalysis.GetGoToOperatorCount(node);
```

```
                metric.LabelsCount =
ConditionalAnalysis.GetGoToOperatorCount(node);

                // 5. Loops
                metric.ForCounter = LoopAnalysis.GetForLoopCount(node);
                metric.ForeachCounter =
LoopAnalysis.GetForEachLoopCount(node);
                metric.WhileCounter =
LoopAnalysis.GetWhileLoopCount(node);
                metric.DoWhileCounter =
LoopAnalysis.GetDoWhileLoopCount(node);

                // 6. Fitzpatrick metric
                metric.FitzpatrickCounters =
FitzpatrickAnalysis.GetFitzpatrickMetrick(node);

                // 7. Spen metric
                metric.SpenMetric =
SpenVariablesAnalysis.GetSpenMetric2(node);
                metric.HierarchySctructureMetric =
SpenVariablesAnalysis.GetHierarchySctructureMetric(node);
                metric.AllVariables =
SpenVariablesAnalysis.GetAllVariables(node);
                metric.ClassAndDeclarations =
SpenVariablesAnalysis.GetClassDeclarations(node);
                metric.ConstAndStaticDeclarations =
SpenVariablesAnalysis.GetConstAndStaticVariables(node);

                // 8. Functions' metrics
```

```
                metric.FunctionCallsMetric =
FunctionAnalysis.GetFunctionsInvocationDegree(node);
                metric.FunctionParasiticCallsMetric =
FunctionAnalysis.GetParasiticCalls(node);

                return metric;
            }

            private static List<SourceCodeMetrics> GetAllMetrics(string
fileName, bool extended)
            {
                var fileContent = File.ReadAllText(fileName);
                var metrics = new List<SourceCodeMetrics>();

                SyntaxTree tree =
CSharpSyntaxTree.ParseText(fileContent.Replace(@"\u","_"));
                var root = tree.GetRoot();

                if (!extended)
                {
                    var m = GetAllMetrics(root);
                    m.Identifier = root.Parent == null ? "CompilationUnit" :
root.GetParentName("") + root.GetNodeName();
                    m.Path = fileName;
                    m.IdentifierType = IdentifierTypeEnum.File;
                    metrics.Add(m);
                    return metrics;
                }
```

```
                // Namespaces
                foreach (var namespaceNode in
root.DescendantNodesAndSelf().Where(n =>
n.IsKind(SyntaxKind.NamespaceDeclaration)))
                {
                    var m = GetAllMetrics(namespaceNode);
                    m.Identifier = namespaceNode.GetParentName("") +
namespaceNode.GetNodeName();
                    m.Path = fileName;
                    m.IdentifierType = IdentifierTypeEnum.Namespace;
                    metrics.Add(m);
                }
                //Classes
                foreach (var classNode in
root.DescendantNodesAndSelf().Where(n =>
n.IsKind(SyntaxKind.ClassDeclaration)))
                {
                    var m = GetAllMetrics(classNode);
                    m.Identifier = classNode.GetParentName("") +
classNode.GetNodeName();
                    m.Path = fileName;
                    m.IdentifierType = IdentifierTypeEnum.Class;
                    metrics.Add(m);

                }
                //Methods
                foreach (var methodNode in
root.DescendantNodesAndSelf().Where(n =>
n.IsKind(SyntaxKind.MethodDeclaration)))
```

