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

«На правах рукопису»
УДК 004.056

«До захисту допущено»
В.о. завідувача кафедри
__________ М.В.Грайворонський
“___”_____________2019 р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності: 125 Кібербезпека

на тему: Технології автоматичного виправлення помилок безпеки в програмному забезпеченні

Виконав (-ла): студент (-ка) 2 курсу, групи ФБ – 81мп
(шифр групи)

Савченко Артем Юрійович____________________________ __________
(прізвище, ім’я, по батькові) (підпис)

Науковий керівник к.т.н., доц. Стьопочкіна Ірина Валеріївна __________
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент ____ _________
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____________
(підпис)

Київ – 2019 року

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

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою
Спеціальність (освітньо-професійна програма) – 125 Кібербезпека («Системи, технології та математичні методи кібербезпеки)

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
__________ М.В.Грайворонський
(підпис)
«___»_____________2019 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Савченко Артем Юрійович
(прізвище, ім'я, по батькові)

1. Тема дисертації Технології автоматичного виправлення помилок безпеки в програмному забезпеченні

науковий керівник дисертації ____ ,
к.т.н., доц. Стьопочкіна Ірина Валеріївна ______ ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «15» листопада 2019 р. № 3927-с

2. Термін подання студентом дисертації 10.12.2019 р.

3. Об'єкт дослідження ____

____

4. Вихідні дані ____

____

____

____

5. Перелік завдань, які потрібно розробити ____________________________

____

____

____

____

6. Орієнтовний перелік ілюстративного матеріалу ____________________
____
____

7. Орієнтовний перелік публікацій __________________________________
____
____

8. Дата видачі завдання __________

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка

Студент ____________ ____________________
(підпис) (ініціали, прізвище)

Науковий керівник дисертації ____________ ____________________
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Обсяг роботи 96 сторінок, 24 ілюстрації, 23 таблиці, 88 джерел літератури.

Об'єктом дослідження є вразливе програмне забезпечення, що містить помилки безпеки.

Предметом дослідження є методи аналізу проміжного представлення коду, методи глибинного навчання для пошуку вразливостей та методи автоматичного виправлення помилок безпеки в коді програмного забезпечення.

Методи дослідження - представлення початкового коду у вигляді абстрактного синтаксичного дерева, методи глибинного навчання, що дозволяють генерувати виправлення для помилок безпеки.

Наукова новизна полягає в тому, що отримав подальший розвиток метод виправлення помилок безпеки в програмному забезпеченні написаному мовою програмування С/С++ на основі детермінованих правил шляхом додавання специфічних шаблонів, що автоматично трансформують абстрактне синтаксичне дерево виправлячи відповідну помилку безпеки. Також отримав подальший розвиток метод виправлення помилок безпеки на основі глибинного навчання шляхом попередньої обробки коду для підвищення точності завдяки видобуванню найбільш істотних ознак для помилки безпеки.

Результати роботи викладені у третьому розділі, що демонструють роботу систем виправлення помилок безпеки на основі детермінованих шаблонів та на основі глибинного навчання.

Результати роботи можуть бути використані виправлення специфічних помилок безпеки в початковому коді програмного забезпечення.

АСД, помилка безпеки, автоматичне виправлення помилок

ABSTRACT

The volume of work is 96 pages, 24 illustrations, 23 tables, 88 sources of literature.

The object of the study is vulnerable software that contains security issues.

The subject of the study is methods of analysis of the intermediate code representation, methods of deep learning to find vulnerabilities and methods of automatic patch generation for security issues in software.

Research methods - presenting the source code in the form of an abstract syntax tree, deep learning methods that allow you to generate patches for security issues.

The scientific novelty is that the method of correcting security errors in software written in C / C ++ programming language based on deterministic rules has been further developed by adding specific templates that automatically transform the abstract syntax tree by correcting the corresponding security error. A method of correcting security errors based on deep learning has also been further developed by pre-processing code to improve accuracy by extracting the most essential features for a security error.

The results of the work are presented in Section 3, which demonstrates the performance of security-based path generation systems based on deterministic patterns and deep learning.

The results of the work can be used to generate patches for specific security issues in the source code of the software.

AST, security bug, security issues, automatic patch generation

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів ... 7
Вступ ... 8
1 Аналіз існуючих методів автоматичного виправлення помилок безпеки ... 11
1.1 Поняття «помилка безпеки» ... 11
1.2 Ключові поняття автоматичного виправлення помилок безпеки ... 11
1.3 Порівняння існуючих підходів та систем ... 13
2 Розробка методу виправлення помилок безпеки ... 27
2.1 Метод представлення початкового коду ... 27
2.2 Класифікація помилок безпеки ... 28
2.3 Теоретичні відомості ... 30
2.4 Архітектура розроблених методів виправлення помилок безпеки ... 38
Висновки до розділу 2 ... 42
3 Аналіз ефективності розробленої технології ... 43
3.1 Архітектура та програмна реалізація запропонованих методів ... 43
3.2 Результати дослідження ... 53
Висновки до розділу 3 ... 58
4 Розробка стартап-проекту ... 59
4.1 Опис ідеї проекту ... 59
4.2 Технологічний аудит ідеї проекту ... 62
4.3 Аналіз ринкових можливостей запуску стартап-проекту ... 64
Висновки до розділу 4 ... 84
Висновки ... 85
Перелік джерел посилання ... 87

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

АСД – абстрактне синтаксичне дерево

NN – neural network

RNN – recurrent neural network

LSTM – long short-term memory

GRU – gated recurrent unit

seq2seq – sequence-to-sequence

ВСТУП

Помилки безпеки стають головною загрозою для безпеки багатьох систем. Зловмисники можуть вкрасти приватну інформацію та вчиняти шкідливі дії, використовуючи помилки безпеки програмного забезпечення. Помилки безпеки часто залишаються незнайденими протягом тривалого часу, оскільки вони можуть не впливати на типові функціональні можливості систем. Крім того, розробнику часто важко правильно виправити помилку, якщо він не є експертом у сфері інформаційної безпеки.

Написання вручну шаблонів та визначення правил виправлення помилок безпеки в програмному забезпеченні – дуже клопітка та трудоємна діяльність. З розвитком технологій та комп’ютерних систем можуть виникати різні помилки, і виправлення кожної з них може потребувати створення нових шаблонів виправлення помилок безпеки. Вищевказані факти підкреслюють важливість розробки методик, які можуть автоматично створювати шаблони для автоматичного виправлення помилок безпеки програмного забезпечення.

На сьогодні існує багато статичних аналізаторів коду, які знаходять помилки безпеки в існуючих продуктах. Більшість з цих продуктів не мають підтримки з боку розробників, саме тому ця робота є **актуальною**, адже виникає необхідність автоматизації процесу виправлення помилок безпеки у програмному забезпеченні, які раніше уже були виправлені в інших програмних продуктах. Відомий тренд нейронних мереж – вони автоматизують рутину роботу, а тому можуть допомогти з проблемою виправлення помилок безпеки, що і було доведено у даній роботі.

Об'єктом дослідження є вразливе програмне забезпечення, що містить помилки безпеки.

Предметом дослідження є методи аналізу проміжного представлення коду, методи глибинного навчання для пошуку вразливостей та методи автоматичного виправлення помилок безпеки в коді програмного забезпечення.

Метою даної роботи є дослідження алгоритмів глибинного навчання, що дозволяють автоматично створювати виправлення для помилок безпеки. А також, розробка відповідної технології автоматичного виправлення помилок безпеки у програмному забезпеченні.

Для досягнення цієї мети було поставлено наступні **завдання**:

- проаналізувати існуючі підходи автоматичного виправлення помилок безпеки у програмному забезпеченні, визначити їх переваги та недоліки;
- розробити власну систему, яка б базувалася на основі власноруч створених шаблонів для специфічних помилок безпеки;
- проаналізувати існуючі методи глибинного навчання, які можна використати для автоматизації процесу створення виправлень для помилок безпеки, що мають приклади реальних виправлень;
- розробити технологію автоматичного виправлення помилок безпеки у програмному забезпеченні, що базується на одному з алгоритмів глибинного навчання;
- провести тестування програмних реалізацій розроблених технологій та зробити висновки.

Методами дослідження є представлення початкового коду у вигляді абстрактного синтаксичного дерева, методи глибинного навчання, що дозволяють генерувати виправлення для помилок безпеки.

Наукова новизна: отримав подальший розвиток метод виправлення помилок безпеки в програмному забезпеченні написаному мовою програмування C/C++ на основі детермінованих правил шляхом додавання специфічних шаблонів, що автоматично трансформують абстрактне синтаксичне дерево виправлячи відповідну помилку безпеки. Також отримав подальший розвиток метод виправлення помилок безпеки на основі глибинного навчання шляхом впровадження попередньої обробки коду для підвищення точності завдяки видобуванню найбільш істотних ознак для помилки безпеки.

Практичне значення: результати роботи можуть бути використані компаніями, що розробляють програмне забезпечення та програмістами для швидкого і якісного виправлення специфічних помилок безпеки.

Апробація: результати роботи доповідалися на конференції Samsung Machine Learning Seminar та відповідну статтю було прийнято до друку в журналі «Theoretical and applied cybersecurity».

1 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ АВТОМАТИЧНОГО ВИПРАВЛЕННЯ ПОМИЛОК БЕЗПЕКИ

1.1 Поняття «помилка безпеки»

Помилка безпеки – ненавмисний недолік програмного коду або системи, яка робить його відкритим для потенціальної експлуатації у вигляді несанкціонованого доступу або зловмисної поведінки. Також стосуючись експлойтів безпеки, помилки безпеки можуть бути наслідком помилок програмного забезпечення, слабких паролів або програмного забезпечення, яке вже було заражене комп’ютерним вірусом або вживлянням скрипту, і ці помилки безпеки потребують патчів або виправлень, щоб уникнути можливих порушень цілісності з боку хакерів або зловмисних програм [1].

Помилка безпеки – наявність слабкості, помилки в розробці чи впровадженні програмного забезпечення, яка може призвести до несподіваної, небажаної події, що погрожує безпеці комп’ютерної системи, мережі чи програми [2].

1.2 Ключові поняття автоматичного виправлення помилок безпеки

У літературі є багато синонімів до слова «помилка»: дефект, несправність, недолік, збій у роботі, нездатність, хиба. Існують досить прийняті визначення між несправностями, помилками та збоями у роботі[3]: збій - це спостережувана неприйнятна поведінка; помилка - це неправильний стан, що поширюється до настання збою (яку ще не було помічено); несправність - першопричина помилки (зокрема, неправильний код). Хоча ясність цих трьох понять є відносною, навряд

чи можна сказати, що в літературі, у тому числі в нових публікаціях, строго дотримуються цих визначень. Більш того, якщо розглядати лише літературу про виправлення помилок, то абсолютно не виникає розмежування між поняттями «автоматичне виправлення збоїв у роботі», «автоматичне виправлення помилок», «автоматичне виправлення несправностей». Однак нам потрібна загальна концепція для всіх слів, і в цій роботі термін "помилка" використовується як узагальнене слово через його інтуїтивність та широке використання, з таким визначенням: помилка - це відхилення між очікуваною поведінкою виконання програми та тим, що насправді сталося [4].

Це визначення помилки включає поняття "поведінки", "виконання", "програми", але має неявну третю тему: спостерігача або опорну точку, яка вважає поведінку несподіваною. Цей "спостерігач", очевидно, може бути людиною, кажучи, що "цей вихід невірний". Класично це також специфікація, в загальному її значенні: специфікація - це сукупність очікуваної поведінки. Технічні характеристики поліморфні: вони можуть бути документами на природній мові, формальними логічними формулами, тестовими наборами та ін. Вони можуть бути навіть неявними: наприклад, специфікація "програма не виходить з ладу на будь-якому вході" має значення для багатьох програм, але часто не чітко пишеться. В якійсь мірі користувач, який каже "цей вихід невірний", констатує специфікацію на льоту. Отже, автоматичне виправлення помилок безпеки завжди посилається на специфікацію і дає наступне визначення. Автоматичне виправлення - це перетворення неприйнятної поведінки виконання програми у прийнятну відповідно до специфікації.

Концепція, близька до специфікації - концепція валідаторів. Простіше кажучи, валідатор визначає, чи правильний результат виконання програми [5]. У цій мірі специфікація та валідатор стосуються одного і того ж: очікування, прийнятності, правильності. Однак існує велика різниця між обома. Валідатор є лише частиною специфікацій, це частина, що стосується очікуваного виходу (коли такий існує). Крім того, специфікація містить інформацію про вхідні діапазони, про нефункціональні властивості, тощо. Наприклад, тестовий набір -

це специфікація, він містить тестові випадки, які самі містять твердження, останні - валідатори.

Що стосується виправлення помилок, валідатори можна розділити на два: валідатор помилок відноситься до валідаторів, які виявляють несподівані поведінки; регресійний валідатор відноситься до валідаторів, які перевіряють, чи не було введено нових помилок під час виправлення. Причина полягає в тому, що програма після виправлення вже задовольняє всі регресивні валідатори, але застосування виправлення може випадково призвести до регресії.

У загальному, техніка виправлення часто націлена на клас помилок. Клас помилок - це абстрактне поняття, що відноситься до сімейства помилок, у яких є щось спільне: ті ж симптоми, однакова першопричина, те саме рішення [6]. Наприклад, добре відомі класи помилок вихід за межі масивів, витоки пам’яті тощо. Однак є багато класів помилок, для яких в літературі немає чіткого визначення та обсягу, і деякі з них навіть не мають імені. Хоча існують деякі початкові систематики [7-8], побудова всеосяжної систематики класів помилок безпеки вимагатиме багаторічних досліджень.

1.3 Порівняння існуючих підходів та систем

Виправлення помилок складається зі зміни поведінки програми, тобто зміни її коду. Модифікація може бути здійснена як у вихідному коді, так і у двійковому коді (наприклад, байт-код Java або нативний код x86). Виправлення помилок безпеки можна виконати в режимі офлайн або онлайн під час виконання.

У офлайн режимі виправлення помилок може бути здійснено в середовищі розробки (IDE) розробників технічного обслуговування або на сервері безперервної інтеграції. Онлайн виправлення помилок безпеки означає, що дії виконуються на розгорнутому програмному забезпеченні. Технічно виправлення

помилок під час виконання передбачає своєрідне динамічне оновлення програмного забезпечення (DSU), яке саме по собі є дослідницькою темою.

Виправлення помилок безпеки потребує оператора виправлення, який є своєрідною модифікацією програмного коду. Наприклад, одним з операторів виправлення є додавання попередньої умови, як показано нижче:

```
+    if (age<16)
         serve_kid_content ()
```

У літературі визначено багато різних операторів виправлення, які будуть представлені далі. Іноді оператор виправлення включає шаблон виправлення, який є параметризованим фрагментом коду, що спрямовано на відновлення певного класу помилок. Модель виправлення [9] - це сукупність операторів виправлення.

Наприклад, розглядаючи тестовий набір як специфікацію, проблема виправлення помилок безпеки полягає у наданні програмі та її тестовому набору, до складу якого, щонайменше, входить один неуспішний тестовий випадок, створеного виправлення, після якого задовольняються всі випадки тестового набору. Таку постановку проблеми можна назвати виправленням помилок безпеки на основі тестового набору [6], вона найбільш широко була досліджена у статті про Genprog і більш детально буде описана у підрозділі 1.3.1.

Базуючись на описаній у підрозділі 1.2 класифікації автоматичне виправлення помилок залежить від обраного валідатора. У подальших підрозділах існуючі підходи автоматичного виправлення помилок класифікуються згідно з обраним валідатором.

1.3.1 Тестові набори

Тестовий набір - специфікація, заснована на вході-виході. У сучасному об'єктоорієнтованому програмному забезпеченні вхід може бути таким же складним, як набір взаємопов'язаних об'єктів, побудованих з багатою послідовністю викликів методів, а вихід може також являти собою послідовність викликів методів, які спостерігають за станом виконання та поведінкою різними способами. У виправленні помилок на основі тестового набору несправний тестовий випадок виконує функцію валідатора помилок, решта тестові випадки, що проходять, виступають як валідатор регресії.

Genprog - це система виправлення помилок безпеки на основі тестів та архетипів, створена в університеті Вірджинії [10-12]. GenProg використовує генетичне програмування для пошуку можливих виправлень. Його еволюційні обчислення представляють кандидатів виправлення, як послідовність редагувань вихідного коду програмного забезпечення. Кожне потенційне виправлення з великої кількості кандидатів застосовується до оригінальної програми для створення нової програми, яка оцінюється за допомогою тестових наборів. Потім, ті кандидати, які проходять більше тестів, мають більш високу придатність та ітеративно піддаються обчислювальному аналогу біологічних процесів мутації та схрещування. Цей процес закінчується, коли знайдено кандидата виправлення, який зберігає всі необхідні функціональні можливості та виправляє помилку. GenProg не вимагає спеціальних анотацій на коді чи формальних специфікацій, і застосовується до немодифікованого застарілого програмного забезпечення [13]. Найбільша оцінка Genprog [14] стверджує, що він може виправити 55 із 105 помилок. Ідеї, що були закладені у Genprog були далі розвинуті у статтях різних колективів [15-16] та інших лабораторіях [17-18]. Тепер, коли основні ідеї Genprog добре відомі та прийняті, необхідно провести роботу з удосконалення основних операторів виправлення (таких як [19]).

Набагато раніше до Genprog, у середині 90-х, у статті [20] було запропоновано автоматичне виправлення у звичайній іграшковій мові під назвою EXP. Специфікація - це набір тестових випадків (тобто тестовий набір). Наскільки відомо, це перше явище виправлення помилок на основі тестового набору в літературі.

У статтях [21-23] визначено 7 операторів виправлення на основі модифікації абстрактного синтаксичного дерева. Наприклад, для "сприяння мутації" вузол замінюється одним його дочірніх вузлів. Оператори укладаються випадковим чином. Реалізація прототипу під назвою Jaff обробляє підмножину програм написаних мовою програмування Java.

Автори статей [24-25] пропонують використовувати стандартні мутації з мутаційної літератури для тестування для виправлення помилок у програмах. Тому, їх моделями виправлення помилок є: заміна арифметичного, реляційного, логічного операторів, операторів збільшення/зменшення або оператору присвоєння іншим оператором із того ж класу; заперечення рішення у конструкції if або while. Зазвичай вони знаходять помилки за допомогою методики локалізації несправностей, заснованої на діапазоні. У статті [25] також використовують мутації для виправлення помилок і у порівнянні з попередніми авторами, вони всебічно досліджують простір усіх мутацій.

Ключова ідея авторів статті [26] полягає у створенні мета-програми, яка інтегрує всі можливі мутації відповідно до оператора мутації. Мутації, які фактично виконуються, керуються мета-змінними. Виправлення - це набір значень для цих мета-змінних. Мета-змінні оцінюються за допомогою символічного виконання.

У статті [27] було запропоновано підхід під назвою Semfix для виправлення помилок на основі символічного виконання та синтезу коду. Місце помилки виявляється за допомогою "ангельського зневадження"(angelic debugging) [28], потім виправлений вираз синтезується на основі синтезу вхідних та вихідних компонентів [29]. Одна з проблем із Semfix - масштабованість. Для подолання цієї проблеми та сама група авторів запропонувала Angelix [30].

Angelix - це система виправлення помилок безпеки, подібна до Semfix, де фаза символьного виконання була серйозно оптимізована з метою масштабування до великих програм та отримання більш ніж одного значення.

Система PAR [31] - це підхід для автоматичного виправлення помилок коду Java. PAR заснований на шаблонах виправлення: кожен з десяти шаблонів виправлень PAR являє собою загальний спосіб виправлення помилки. Наприклад, звичайна помилка - це доступ до нульового вказівника, і загальним виправленням цієї помилки є додавання перевірки на недійсність безпосередньо перед небажаним доступом: це шаблон "Null Pointer Checker". Деякі шаблони параметризовані змінними, наприклад, шаблон «Null Pointer Checker» приймає ім'я змінної як параметр. Шаблони застосовуються та тестуються випадковим способом пошуку.

Система Nopol [32] націлена на конкретний клас несправності: умовні помилки. Він виправляє програми, модифікуючи існуючий умовний оператор if або додаючи попередню умову до будь-якого оператора чи блоку в коді. Змінена або вставлена умова синтезується за допомогою синтезу коду на основі вводу-виводу за допомогою SMT [29] та предикатної комутації [33]. Система Nopol була розширена для виправлення нескінченних циклів [34].

Автори статті [35] запропонували Relifix - систему виправлення регресійних помилок. Підхід складається з 8 шаблонів виправлення, одні з яких є операторами трансформації, а інші - параметризованими шаблонами виправлення. Ключова ідея Relifix полягає в тому, що застосунок шаблонів керується попередніми змінами, наприклад, шаблон "додати вираз" додає лише вирази, які були задіяні в попередніх комітах, пов'язаних з регресією.

У статті [36] також виконують виправлення помилок на основі тестового набору, з метою синтезувати прості патчі. Для цього використовуються дуже специфічний вид програм: ті, які можна виразити як сліди матриці (пов'язані з булевими програмами [37]). Згідно з цим припущенням, вони можуть констатувати проблему з виправленням, як проблему максимальної

задоволеності (MaxSAT), де найменший патч - той, який задовольняє найбільші обмеження.

SPR [38] визначає набір операторів з поетапного виправлення таким чином, щоб якнайшвидше відмовитись від багатьох виправлень, які не можуть пройти доданий тестовий набір. Це дозволяє цілком дослідити невеликий і необхідний простір пошуку.

Ідея CodePhage [39] полягає в перенесенні перевірки з однієї програми на іншу програму, щоб уникнути збоїв. Система передбачає введення помилок, які виводять з ладу одну програми, та не виводять іншу. Розглянуті помилки це доступ до виходу за межі, переповнення цілих чисел та ділення на нуль. Пропущена перевірка виводиться із символічного вираження над полями введення та підтверджується набором тесту регресії.

Автори статті [40], пропонують SearchRepair, систему, що бере початок у пошуку коду. SearchRepair спочатку індексує фрагменти коду як обмеження SMT, потім під час виправлення помилки фрагмент виправляється, поєднуючи потрібні пари введення-виводу та фрагменти в одній проблемі обмеження. Система оцінюється на невеликих програмах на мові програмування C, написаними студентами в онлайн-курсі.

Prophet [41] – це система виправлень помилок безпеки, яка використовує попередні коміти для управління процедурою створення виправлення. Все, що можна вивчити з попередніх комітів з систем контролю версій є розподілом ймовірностей над характеристиками виправлення. Цей розподіл ймовірностей потім використовується для прискорення створення виправлення та для збільшення ймовірності знайти правильні виправлення.

Prophet – це перша система, що вивчає ймовірнісну модель правильного коду. Prophet використовує цю модель для автоматичного створення правильних виправлень для дефектів в неправильних програмах:

- Вивчення моделі правильного коду: Працюючи з набором успішних виправлень, отриманих із відкритих репозиторіїв програмного забезпечення, Prophet вивчає ймовірнісну модель правильного коду. А

потім використовує цю модель для ранжирування та ідентифікації правильних виправлень у автоматично створеному просторі кандидатів на виправлення (лише невелика частина яких є правильними).

- Взаємодія з кодом: Кожне виправлення вставляє новий код у програму. Але правильність залежить не лише від нового коду, а також, від того як новий код взаємодіє з програмою в яку його було вставлено. Отже, вивчена модель коректності працює з характеристиками, які характеризують аспекти того, як новий код взаємодіє з оточуючим кодом програми, що потребує виправлення.

- Універсальні характеристики: Основною новою гіпотезою, що закладена в проект Prophet є те, що правильний код у всіх застосунках має основний набір властивостей універсальної коректності. Для знаходження і вивчення цих властивостей Prophet працює з універсальними характеристиками, які абстрагують неглибокі синтаксичні деталі, що прив'язують код до конкретних програм залишаючи на семантичному рівні недоторканими основні властивості коректності. Ці універсальні характеристики мають вирішальне значення для того, щоб Prophet міг вивчити модель правильності виправлень для одного набору застосунків, а потім успішно застосувати цю модель до нових, раніше небачених застосунків.

- Великі програми: на відміну від багатьох попередніх систем створення автоматичних виправлень, які працюють з невеликими програмами на декілька сотень рядків коду [42-45], Prophet створює правильні виправлення для великих реальних програм із десятками тисяч або навіть мільйонами рядків коду. Попередня сучасна система для таких програм використовує набір евристик, отриманих вручну, для ранжування кандидатів на виправлення [46]. Експериментальні результати показують, що модель коректності коду Prophet дозволяє перевершити попередню найсучаснішу систему на тому ж наборі тестів.

Існуючі системи «генерації та валідації» у якості вхідних даних приймають програму та набір тестових випадків, серед яких є принаймні один, що виявляє помилку безпеки в програмі. Потім система генерує простір кандидатів на виправлення і шукає у цьому просторі правдоподібні виправлення, які дають правильні результати для всіх тестових випадків у тестовому наборі. На жаль, наявність правдоподібних, але неправильних виправлень (які проходять усі тестові випадки з тестового набору, але у інших випадках дають неправильний вихід) ускладнила здатність пошуку правильних виправлень у попередніх систем «генерації та валідації» [46-47]. Prophet використовує свою навчену модель правильного коду для ранжирування патчів у своєму просторі пошуку з метою отримання правильного виправлення, як першого (або одного з перших) виправлень для перевірки.