```
                {
                    var m = GetAllMetrics(methodNode);
                    m.Identifier = methodNode.GetParentName("") + methodNode.GetNodeName();
                    m.Path = fileName;
                    m.IdentifierType = IdentifierTypeEnum.Method;
                    metrics.Add(m);
                }
                return metrics;
            }
            private static void GetAllCsFilesUnderDirectory(string path, int indent, ref List<string> files)
            {
                try
                {
                    files.AddRange(Directory.EnumerateFiles(path).Where(f => f.EndsWith(".cs")));
                    foreach (string folder in Directory.GetDirectories(path))
                    {
                        Console.WriteLine("{0}{1}", new string(' ', indent), Path.GetFileName(folder));
                        GetAllCsFilesUnderDirectory(folder, indent + 2, ref files);
                    }
                }
                catch (UnauthorizedAccessException) { }
            }
        }
```

Додаток В. Список публікацій здобувача за темою дисертації та відомості про апробацію результатів дисертації

1. Наукові праці, в яких опубліковані основні наукові результати дисертації:

1. Давидов В. В. Моделі та методи підвищення безпеки програмного забезпечення (монографія). Харків, 2021. 146 с.

2. Давидов В. В., Бульба С. С., Кучук Г. А. Метод розподілу ресурсів між композитними застосунками // Системи управління, навігації та зв'язку. 2018. Том 4 (50). С. 99-104. DOI: https://doi.org/10.26906/SUNZ.2018.4.

3. Давидов В. В., Гавриленко С. Ю., Прохорова Т. М. Дослідження методів побудови синтаксичних аналізаторів // Системи обробки інформації. 2015. № 11(136). С. 125-128.

4. Давидов В. В., Гавриленко С. Ю., Челак В. В. Разработка системы фиксации аномальных состояний компьютера // Вісник Національного технічного університету "ХПІ": Серія: Інформатика та моделювання. 2018. № 42 (1318). С. 109-121.

5. Давидов В. В., Гребенюк Д. С. Метод первинного виділення хмарних обчислювальних ресурсів на основі аналізу ієрархій // Системи управління, навігації та зв'язку. 2020. Том 3 (61). С. 80-85. DOI: https://doi.org/10.26906/SUNZ.2020.3.080.

6. Давидов В. В., Гребенюк Д. С. Метод прогнозування навантаження ресурсів хмарних обчислюваних систем // Проблеми інформатизації : міжнародна наук.-технічн. конф., тези доповідей. Черкаси. 2020. С. 73.

7. Давидов В.В., Мовчан А. В., Сидоренко И. И. Разработка системы формирования цифрового идентификатора программного обеспечения для защиты авторских прав // Системи обробки інформації. 2016. № 3. С. 15-17.

8. Давидов В. В., Можаєв М. А., Ліцзян Джан. Аналіз та порівняльні дослідження методів підвищення рівня безпеки програмного забезпечення // Сучасні інформаційні технології. 2020. Том 4. № 3. С. 124–132. DOI: https://doi.org/10.20998/2522-9052.2020.3.18.

9. Давидов В.В., Пасько Д.А., Молчанов Г.І. Управління необмеженою кількістю хмарних сховищ // Сучасні інформаційні технології. 2018. Том 2. №3. С. 49–53. DOI: https://doi.org/10.20998/2522-9052.2018.3.08.

10. Давидов В. В., Семенов С. Г., Зиков І. С. Дослідження ризиків моніторингу технічного стану об'єктів авіації // Системи озброєння і військова техніка. 2015. № 4(44). С. 108-110.

11. Давыдов В. В., Гребенюк Д. С. Комплекс процедур генерации лицензионного ключа для защиты авторских прав интеллектуальной собственности на программное обеспечение // Системи управління, навігації та зв'язку. 2017. № 1(41). С. 11-15. Режим доступу: http://journals.nupp.edu.ua/sunz/article/view/621.

12. Золотухіна О. А., Волошин Д. Г., Давидов В. В., Бречко В. О. Розробка імітаційної моделі процесу розрахунку і коригування безпечної польотної траєкторії безпілотного літального апарату // Телекомунікаційні та інформаційні технології. 2020. № 4 (69). С. 87–94.

13. Кучук Н. Г., Давидов В. В. Моделі і методи захисту інформаційних структур для комп'ютерних систем на інтегрованих програмних платформах (монографія). Харків, 2021. 160 с.

14. Семенов С. Г., Давидов В. В., Волошин Д. Г., Гребенюк Д. С. Метод захисту модуля програмного забезпечення на основі процедури обфускації // Телекомунікаційні та інформаційні технології. 2019. № 4 (65). С. 71–80. DOI: https://doi.org/10.31673/2412-4338.2019.047180.

15. Davydov V., Hrebeniuk D. Development the resources load variation forecasting method within cloud computing systems // Advanced Information

Systems. 2020. Vol. 4. No. 4. P. 128–135. DOI: https://doi.org/10.20998/2522-9052.2020.3.18.

16. Kuchuk N., Cherneva G., Davydov V. Method of packet fragmentation in unstable data exchange in computer networks on transport // Mechanics Transport Communications: Academic journal. Vol. 19. Is. 1. 2021. P. X1-1 – X1-7, Article № 2064.

17. Liqiang Zhang, Weiling Cao, Davydov V., Brechko V. Analysis and comparative research of the main approaches to the mathematical formalization of the penetration testing process // Information Processing Systems. 2021. № 2 (64). С.73-77.