Оцінка проводилась на 69 реальних помилок безпеки за базою Genprog і показала результат у 15 створених правильних виправлень. Автори статті [48] також використовують історію для вибору найбільш ймовірного виправлення. На відміну від Prophet, експерименти проводилися на програмах написаних мовою програмування Java.

1.3.2 Передумови та післяумови

У деяких роботах використовуються передумови та післяумови, як валідатори для виправлення помилок.

Автори статті [49] використовують передумови та післяумови для обчислення «гіпотезованих станів програми» (від післяумови) та «фактичних програмних станів» (від входу, що призводить до помилки). Оператори виправлення полягають у зміні LHS або RHS призначень або зміні булевої умови простими модифікаціями (зміна змінної, зміна реляційного оператора), щоб

гіпотезований стан програми став сумісним з фактичним станом програми. Класичний тестовий набір використовується для виявлення регресій.

AutoFix-E [50-51] – це підхід, що створює виправлення для програм написаних мовою програмування Eiffel опираючись на контракти (передумови, постумови, інваріанти). AutoFix-E використовує чотири шаблони виправлення, які складаються з фрагменту та порожнього умовного виразу для синтезу. Основна ідея AutoFix-E полягає у тому, що і фрагмент коду, і умовний вираз беруться з існуючих контрактів.

У статті [52] використовуються передумови та післяумови написані мовою специфікації Alloy. Тіло функції також переведено на формули мови Alloy. Потім обмежений механізм верифікації Alloy використовуються як для виявлення помилок (аналогічно [53]), так і для ідентифікації виправлення. Оператори виправлення змінюють RHS призначень та змінюють існуючі кондиції умови.

Автори статті [54] розглядають твердження, як специфікації в програмах, які можна перевести на SMT. Підхід статичний і показано, що виправлення не порушує твердження для розглянутої вхідної області. Підхід заснований на шаблонах виправлень, таких як зміна RHS призначень або зміна арифметичного вираження лінійною комбінацією. Пусті місця в шаблонах заповнюються розв’язувачем SMT.

1.3.3 Абстрактні моделі поведінки

Абстрактна модель поведінки, машинний стан, що кодує стан об’єкта або відповідні виклики дозволеного методу можуть використовуватися для управління виправленням помилок безпеки.

У 2006 році, перед створенням Genprog, у статті [55] було запропоновано першу техніку автоматичного створення виправлень помилок безпеки. Вона вимагає у якості входу політику безпеки (наприклад, правила використання

застосунка) та граф потоку управління відповідного методу, що містить помилку безпеки. Весь підхід статичний: помилка безпеки виявляється як статичне порушення властивості безпеки, а правильність умови виправлення полягає лише в тому, щоб пройти перевірку безпеки.

Автори статті [56] представили Pachika, підхід виправлення програм написаних мовою програмування Java. Основна ідея Pachika полягає в тому, щоб спочатку зробити висновок про модель використання об'єктів із виконань застосунку, а потім створити виправлення для невдалого запуску, щоб відповідати початковій очікуваній правильній поведінці. Оцінювання складається з виправлення 18 помилок ASPECTJ (75KLOC) та 8 помилок RHINO (38KLOC). Два оператори виправлення Pachika здійснюють додавання та видалення викликів методів. Основна відмінність від попереднього підходу полягає в тому, що модель поведінки добувається у ході виконання, а не надається на вході.

1.3.4 Статичний аналіз

Засоби статичного аналізу видають помилки та попередження, які можна автоматично виправляти. У цьому випадку валідатором є сам статичний аналізатор.

Автори статті [57] пропонують підхід для виправлення помилок безпеки на основі свого статичного аналізатору .Net коду. Для набору класів помилок, що ідентифікуються статично, вони пропонують відповідні операції з виправлення. Оператори виправлення є специфічними для кожного класу помилок, наприклад, це може бути додавання передумови, зміна розміру масиву, тощо. Потім знову проводиться статичний аналіз, щоб перевірити правильність виправлення помилок безпеки.

У статті [58] виправлення націлені на специфічний клас помилок безпеки, а саме на помилки, пов'язані з арифметикою цілих чисел (лінійні комбінації). Арифметичні переповнення виявляється статично, а запропоноване виправлення – це переупорядкування арифметичних операцій. Виправлення гарантує, що переповнення більше не може відбутися. Про арифметичні переповнення також написана робота [59].

Автори статті [60] представляють підхід автоматичного виправлення помилок безпеки пов'язаних з витоками пам'яті у програмах написаних мовою програмування С. Підхід складається з статичного виявлення та виправлення витоків пам'яті шляхом вставлення операторів делокації. Оцінка проводилась на 14 програмах у яких було виправлено 242 помилки.

У статті [61] було запропоновано підхід до виправлення помилок компілятора, який у свою чергу є статичним валідатором. Оригінальність підходу DeepFix полягає у використанні мовної моделі заснованої на глибинному навчанні для пропонування виправлень. Вони оцінюють свій підхід виправляючи помилки у студентських програмах з онлайн-курсів.

Автори статті [62] статично виявляють переповнення буферу. Також вони мають шаблони з параметризованою змінною. Правильна змінна, що буде використовуватись у шаблоні знаходиться за допомогою статичного машинного перекладу.

1.3.5 Вхідні дані, що приводять до збоїв

Поведінкове виправлення може застосовуватися у відповідь на падіння програми. Процес виправлення відбувається після того, як ідентифікуються вхідні параметри, що призводять до краху програмного забезпечення, та за можливості мінімізуються. Тестовий випадок, що призводить до падіння програми також може використовуватися у якості вхідних даних, що призводять

до падіння. Однак основна відмінність вхідних даних, що призводять до падіння програми, від тестових наборів з точки зору валідаторів для ремонту полягає у наступному. Тестовий набір також містить тестові випадки (так званий регресійний валідатор) і містить очікуване результуюче значення, у той час як вхідні дані, що призводять до краху стоються лише порушення нефункціонального контракту «програма не повинна падати».

У статті [63] виправлення винятків базується на рекомендаціях Stackoverflow. Ця система називається QACrashFix та виконує виправлення створюючи пару між кодом з помилкою безпеки та виправленим кодом з Stackoverflow, для того щоб витягти сценарій редагування. Сценарії редагування послідовно приміняються для того щоб знайти такий, який виправить виняток, що призводив до краху. Автори статті [64] запропонували виявлення помилок, що призводять до падіння Android застосунків у смартфонах.

Автори статті [65] виявляють помилки, що призводять до переповнення буфера у процесі виконання програми, далі вони отримують джерело помилки за допомогою ProPolice [66]; потім вони використовують рукописні правила перетворення коду, написані мовою перетворення TXL для зміни вихідного коду. Регресії виявляють надані вручну тестові набори.

У статті [67] намагаються генерувати виправлення вихідного коду з працюючого експлойту, що використовує переповнення масиву у коді написаному мовою програмування C. Оператори виправлення полягають у виправленні читання поза меж масиву шляхом додавання модулів у вирази читання і операцій запису поза межі масиву шляхом обрізання даних, що повинні бути записаними.

Автори статті [68] спробували виконувати автоматичні виправлення цілочислених переповнень. У них є три оператори виправлення. Перший-примусовий перехід на гілку посилки до того, як відбудеться переповнення, другий – примусовий перехід на гілку помилки після того, як відбулось переповнення, а останній – це вихід з програми. Створені умови являються умовами шляху, що отримані із динамічного символьного виконання.

1.3.6 Інші валідатори

Інші специфічні валідатори, що були використані в настройках автоматичного виправлення помилок безпеки.

Було запропоновано ряд методів для виправлення помилок паралелізму. У статті [69] було представлено AFix: модель виправлення Afix заснована на введені інструкцій в критичні області. Це дослідження автоматичного виправлення помилок паралелізму отримала подальший розвиток у дослідженні наведеному у статті [70]. У статті [71] було запропоновано інший оператор виправлення помилок паралелізму в інструменті, що має назву HFix: вони пропонують автоматично додавати операції об'єднання потоків.

Автори статті [72] представили підхід для автоматичного виправлення помилок у веб-застосунках написаних мовою програмування PHP, що генерують HTML теги. Валідатором є перевірка чи являється вихідна HTML строка деформованою, наприклад, якщо вона не містить непослідовної послідовності тегів, що відкриваються і закриваються. У статті [73] також виправляється вихід HTML коду за допомогою PHP коду, використовуючи трасування під час виконання замість вирішення обмежень. Автори статті [74] також виправляють помилки у веб-застосунках, але розглядає SQL-ін'єкції і їх оператор виправлення складається з завершення певного виклику функції.

У статті [75] автори використовують в якості валідатора написаний вручну звіт про помилки безпеки. Параметризовані шаблони виправлення помилок безпеки заповнюються змінними зі звіту. Наприклад, для ненульового шаблона перевірки вони беруть ім'я змінної, яку слід перевіряти, зі звіту.

Автори статті [76] використовують перевірку для ML програм на основі випробувань використовуючи Isabel у якості валідатора. Коли випробування не вдається, приклад, що використовувався для випробування визначає підхід для виправлення на основі шаблонів (замінюючи один виклик методу іншим, додаючи якийсь код).

У статті [77] запропоновано використовувати метаморфічні відносини у якості валідатора виправлень. Вони оцінюють свій підхід за бенчмарком Introclass, який складається зі студентських програм. Через обмежений розмір експериментальних суб’єктів, досі не доведеною, що метаморфічні відносини можуть допомогти у виправлені помилок безпеки у великих реальних програмах.

Висновки до розділу 1

У цьому розділі були розглянуті основні підходи виправлення помилок безпеки у коді програмного забезпечення. Вони засновуються на вихідних даних різних валідаторів – засобів, що виявляють помилки безпеки та перевіряють виправлення. Більшість існуючих підходів направлені на виправлення помилок безпеки пов’язаних з переповненням буферів чи виходом за межі масивів для різних мов програмування. Але для мов програмування C/C++ більшість існуючих рішень виправляють лише стилістичні помилки (наприклад, відсутність знаку «;» у кінці строки), що виникають при компіляції програми.

2 РОЗРОБКА МЕТОДУ ВИПРАВЛЕННЯ ПОМИЛОК БЕЗПЕКИ

2.1 Метод представлення початкового коду

У сучасному світі інформаційних технологій існує багато різних способів представлення початкового коду для роботи з ним, зокрема можна працювати безпосередньо з початковим кодом, або з його бінарним представленням, або з різними представленнями у вигляді дерев та графів. У цій роботі основним методом представлення початкового коду є Абстрактне Синтаксичне Дерево (АСД).

Абстрактне синтаксичне дерево – це спосіб представлення синтаксису мови програмування у якості ієрархічної деревоподібної структури. Ця структура використовується для генерації таблиць символів для компіляторів та подальшого генерування коду. Дерево представляє всі конструкції в мові та їх правила.

Абстрактне синтаксичне дерево представляє всі синтаксичні елементи мови програмування, подібні до синтаксичних дерев, що використовують мовознавці для людських мов. Дерево фокусується переважно на правилах, а не на елементах, таких як дужки або крапки з комою, які закінчують речення в деяких мовах. Ієрархія дерева складається з елементів програмного коду розбитих на дрібні частини. Наприклад, дерево для умовного оператора містить правила для змінних, що йдуть від потрібного оператора [78].

На Рисунку 2.1 зображено приклад абстрактного синтаксичного дерева для циклу:

```
while (x<20)
{
        x = x+y*2;
}
```

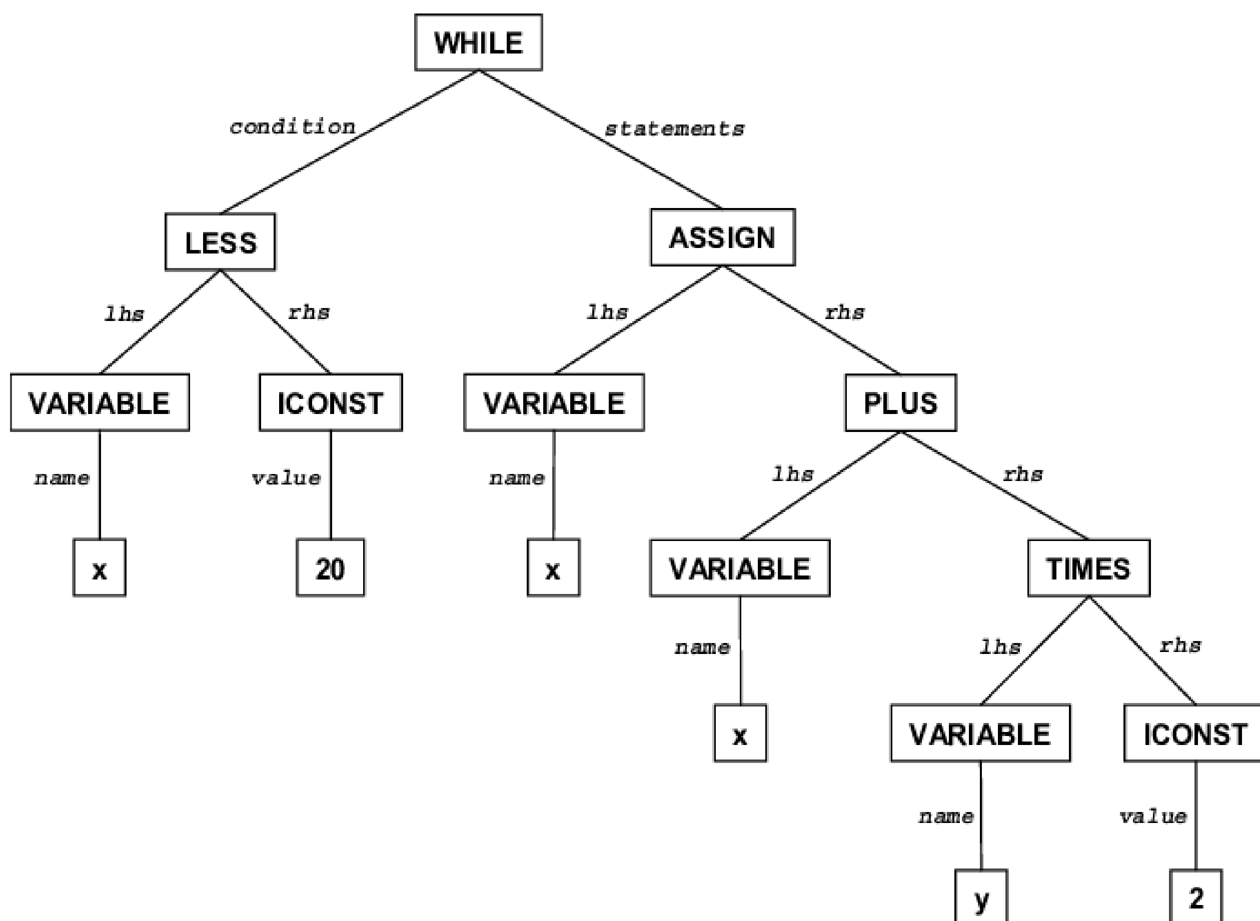

Рисунок 2.1 – Приклад АСД для звичайного циклу while

2.2 Класифікація помилок безпеки

Проаналізувавши велику кількість різних типових помилок безпеки було вирішено зупинитись на класах CWE – 119, CWE – 369 та CWE - 399.

CWE (Common Weakness Enumeration) - це загальнодоступний перелік поширених помилок безпеки програмного забезпечення. Він слугує загальною мовою для опису помилок безпеки програмного забезпечення, спрямованих на ці помилки, і є базовим стандартом для виявлення, пом'якшення та запобігання помилок безпеки програмного забезпечення [79].

CWE – 119 – клас помилок безпеки програмного забезпечення пов’язаний з неправильними обмеженнями операцій в межах буфера пам’яті. Деякі мови програмування дозволяють пряму адресацію місць в пам’яті і автоматично не гарантують, що ця частина пам’яті доступна для буфера пам’яті на який посилаються. Це може призвести до того, що операції зчитування чи запису будуть виконуватися у тих місцях пам’яті, які можуть бути пов’язані з іншими змінними, структурами даних або внутрішніми програмними даними. У результаті цього зловмисник може мати можливість виконувати довільний код, змінювати конкретний потік управління, читати конфіденційну інформацію або спричиняти збій системи.

CWE – 369 – клас помилок безпеки програмного забезпечення пов'язаний з діленням на нуль. Зазвичай ці помилки виникають, коли продукту передається несподіване значення або якщо виникає помилка, яка неправильно виявлена. Це часто трапляється при обчисленнях за фізичними розмірами, такими як розмір, довжина, ширина та висота.

CWE – 399 – клас помилок безпеки програмного забезпечення пов’язаний з помилками управління ресурсами. Неправильне управління ресурсів, таке як подвійне закриття ресурсів або подвійне вивільнення змінних, може призвести до неочікуваної поведінки програмного забезпечення.

У Таблиці 2.1 можна ознайомитись з помилками безпеки для яких було розроблено методи виправлення у ході роботи.

Таблиця 2.1 – Помилки безпеки, що розглядаються

Клас CWE	Тип помилки безпеки
CWE - 119	Out of bounds
CWE - 369	Division by zero

Продовження таблиці 2.1

Клас CWE	Тип помилки безпеки
CWE - 399	Double free resources, double closed resources, string buffer overflow, not closed resources, usage resource after free, usage resource after close

2.3 Теоретичні відомості

2.3.1 Нейронна мережа

Алгоритм навчання штучної нейронної мережі, або нейронна мережа - це обчислювальна система навчання, яка використовує мережу функцій для розуміння та переведення введених даних однієї форми в потрібний вихід, як правило, в іншу форму. Концепція штучної нейронної мережі була натхненна біологією людини та способом взаємодії нейронів людського мозку для розуміння входів людських сенсорів.

Нейронні мережі - лише один із багатьох інструментів та підходів, що застосовуються в алгоритмах машинного навчання. Сама нейронна мережа може використовуватися як фрагмент у багатьох різних алгоритмах машинного навчання для обробки складних входів даних у простір, який комп'ютери можуть зрозуміти.

Сьогодні нейронні мережі застосовуються до багатьох проблем із реального життя, включаючи розпізнавання мови та зображень, фільтрацію спаму на електронній пошті, у фінансах та медичній діагностиці.

Алгоритми машинного навчання, що використовують нейронні мережі, зазвичай не потрібно програмувати за допомогою конкретних правил, які визначають, чого очікувати від вхідних даних. Алгоритм навчання нейронної мережі натомість навчається на обробці багатьох розмічених прикладів (тобто даних з "відповідями"), які постачаються під час навчання, та за допомогою цієї відмітки дізнаються, які характеристики вводу потрібні для побудови правильного результату. Після опрацювання достатньої кількості прикладів нейронна мережа може почати обробляти нові, небачені входи та успішно повертати точні результати. Чим більше прикладів та різноманітності входів бачить програма, тим точнішими стають результати, оскільки програма вчиться з досвідом.

Цю концепцію найкраще зрозуміти на прикладі. Уявіть собі "просту" проблему спроби визначити, чи містить зображення кішку чи ні. Хоча людині це досить легко зрозуміти, набагато складніше навчити комп’ютер ідентифікувати кішку на зображенні за допомогою класичних методів. Враховуючи різноманітні можливості того, як кішка може виглядати на малюнку, писати код для врахування кожного сценарію майже неможливо. Але за допомогою машинного навчання, а точніше нейронних мереж, програма може використовувати узагальнений підхід до розуміння вмісту зображення. Використовуючи кілька шарів функцій для розкладання зображення на точки даних та інформацію, якими може користуватися комп’ютер, нейронна мережа може почати визначати тенденції, що існують у багатьох прикладах, які вони обробляють та класифікують зображення за їх подібністю.

Опрацювавши під час тренування багато прикладів зображень котів, алгоритм має модель того, які елементи та їх відповідні відносини на зображенні важливо враховувати, вирішуючи, присутній кіт на малюнку чи ні. Оцінюючи нове зображення, нейронна мережа порівнює дані про нове зображення зі своєю моделлю, яка базується на всіх попередніх оцінках. Потім вона використовує

просту статистику, щоб визначити, чи містить зображення кота чи ні, виходячи з того, наскільки воно відповідає моделі.

У наведеному прикладі шари функцій між входом і виходом - це те, що складає нейронну мережу. На практиці нейронна мережа трохи складніша. На рисунку 2.2 наведено взаємодію між шарами, але потрібно, що існує багато варіацій взаємозв'язків між вузлами або штучними нейронами [80].

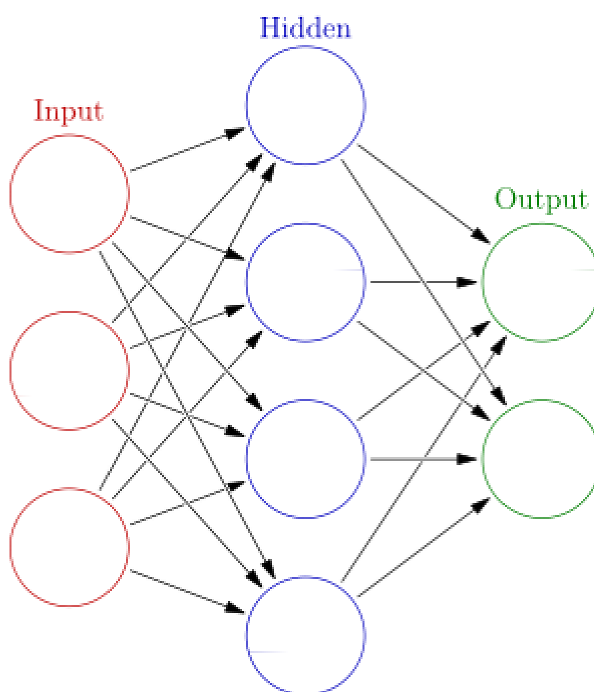

Рисунок 2.2 – Взаємозв’язки між вузлами нейронної мережі [80]

2.3.2 Рекурентна нейронна мережа

Рекурентна нейронна мережа (RNN) - це тип нейронної мережі, де вихід з попереднього кроку подається як вхід до поточного кроку. У традиційних нейронних мережах всі входи та виходи незалежні один від одного, але у випадках, коли потрібно передбачити наступне слово речення, потрібні

попередні слова, і тому виникає необхідність запам'ятовувати попередні слова. Таким чином, виникла RNN, яка вирішила це питання за допомогою прихованого шару. Основна і найважливіша особливість RNN - прихований стан, який запам'ятовує певну інформацію про послідовність.

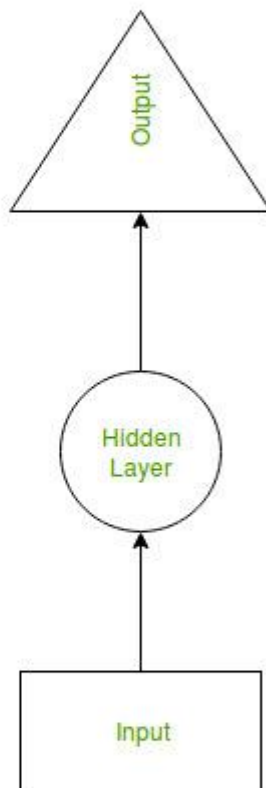

Рисунок 2.3 – Взаємозв’язок між входом та виходом у RNN через прихований шар [81]

Рекурентні нейронні мережі мають "пам'ять", яка запам'ятовує всю інформацію про те, що було обчислено. Вони використовують однакові параметри для кожного входу, оскільки виконують одне і те ж завдання на всіх входах або прихованих шарах для отримання виходу. Це зменшує складність параметрів, на відміну від інших нейронних мереж [81].

2.3.3 LSTM

Нейронні мережі LSTM, які розшифровуються як **L**ong **S**hort-**T**erm **M**emory, є особливим типом періодичних нейронних мереж, які останнім часом привернули багато уваги у спільноті машинного навчання.

Простими словами мережі LSTM мають деякі клітини внутрішнього контекстуального стану, які діють як комірки довгострокової або короткострокової пам'яті.

Вихід мережі LSTM модулюється станом цих комірок. Це дуже важлива властивість, коли нам потрібно, щоб прогнозування нейронної мережі залежало від історії контекстів вхідних даних, а не лише від останнього вводу.

Нейронним мережам LSTM вдається зберігати контекстну інформацію входів, інтегруючи цикл, який дозволяє інформації переходити від одного кроку до іншого. Наприклад, коли ви читаєте якусь публікацію, ви розумієте кожне слово, виходячи з вашого розуміння попередніх слів. Ви не кидаєте все і починаєте думати з нуля при кожному слові. Аналогічно, прогнози LSTM завжди зумовлені минулим досвідом вхідних даних мережі.

З іншого боку, чим більше часу проходить, тим менше ймовірність, що наступний вихід залежить від дуже старого входу. Ця відстань від часу сама по собі є також контекстною інформацією, яку потрібно засвоїти. Мережі LSTM керують цим, вивчаючи, коли пам’ятати, а коли забувати, через їх ваги [82].

2.3.4 GRU

Для вирішення проблеми зникаючих градієнтів, що часто виникають під час роботи основної рекурентної нейронної мережі, було розроблено багато варіантів. Однією з найвідоміших варіацій є мережа довготривалої пам’яті (LSTM). Трохи менш відомою, але настільки ж ефективним варіантом є мережа вентильних рекурентних вузлів (GRU).