18. Semenov S., Davydov V., Hrebeniuk D. Research of the software security model and requirements // Advanced Information Systems. 2021. Vol. 5. № 1. P. 87–92. DOI: https://doi.org/10.20998/2522-9052.2021.1.12.

19. Semenov S., Davydov V., Lipchanska O., Lipchanskyi M. Development of unified mathematical model of programming modules obfuscation process based on graphic evaluation and review method // Eastern-European Journal of Enterprise Technologies. 2020. № 3(2(105)). P. 6–16. DOI: https://doi.org/10.15587/1729-4061.2020.206232.

20. Semenov S., Liqiang Zhang, Weiling Cao, Davydov V. Development of protecting a software product methematical model from unlicensed copying based on the GERT method // Information Processing Systems. 2021. № 1 (164). P. 73-82. DOI: https://doi.org/10.30748/soi.2021.164.08.

2. Наукові праці, які засвідчують апробацію матеріалів дисертації:

21. Давидов В. В., Волошин Д. Г. Семенов С. Г. Про завдання позиціонування безпілотних літальних апаратів в умовах кібератак // Актуальні питання протидії кіберзлочинності та торгівлі людьми : Всеукраїнська наук.-практич. конф., збірник матеріалів. Харків, 2018. С. 311.

22. Давыдов В. В. Анализ методов обнаружения злоумышленного кода в Android приложениях // Інформаційні технології, наука, техніка, технологія, освіта, здоров'я: міжнар. наук.-практ. конф., тези доповідей. Харків, 2015. С. 36.

23. Давыдов В. В., Гребенюк Д. С. Особенности распределения ресурсов для многомашинных вычислительных комплексов // Проблеми інформатизації: міжнар. наук.-техн. конф., тези доповідей. Полтава, 2017. С. 21.

24. Давыдов В. В. Процедура генерации лицензионного ключа для защиты авторских прав // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління : міжнар. наук.-техн. конф., тези доповідей. Полтава, 2017. С. 52.

25. Кучук Н. Г., Давидов В. В., Гребенюк Д. С. Аналіз методів розрахунку розмірності мінковського для фрактального трафіка мультисервісної мережі e-learning // Проблеми інформатизації : міжнародна наук.-техн. конф., тези доповідей. Черкаси, 2018. С. 34.

26. Семенов С. Г., Давыдов В. В., Мовчан А. В. Система формирования цифрового идентификатора программного обеспечения для защиты авторских прав // Сучасні проблеми інформатики в управлінні, економіці, освіті та подоланні наслідків Чорнобильської катастрофи : міжнарод. наук. сем., тези доповідей. Київ, 2016. С. 110-116.

27. Семенов С. Г., Давидов В. В. Оценка рисков контроля и диагностики технического состояния объектов критического применения // Metrology and Metrology assurance 2016. 26th National Scientific Symposium with international participation. Sozopol, Bulgaria. 2016. P. 395-399.

28. Davydov V., Hrebeniuk D. Development of the methods for resource reallocation in cloud computing systems // Innovative Technologies and Scientific Solutions for Industries. 2020. № 3 (13), P. 25–33. DOI: https://doi.org/10.30837/ITSSI.2020.13.025.

29. Davydov V., Zmiivska V., Shypova T., Lysytsia D. Analysis of fractal noise indicators in measuring systems of technical objects // Metrology and metrology assurance 2018 : 28th International Scientific Symposium, September 10-14, 2018 : Sozopol, Bulgaria. P. 44-47.

30. Kuchuk N., Shyman A., Hrebeniuk D., Davydov V. Mathematical model of the information system synthesis process // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління : міжнародна наук.-техн. конф., матеріали. – Баку: ВА ЗС АР; Харків: НТУ «ХПІ»; Київ: НАУ; Харків: ДП «ПДПРОНДІАВІАПРОМ»; Жиліна: Університет, 2021. С. 21.

31. Semenov S., Bartosh M., Davydov V., Turuta O. Improvement of the Task Scheduler Model Taking Into Account the Heterogeneity of the Entities // Fifth International Scientific and Technical Conference "Computer and Information Systems and Technologies". 2021. P. 42-43. DOI: https://doi.org/10.30837/csitic52021232181.

32. Semenov S., Davydov V., Semenova A., Voloshyn D., Lymarenko V. Method of UAVs Quasi-Autonomous Positioning in the External Cyber Attacks Conditions // 10th International Conference on Dependable Systems, Services and Technologies (DESSERT). 2019.

33. Semenov S., Davydov V., Voloshyn D. Data Protection Method of an Unmanned Aerial Vehicle based on Obfuscation Procedure // CEUR Workshop Proceedings, 2020, Volume 2654. P. 515-525.

34. Semenov S., Davydov V., Voloshyn D. Obfuscated Code Quality Measurement // Metrology and metrology assurance 2019 : 29th International Scientific Symposium. September 6-9, 2019. Sozopol, Bulgaria. P. 44-47.

35. Semenov S., Davydov V., Weilin C., Liqiang Z., Petrovskaya I. Enhanced Software vulnerability model // Інформаційні технології і безпека : міжнародна наук.-практ. конф., матеріали. Київ, 2020. С. 56-60.

36. Semenov S., Hrebeniuk D., Davydov V. Software copyright protection using identification key // Aviation in the XXI-st Century: the 7th World Congress, September 19-21, 2016: Kyiv, Ukraine. P. 1.10.20-1.10.23.