На відміну від LSTM, вона складається лише з трьох воріт і не підтримує внутрішній стан. Інформація, що зберігається у внутрішньому стані клітини у періодичній одиниці LSTM, вкладається у прихований стан GRU. Ця колективна інформація передається до наступного GRU.

Основний принцип роботи GRU аналогічний принципу роботи базової рекурентної нейронної мережі, основна відмінність між ними полягає у внутрішній роботі всередині кожного періодичного блоку, оскільки мережі GRU, що повторюються, складаються з воріт, які модулюють поточний вхід і попередній прихований стан [83].

2.3.5 Sequence-to-sequence

Перш за все, ідея даної роботи полягає в заміні поганого коду на хороший код або в додаванні якогось коду, щоб поганий код став хорошим кодом, тому нам потрібно замінити частину тексту на інший.

Для повного розуміння логіки моделі машинного навчання sequence-to-sequence слід розглянути рисунок 2.4.

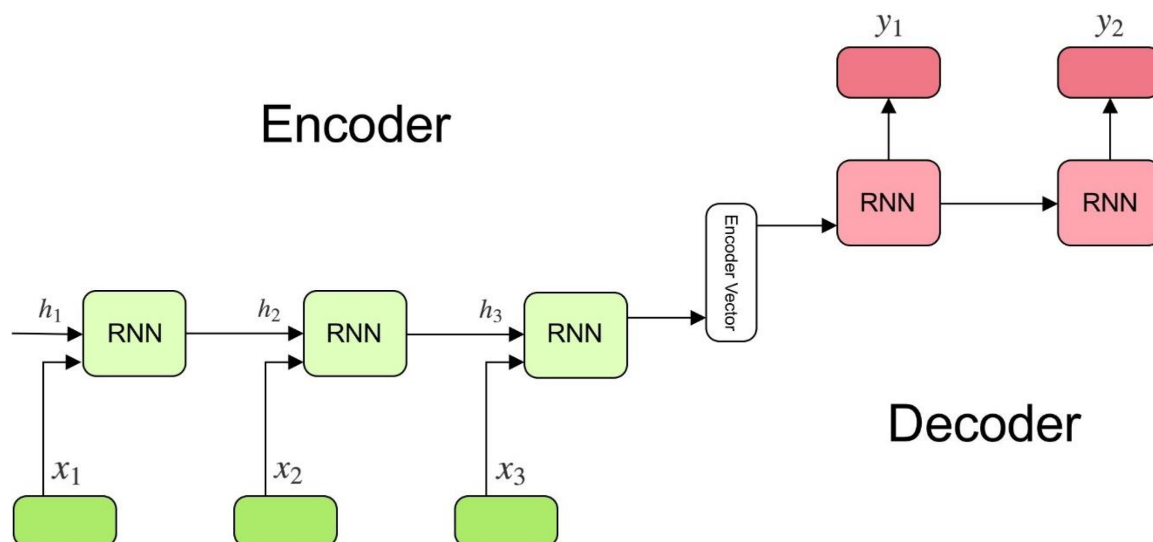

Рисунок 2.4 – Архітектура моделі машинного навчання sequence-to-sequence

Модель складається з 3 частин: енкодеру, проміжний (енкодер) вектор і декодеру.

Енкодер:

- Стек з декількох повторюваних одиниць (LSTM або GRU комірки для кращої продуктивності), де кожен приймає один елемент вхідної послідовності, збирає інформацію для цього елемента і поширює його далі.
- У проблемі відповіді на запитання послідовність введення - це сукупність усіх слів із запитання. Кожне слово представлене як x_i, де i - порядок цього слова.
- Приховані стани h_i обчислюються за формулою:

$$h_t = f(W^{(hh)}h_{t-1} + W^{hx}x_t) \tag{2.1}$$

Ця проста формула представляє результат звичайної рекурентної нейронної мережі. Як можна побачити, тут просто застосовуються відповідні ваги до попереднього прихованого стану h_{t-1} та вхідного вектора x_t.

Проміжний вектор:

- Це остаточний прихований стан, що утворюється з частини енкодера моделі. Він розраховується за формулою 2.1.
- Цей вектор спрямований на інкапсуляцію інформації для всіх вхідних елементів, щоб допомогти декодеру зробити точні прогнози.
- Він виступає в якості початкового прихованого стану декодерної частини моделі.

Декодер:

- Стек з декількох повторюваних одиниць, де кожен прогнозує вихід y_t на етапі часу *t*.
- Кожна повторювана одиниця приймає прихований стан від попереднього блоку і вихід, також є прихованим станом.
- У проблемі відповіді на запитання, вихідна послідовність - це сукупність усіх слів із відповіді. Кожне слово представлено у вигляді y_i, де i є порядком цього слова.
- Будь-який прихований стан h_i обчислюється за формулою:

$$h_t = f(W^{(hh)} h_{t-1})$$

- Вихід y_t на етапі часу *t* обчислюється за формулою:

$$y_t = softmax(W^S h_t)$$

Виходи обчислюються використовуючи прихований стан на поточному кроці часу разом із відповідною вагою W^S. Softmax використовується для створення вектора ймовірностей, який допоможе нам визначити кінцевий результат (наприклад, слово в задачі на відповідь на питання).

Сильна сторона цієї моделі машинного навчання полягає в тому, що вона може відображати послідовності різної довжини один до одного. Входи та виходи не співвідносяться, і їх довжина може відрізнятися. Це відкриває цілий

новий спектр проблем, які тепер можна вирішити за допомогою такої архітектури [84].

2.4 Архітектура розроблених методів виправлення помилок безпеки

2.4.1 Підхід на основі правил

Для виправлення помилок безпеки програмного забезпечення наведених у Таблиці 2.1 спочатку було запропоновано розробити підхід на основі рукописних правил.

На Рисунку 2.5 зображено послідовність кроків у роботі запропонованого методу:

- Аналіз коду. На цьому кроці відбувається перевірка початкового коду вхідного проекту статичним та динамічним аналізаторами.
- Створення файлу конфігурації. На основі результатів перевірки початкового коду сформовується спеціальний конфігураційний файл, що містить метадані про положення знайденої помилки у початковому файлі.
- Препроцесинг початкового коду. На цьому кроці відбувається побудова абстрактного синтаксичного дерева, використовуючи бібліотеку clang для мови програмування Python. Витягається вся необхідна інформація для кожного токена в АСД.
- Створення патчу. Після попередньої обробки початкового файлу та основуючись на даних про помилку з файла конфігурації відбувається створення патчу. Після чого він приміняється до початкового файлу.

- Повторний аналіз коду. На цьому кроці відбувається повторна перевірка початкових файлів з застосованими до них патчами на тих самих аналізаторах, що були на початку.
- Обробка результатів аналізу. Базуючись на результатах повторного аналізу коду, якщо помилку безпеки було виправлено і не створено нових, відбувається пропозиція патча.

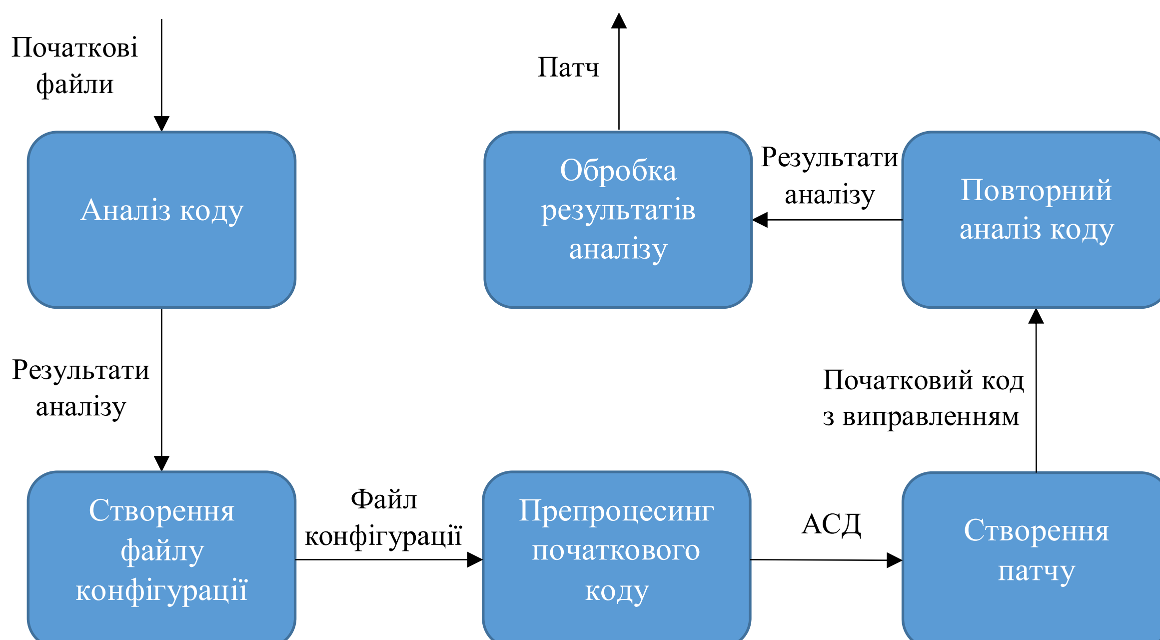

Рисунок 2.5 – Етапи роботи методу автоматичного виправлення помилок безпеки на основі правил

2.4.2 Підхід на основі глибинного навчання

Після виправлень помилок безпеки за допомогою підходу на основі правил стало зрозуміло, що створення та застосування шаблонів є одноманітними та потребує великої інтенсивної ручної праці, тому ми вирішили автоматизувати збір та застосування шаблонів, заснованих на правилах. Згідно з процедурою

Microsoft SDL, можливо, існують приклади хорошого та поганого використання деяких цільових викликів, тому згідно з цими прикладами ми можемо генерувати деякий набір даних, що буде підходити для нейронних мереж.

Потрібно створити застосунок, який може зрозуміти контекст коду та перетворити його на хороший код, тому було вирішено використати метод, який перекладає речення з однієї мови на іншу з деякими модифікаціями. Одним із стандартних методів перекладу з однієї мови на іншу є модель seq2seq (модель енкодера-декодера).

Для створення більш якісної моделі потрібно додати механізм уваги [85]. Основна ідея механізму уваги полягає у встановленні прямих коротких зв’язків між виходом та входом, звертаючи «увагу» на відповідний вміст входу під час перекладу.

У простій моделі seq2seq ми передаємо останній стан з енкодера до декодера при запуску процесу декодування. Це добре працює для коротких і середніх речень; але для довгих речень один прихований стан фіксованого розміру стає вузьким місцем інформації. Замість того, щоб відкидати всі приховані стани, обчислені у вихідному RNN, механізм уваги забезпечує підхід, який дозволяє декодеру спостерігати за ними (трактуючи їх як динамічну пам'ять вихідної інформації). Тим самим механізм уваги покращує переклад довших речень. Нині механізми уваги є стандартом де-факто і їх успішно застосовують у багатьох інших завданнях (включаючи створення підписів зображень, розпізнавання мови та узагальнення тексту).

Обчислення уваги відбувається на кожному кроці часу декодера. Він складається з наступних етапів:

- Поточний прихований стан порівнюється з усіма вихідними станами для отримання вагів уваги.
- Виходячи з вагів уваги, ми обчислюємо контекстний вектор як середньозважене середнє значення вихідних станів.
- Вектор контексту комбінується з поточним прихованим станом, щоб отримати кінцевий вектор уваги.

- Вектор уваги подається як вхід до наступного за часом етапу.

На рисунку 3.2 зображено послідовність кроків у роботі запропонованого методу:

- Аналіз коду. На цьому кроці відбувається перевірка початкового коду вхідного проекту статичним аналізатором.
- Добування код-гаджетів. На основі результатів перевірки початкового коду добуваються код-гаджети, що містять найбільш істотну інформацію про помилку безпеки.
- Sequence-to-sequence. На цьому кроці відбувається трансформація код-гаджету, що виправляє помилку безпеки, за допомогою sequence-to-sequence моделі.
- Створення патчу. Після виправлення помилки безпеки у код-гаджеті, він додається у АСД на місце де був код-гаджет з помилкою.
- Повторний аналіз коду. На цьому кроці відбувається повторна перевірка початкових файлів з приміненими до них патчами на тих самих аналізаторах, що були на початку.
- Обробка результатів аналізу. Базуючись на результатах повторного аналізу коду, якщо помилку безпеки було виправлено і не створено нових, відбувається пропозиція патча.

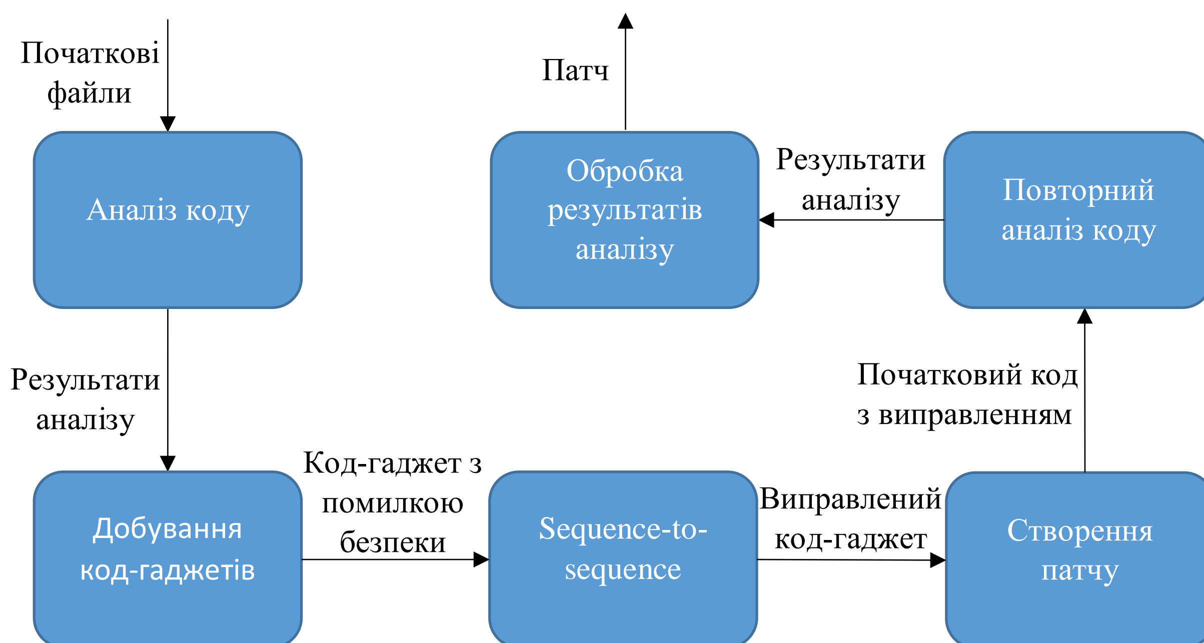

Рисунок 2.6 - Етапи роботи методу автоматичного виправлення помилок безпеки на основі глибинного навчання

Висновки до розділу 2

У даному розділі було запропоновано дві технології виправлення помилок безпеки в програмному забезпеченні засновані на фіксованих рукописних шаблонах та на глибинному навчанні з використанням моделі машинного навчання sequence-to-sequence.

У розділі розглядаються основні базові поняття нейронних мереж та використані моделі машинного навчання, методи представлення початкового коду та класифікація помилок безпеки.

Для кожної з запропонованих технологій наведено етапи їх роботи та проміжні дані після кожного етапу роботи.

3 АНАЛІЗ ЕФЕКТИВНОСТІ РОЗРОБЛЕНОЇ ТЕХНОЛОГІЇ

3.1 Архітектура та програмна реалізація запропонованих методів

На Рисунку 3.1 зображено діаграму класів UML програмної реалізації запропонованого підходу на основі правил. На ній зображені такі статичні елементи: класи, їх відношення, їх типи даних та їх функції.

Рисунок 3.1 – Діаграма класів програмної реалізації запропонованого підходу на основі правил

На Рисунку 3.2 зображена діаграма послідовностей для програмної реалізації запропонованого підходу на основі правил. На ній відображені взаємодії між об’єктами та повідомлення якими вони обмінюються впродовж генерації виправлень для помилок безпеки у початковому коді.

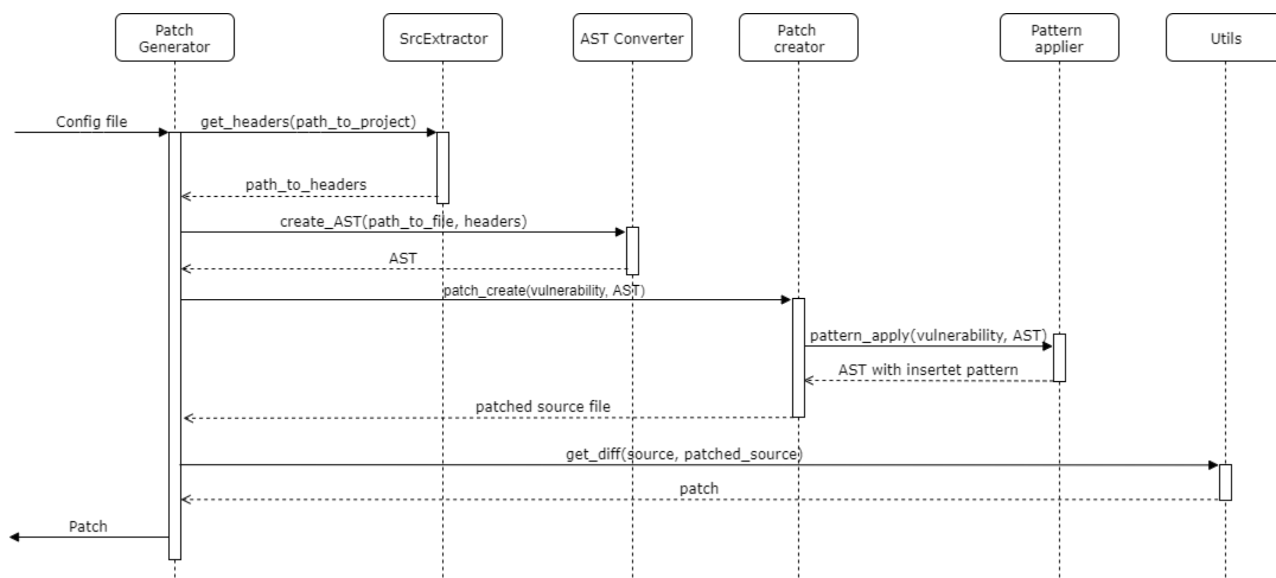

Рисунок 3.2 – Діаграма послідовностей для програмної реалізації запропонованого підходу на основі правил

На Рисунках 3.3 – 3.7 зображені шаблони, які використовуються для виправлення помилок безпеки у програмному коді, що наведені у Таблиці 2.1. Наведені шаблони є узагальненням типових людських патчів, що використовувалися для виправлення помилок безпеки у реальних проектах. Для найбільш легкого вставлення до АСД початкового коду програми з помилкою безпеки, шаблони були написані у спеціальному форматі з використанням:

- загальних змінних: ARG_1, ARG_2, ARG_3, ARG_4, ARG_5, ARG_6
- загальних типів: TYPE_1, TYPE_2, TYPE_3, TYPE_4
- спеціальних змінних: idx_target_ - використовується для зазначення індексу цільової змінної; denom_target_ - використовується для зазначення цільового виразу; len_target_ - використовується для зазначення довжини цільового масиву або строки

- спеціальних символів: R – використовується для позначення строки, якою потрібно замінити строку з помилкою безпеки в початковому АСД; _ - префікс, який використовується для позначення змінної, що необхідно видалити, перед тим як вставляти шаблон виправлення до початкового АСД; X – використовується для позначення будь-якого бінарного оператора.

На Рисунку 3.3 зображено шаблони для виправлення помилок пов'язаних з діленням на нуль.

На Рисунку 3.3 а) показано шаблон виправлення, що використовується у випадку коли заздалегідь оголошеній змінній присвоюється результат операції ділення, де дільником потенційно може бути нуль. Для попередження ділення на нуль, цільовий дільник (ARG_2) присвоюється спеціальній змінній denom_target_ і перевіряється в умовному операторі if, що вона відмінна від нуля.

На Рисунку 3.3 б) показано шаблон виправлення, що використовується у випадку коли щойно оголошена змінна ініціалізується результатом операції ділення, де дільником потенційно може бути нуль. Шаблон схожий з попереднім випадком, за виключенням, що оголошення змінної (TYPE_3 ARG_3;) виноситься за умовний оператор, а в тілі оператора if видаляється процедура оголошення змінної (_TYPE_3).

На Рисунку 3.3 в) показано шаблон виправлення, що використовується у випадку коли результат операції ділення повертається у якості результату функції. Шаблон схожий на випадок а), за виключенням, що у випадку якщо спеціальна змінна denom_target_ буде проініціалізована нулем, то результатом функції буде повернено нуль (return 0;).

```
1 TYPE_4 denom_target_ = ARG_2;
2 if (denom_target_ != 0) {
3     R ARG_3 = ARG_1 X denom_target_;
4 }
```

а)

```
1 TYPE_3 ARG_3;
2 TYPE_4 denom_target_ = ARG_2;
3 if (denom_target_ != 0) {
4     _TYPE_3 ARG_3 =  ARG_1 X denom_target_;
5 }
```

б)

```
1 TYPE_4 denom_target_ = ARG_2;
2 if (denom_target_ != 0) {
3     return ARG_1 X denom_target_;
4 }
5 return 0;
```

в)

Рисунок 3.3 – Шаблони для виправлення помилок типу «division by zero»

На Рисунку 3.4 зображено шаблони для виправлення помилок пов’язаних з виходом за межі у масивах.

На Рисунку 3.4 а) показано шаблон виправлення, що використовується у випадку коли заздалегідь оголошеній змінній присвоюється значення одного з елементів масиву, або у якості значення одного з елементів масиву приймається заздалегідь оголошена змінна. Для попередження виходу за межі масиву попередньо обраховується довжина масиву і записується у спеціальну змінну len_target_, а індекс необхідного елемента записується у спеціальну змінну idx_target_, яка потім перевіряється в умовному операторі if на те, що вона лежить в межах від 0 до len_target_.

На Рисунку 3.4 б) показано шаблон виправлення, що використовується у випадку коли щойно оголошена змінна ініціалізується значенням одного з елементів масиву, індекс якого ARG_2 може не входити в межі від 0 до len_target_. Шаблон схожий на попередній випадок, за виключенням, що оголошення змінної (TYPE_3 ARG_3;) виноситься за умовний оператор, а в тілі оператора if видаляється процедура оголошення змінної (_TYPE_3).

На Рисунку 3.4 в) показано шаблон виправлення, що використовується у випадку коли результат операції ділення повертається у якості результату функції. Шаблон схожий на випадок а), за виключенням, що у випадку якщо спеціальна змінна idx_target_ не буде знаходитись в межах від 0 до len_target_, то результатом функції буде повернено останній елемент масиву ARG_1 (return ARG_1[len_target_ - 1];).

```
1 TYPE_4 len_target_ = sizeof(ARG_1) / sizeof(ARG_1[0]);
2 TYPE_2 idx_target_ = ARG_2;
3 if ((idx_target_ < len_target_) && (idx_target_ >= 0)) {
4     R ARG_3 = ARG_1[idx_target_]; // Or ARG_1[idx_target_] = ARG_3;
5 }
```

а)

```
1 TYPE_3 ARG_3;
2 TYPE_4 len_target_ = sizeof(ARG_1) / sizeof(ARG_1[0]);
3 TYPE_2 idx_target_ = ARG_2;
4 if ((idx_target_ < len_target_) && (idx_target_ >= 0)) {
5     _TYPE_3 ARG_3 =  ARG_1[idx_target_];
6 }
```

б)

```
1 TYPE_4 len_target_ = sizeof(ARG_1) / sizeof(ARG_1[0]);
2 TYPE_2 idx_target_ = ARG_2;
3 if ((idx_target_ < len_target_) && (idx_target_ >= 0)) {
4    return ARG_1[idx_target_];
5 }
6 return ARG_1[len_target_ - 1];
```

в)

Рисунок 3.4 – Шаблони для виправлення помилок типу «out of bound»

На Рисунку 3.5 зображено шаблони для виправлення помилок пов'язаних з подвійним закриттям ресурсів.

На Рисунку 3.5 показано шаблон виправлення, що використовується у випадку коли важко відстежити чи існує попереднє закриття цільового ресурсу ARG_2. Для попередження повторного закриття ресурсу ARG_2 у шаблоні зображеному на Рисунку 3.5 а), його існування перевіряється умовним оператором if, і якщо ця умова істинна, то відбувається закриття ресурсу. У шаблоні зображеному на Рисунку 3.5 б) для попередження повторного закриття ресурсу ARG_2 попередньо перевіряємо його виразом assert(ARG_2), для того, щоб розробник програмного забезпечення під час зневадження отримав помилку перед повторним закриттям ресурсу.

```
1 if (ARG_2) {
2     CLOSE(ARG_2);
3 }
```

а)

```
1 assert(ARG_2);
2 CLOSE(ARG_2);
```

б)

Рисунок 3.5 – Шаблони для виправлення помилок типу «double closed resources»

На Рисунку 3.6 зображено шаблон для виправлення помилок пов'язаних з подвійним вивільненням ресурсів. Для попередження повторного вивільнення ресурсу ARG_5 у початковому АСД відбувається видалення другого (повторного) вивільнення ресурсів _free(ARG_5).

```
1 free(ARG_5)
2 _free(ARG_5)
```

Рисунок 3.6 – Шаблон для виправлення помилок типу «double free resources»

На Рисунку 3.7 зображено шаблон для виправлення помилок пов’язаних з переповненням буферу під час копіювання строк. Для попередження даної помилки потрібно замість небезпечної функції strcpy(ARG_5, ARG_6), що може викликати переповнення буферу у разі якщо sizeof(ARG_5) <= sizeof(ARG_6),

потрібно використовувати безпечну функцію strncpy(), яка у якості третього параметра приймає кількість символів, що необхідно скопіювати. За кількість символів у наведеному шаблоні для безпечного спрацьовування приймається число на одиницю менше ніж довжина строки в яку буде відбуватись копіювання.

```
1 strncpy(ARG_5, ARG_6, sizeof(ARG_5) - 1);
```

Рисунок 3.7 – Шаблон для виправлення помилок типу «string buffer overflow»

Наступним кроком, після розробки підходу на основі правил, було запропоновано розробити технологію генерації виправлень помилок безпеки за допомогою алгоритмів машинного навчання. Для цього було використано sequence-to-sequence TensorFlow [86] модель, що використовується для мовних перекладів, і додано механізм уваги з вагами.

На Рисунку 3.8 зображено деталі обраної моделі глибинного навчання енкодера-декодера з вагами. На рисунку видно, що до кожного вхідного слова, механізмом уваги, присвоюється спеціальна вага, яка потім використовується декодером для передбачення наступного токена в послідовності.

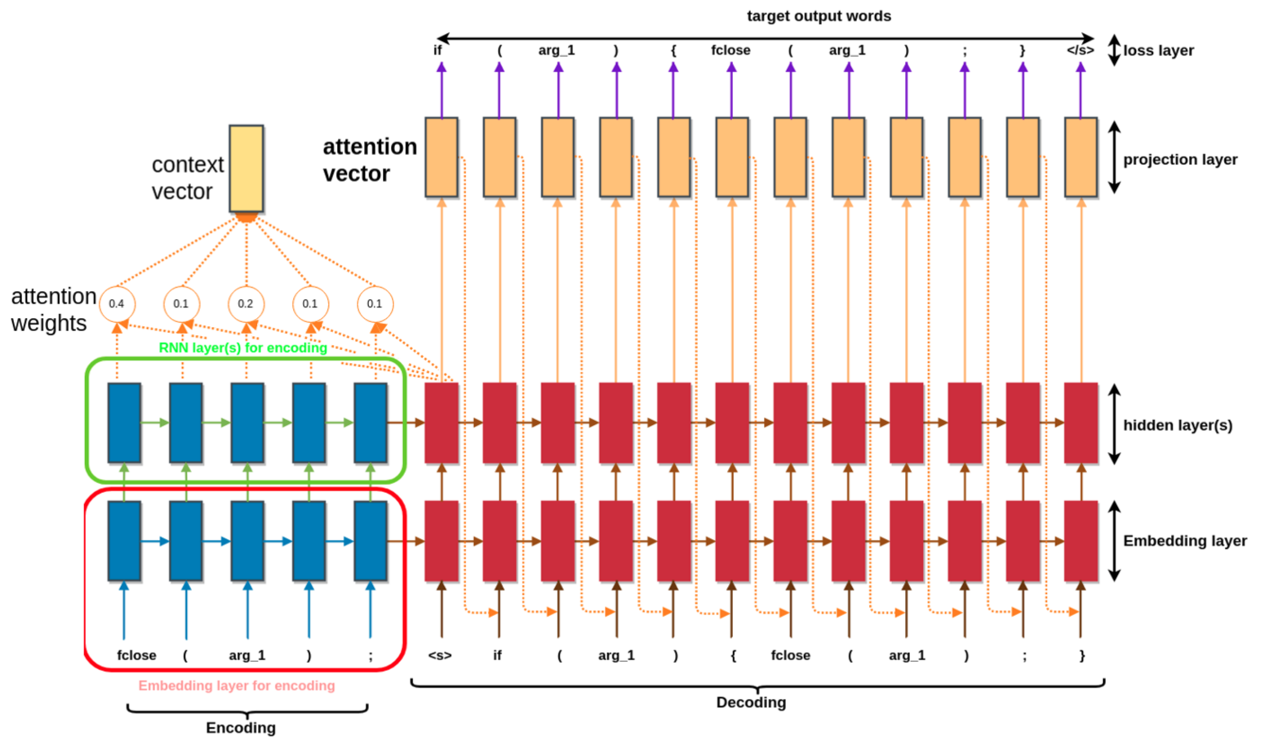

Рисунок 3.8 – sequence-to-sequence модель, що було обрано для підходу на основі глибинного навчання

У обраній моделі були використані наступні формули:

- Ваги уваги:

$$\alpha_{ts} = \frac{\exp(score(h_t,\ \bar{h}_s))}{\sum_{s'=1}^{S} \exp(score(h_t,\ \bar{h}_{s'}))}$$

- Вектор контексту:

$$c_t = \sum_{s} \alpha_{ts} \bar{h}_s$$

- Вектор уваги:

$$\alpha_t = f(c_t, h_t) = \tanh(W_c[c_t; h_t])$$

$$score(h_t, \bar{h}_s) = \begin{cases} h_t^{\mathrm{T}} W \bar{h}_s \\ \vartheta_t^{\mathrm{T}} \tanh(W_1 h_t + W_2 \bar{h}_s) \end{cases}$$

На Рисунку 3.9 зображено програмний код реалізації енкодера:

```
class Encoder(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, enc_units, batch_sz):
    super(Encoder, self).__init__()
    self.batch_sz = batch_sz
    self.enc_units = enc_units
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(self.enc_units,
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_initializer='glorot_uniform')

  def call(self, x, hidden):
    x = self.embedding(x)
    output, state = self.gru(x, initial_state = hidden)
    return output, state

  def initialize_hidden_state(self):
    return tf.zeros((self.batch_sz, self.enc_units))
```

```
encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)

# sample input
sample_hidden = encoder.initialize_hidden_state()
sample_output, sample_hidden = encoder(example_input_batch, sample_hidden)
print ('Encoder output shape: (batch size, sequence length, units) {}'.format(sample_output.shape))
print ('Encoder Hidden state shape: (batch size, units) {}'.format(sample_hidden.shape))
```

```
Encoder output shape: (batch size, sequence length, units) (64, 15, 1024)
Encoder Hidden state shape: (batch size, units) (64, 1024)
```

Рисунок 3.9 – Програмна реалізація моделі енкодера

На Рисунку 3.10 зображено програмний код реалізації механізму уваги:

```
class BahdanauAttention(tf.keras.Model):
  def __init__(self, units):
    super(BahdanauAttention, self).__init__()
    self.W1 = tf.keras.layers.Dense(units)
    self.W2 = tf.keras.layers.Dense(units)
    self.V = tf.keras.layers.Dense(1)

  def call(self, query, values):
    # hidden shape == (batch_size, hidden size)
    # hidden_with_time_axis shape == (batch_size, 1, hidden size)
    # we are doing this to perform addition to calculate the score
    hidden_with_time_axis = tf.expand_dims(query, 1)

    # score shape == (batch_size, max_length, hidden_size)
    score = self.V(tf.nn.tanh(
        self.W1(values) + self.W2(hidden_with_time_axis)))

    # attention_weights shape == (batch_size, max_length, 1)
    # we get 1 at the last axis because we are applying score to self.V
    attention_weights = tf.nn.softmax(score, axis=1)

    # context_vector shape after sum == (batch_size, hidden_size)
    context_vector = attention_weights * values
    context_vector = tf.reduce_sum(context_vector, axis=1)

    return context_vector, attention_weights
```

```
attention_layer = BahdanauAttention(10)
attention_result, attention_weights = attention_layer(sample_hidden, sample_output)

print("Attention result shape: (batch size, units) {}".format(attention_result.shape))
print("Attention weights shape: (batch_size, sequence_length, 1) {}".format(attention_weights.shape))
```

```
Attention result shape: (batch size, units) (64, 1024)
Attention weights shape: (batch_size, sequence_length, 1) (64, 15, 1)
```

Рисунок 3.10 – Програмна реалізація механізму уваги

На Рисунку 3.11 зображено програмний код реалізації моделі декодера:

```
class Decoder(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, dec_units, batch_sz):
    super(Decoder, self).__init__()
    self.batch_sz = batch_sz
    self.dec_units = dec_units
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(self.dec_units,
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_initializer='glorot_uniform')
    self.fc = tf.keras.layers.Dense(vocab_size)

    # used for attention
    self.attention = BahdanauAttention(self.dec_units)

  def call(self, x, hidden, enc_output):
    # enc_output shape == (batch_size, max_length, hidden_size)
    context_vector, attention_weights = self.attention(hidden, enc_output)

    # x shape after passing through embedding == (batch_size, 1, embedding_dim)
    x = self.embedding(x)

    # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
    x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

    # passing the concatenated vector to the GRU
    output, state = self.gru(x)

    # output shape == (batch_size * 1, hidden_size)
    output = tf.reshape(output, (-1, output.shape[2]))

    # output shape == (batch_size, vocab)
    x = self.fc(output)

    return x, state, attention_weights
```

```
decoder = Decoder(vocab_tar_size, embedding_dim, units, BATCH_SIZE)

sample_decoder_output, _, _ = decoder(tf.random.uniform((64, 1)),
                                      sample_hidden, sample_output)

print ('Decoder output shape: (batch_size, vocab size) {}'.format(sample_decoder_output.shape))
```

```
Decoder output shape: (batch_size, vocab size) (64, 39)
```

Рисунок 3.11 – Програмна реалізація моделі декодера

Для тренування описаної sequence-to-sequence моделі було зібрано датасет зі справжніх людських виправлень типових помилок безпеки. Потім виправлення узагальнюються шляхом перейменування користувацьких змінних та функцій на спеціальні назви, що описані у розділі 3.1. Наступним кроком у підготовці датасету є додавання спеціальних токенів «*start*» та «*end*» на початку та в кінці кожного виправлення. Кожне слово у датасеті отримує унікальний індекс для переведення у числовий вектор. При завантажені датасету для тренування моделі глибинного навчання він трансформується у tf.data датасет. На Рисунку 3.12 показано програмний код створення tf.data датасету з гіперпараметрами.

```
BUFFER_SIZE = len(input_tensor_train)
BATCH_SIZE = 64
steps_per_epoch = len(input_tensor_train)//BATCH_SIZE
embedding_dim = 256
units = 1024
vocab_inp_size = len(inp_lang.word_index)+1
vocab_tar_size = len(targ_lang.word_index)+1

dataset = tf.data.Dataset.from_tensor_slices((input_tensor_train, target_tensor_train)).shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)
print (dataset)
```

Рисунок 3.12 – Програмний код створення tf.data датасету

Процес побудови та тренування описаної моделі:

- Передача «входу» до енкодера, який повертає «вихід енкодера» та «прихований стан енкодера».
- Вихідні дані енкодера разом з входом декодера (токен «start») передаються до декодера.
- Декодер повертає «трансформації» та «прихований стан декодера».
- Прихований стан декодера передається назад до моделі для розрахунку втрат.
- Використовується «teacher forcing» для визначення наступного входу для декодера.
- «Teacher forcing» - це техніка у якій «цільове слово» передається у якості «наступного входу» до декодера.
- Останнім кроком є прорахунок градієнтів та їх застосування до оптимізатора та зворотного поширення помилки

Процес створення виправлення:

- При створенні виправлення кроки схожі на процес тренування, за виключенням, що тут не використовується «teacher forcing». Входом до декодера кожний раз є його попередні трансформації разом із прихованим станом та виходом енкодера.
- Також на кожному кроці зберігаються «ваги».

- Закінчення роботи відбувається коли модель на виході дає токен «end».

3.2 Результати дослідження

Результатом дослідження є дві технології автоматичного виправлення помилок безпеки в програмному забезпеченні та їх програмна реалізація. Перша технологія дозволяє автоматично створювати виправлення використовуючи спеціальні правила та шаблони, що є узагальненням до справжніх людських виправлень з реальних проектів для типових помилок безпеки. Друга технологія дозволяє автоматично створювати виправлення використовуючи модель глибинного навчання (sequence-to-sequence), яка навчається на датасеті з справжніх людських патчів з реальних проектів.

3.2.1 Підхід на основі правил

Запропонована технологія була протестована на різних відкритих проектах написаних на мовах C/C++ у яких були попередньо знайдені помилки безпеки для яких були написані відповідні шаблони для виправлення.

При підготовці проектів до виправлення помилок безпеки їх було проскановано програмою VulDetect [87] і у одному з проектів компанії Microsoft, а саме у Microsoft/Terminal, була знайдена потенційна помилка безпеки при закритті файлу. Небезпечна ситуація могла б виникнути через те, що після команди відкриття файла не виконувалась перевірка чи він був відкритий і якби на закриття прийшов порожній потік, то виникла б неочікувана поведінка програми. На рисунках нижче наведено результати перевірки проекту

Microsoft/Terminal до (Рисунок 3.13) і після (Рисунок 3.14) успішного застосування патчу, що було згенеровано програмною реалізацією запропонованого підходу на основі правил.

Рисунок 3.13 – Виявлена помилка безпеки пов’язана з небезпечним закриттям файлу у проекті Microsoft/Terminal

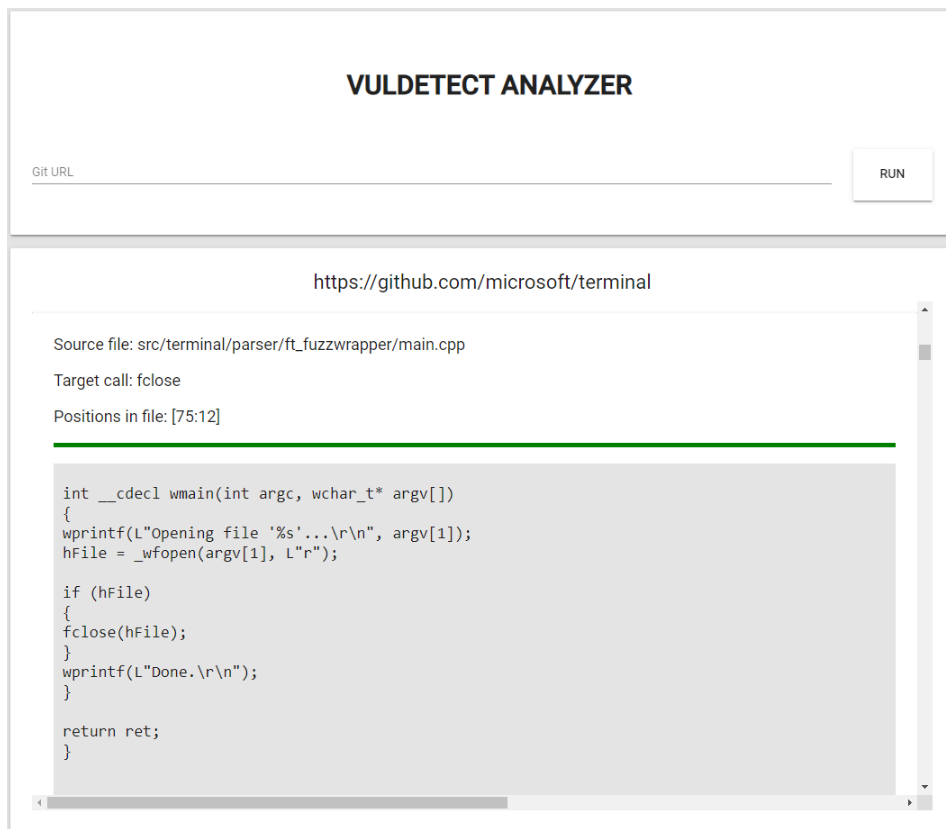

Рисунок 3.14 – Виправлена помилка безпеки пов’язана з небезпечним закриттям файлу у проекті Microsoft/Terminal

Після генерації і перевірки виправлення програмою VulDetect, патч був запропонований розробникам Microsoft і після їхньої перевірки успішно внесений в код продукту Terminal (Рисунок 3.15).

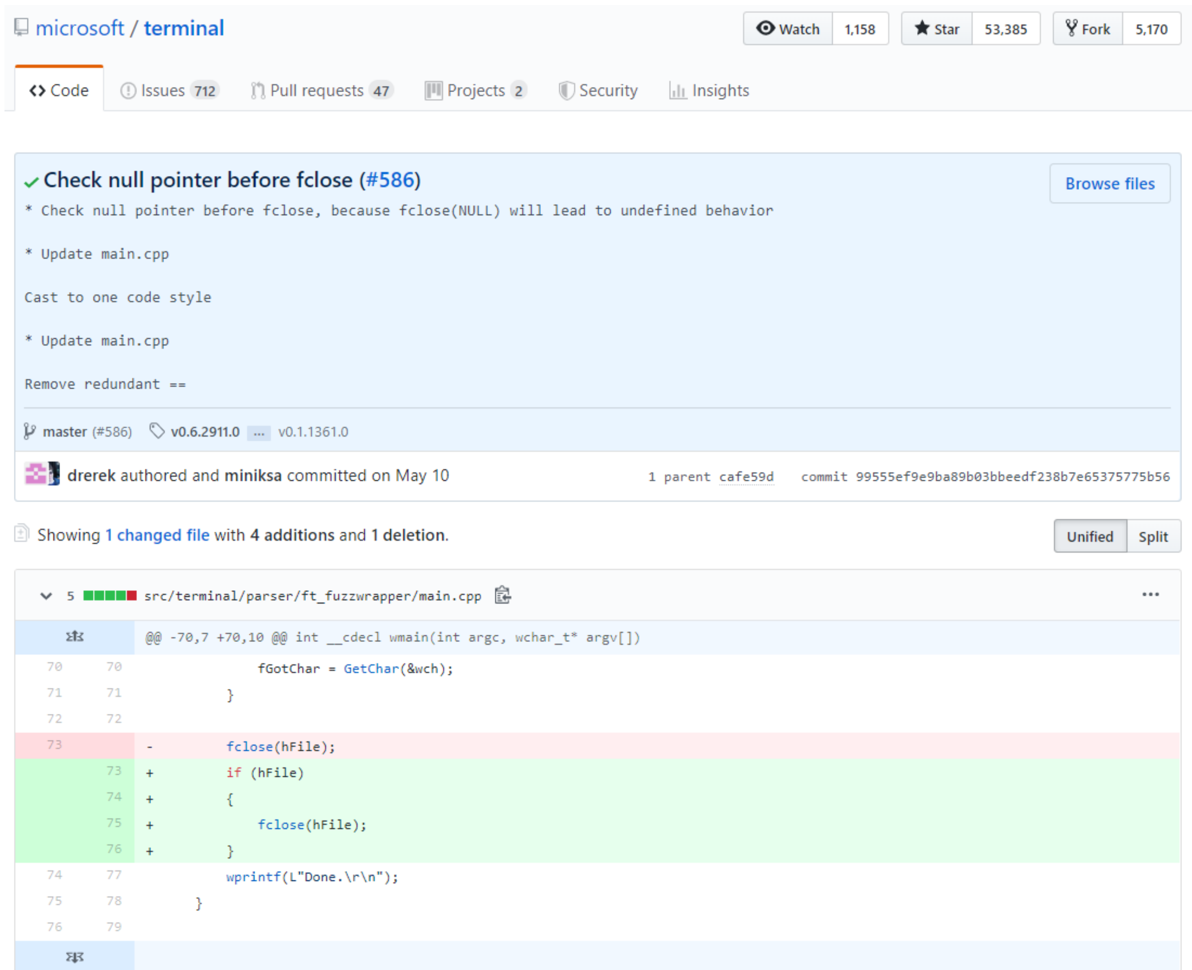

Рисунок 3.15 – патч до програмного продукту Terminal, що був прийнятий розробниками компанії Microsoft [88]

3.2.2 Підхід на основі глибинного навчання

Результатом тестування запропонованого підходу створення виправлень безпеки на основі глибинного навчання за допомогою моделі sequence-to-sequence, коли нейронна мережа сама витягує статистичну залежність між кодом із помилками та виправленим кодом, є згенеровані трансформації коду, що зображені на Рисунках 3.16 – 3.18.

На Рисунку 3.16 можна побачити, що для вхідної послідовності(код-гаджету) «fclose(arg_1);», яка не містить попередньої перевірки вхідного

аргументу, що може призвести до непередбаченої поведінки у разі коли буде помилка зчитування файлу і аргумент буде нульовим, програмна реалізація запропонованої sequence-to-sequence моделі генерує вихідну послідовність «if (arg_1) { fclose (arg_1) ; }» у якій перед закриттям файла проводиться перевірка аргумента.

```
transform(u'fclose(ARG_1);')

Input: <start> fclose ( arg_1 ) ; <end>
Predicted transform:   if ( arg_1 ) { fclose ( arg_1 ) ; } <end>
```

Рисунок 3.16 – Результат виправлення помилки безпеки пов’язаної з небезпечним закриттям файлу, що запропонувала модель на основі глибинного навчання

На Рисунку 3.17 можна побачити, що для вхідної послідовності «file *arg_1 = fopen (arg_2, arg_3);», яка не містить подальшої перевірки аргументу «ARG_1», що може призвести до непередбаченої поведінки у разі коли буде помилка зчитування файлу і аргумент буде нульовим, програмна реалізація запропонованої sequence-to-sequence моделі генерує вихідну послідовність «file *arg_1 = fopen (arg_2, arg_3); if (arg_1 == null) return false;» у якій після відкриття файлу проводиться перевірка та оброблюється випадок якщо файл не відкрився.

```
transform(u'FILE *ARG_1 = fopen(ARG_2, ARG_3);')

Input: <start> file * arg_1 = fopen ( arg_2 , arg_3 ) ; <end>
Predicted transform:   file * arg_1 = fopen ( arg_2 , arg_3 ) ; if ( arg_1 = = null ) return false ; <end>
```

Рисунок 3.17 – Результат виправлення помилок безпеки пов’язаної з відсутністю перевірки відкриття файлу, що запропонувала модель на основі глибинного навчання

На Рисунку 3.18 ситуація аналогічна з Рисунком 3.17, змінюється лише вхідна послідовність, при цьому створюється відповідна вихідна послідовність у якій також змінилась логіка обробки випадку, якщо файл не було відкрито.

```
transform(u'FILE *ARG_1; ARG_1 = fopen(ARG_2, ARG_3);')

Input: <start> file * arg_1 ; arg_1 = fopen ( arg_2 , arg_3 ) ; <end>
Predicted transform:   file * arg_1 ; arg_1 = fopen ( arg_2 , arg_3 ) ; if ( arg_1 = = null ) { printf ( \ unable
to open file \ \ r \ \ n \ ) ; arg_3 = 1 ; return arg_3 ; } <end>
```

Рисунок 3.18 – Результат виправлення помилки безпеки пов’язаної з відсутністю перевірки відкриття файлу, що запропонувала модель на основі глибинного навчання

Висновки до розділу 3

У даному розділі детально описуються запропоновані технології виявлення помилок безпеки у програмному забезпеченні. Наведено діаграму класів та діаграму послідовностей для запропонованого підходу на основі правил. Також детально описано запропоновані шаблони для виправлення помилок безпеки описаних у Розділі 2. Наведено топологію моделі машинного навчання sequence-to-sequence для підходу на основі глибинного навчання. Також представлено програмну реалізацію моделі енкодера, декодера і вагів уваги.

Після розробки програмних реалізацій, їх було випробувано на проектах з відкритим кодом таких корпорацій, як Google та Microsoft. У одному з проектів, а саме Microsoft/Terminal, було виявлено помилку пов’язану з небезпечним закриттям файлу, яку було успішно виправлено запропонованою технологією на основі правил. Також наведено результати створених виправлень за допомогою sequence-to-sequence моделі машинного навчання.

4 РОЗРОБКА СТАРТАП-ПРОЕКТУ

Розділ має на меті проведення маркетингового аналізу стартап проекту задля визначення принципової можливості його ринкового впровадження та можливих напрямів реалізації цього впровадження.

4.1 Опис ідеї проекту

У цьому підрозділі аналізується зміст ідеї, що пропонується, можливі напрямки застосування, основні вигоди, що може отримати користувач товару (за кожним напрямком застосування) (таблиця 4.1) та відмінність від існуючих аналогів та замінників (таблиця 4.2)

Таблиця 4.1 – Опис ідеї стартап-проекту

Зміст ідеї	*Напрямки застосування*	*Вигоди для користувача*
Система виправлення помилок безпеки в програмному забезпеченні	1. IT компанії, що займаються розробкою ПЗ	1. Дешевизна, у порівнянні з сумою грошей, що потрібна для наймання відповідного фахівця
	2. IT компанії, що займаються тестуванням ПЗ	2. Швидкість створення виправлення, в порівнянні з часом необхідним для виправлення відповідним фахівцем

Продовження таблиці 4.1

Зміст ідеї	*Напрямки застосування*	*Вигоди для користувача*
Система виправлення помилок безпеки в програмному забезпеченні	3. Програмісти	3. Легкість у використанні
	4. Люди, що вивчають програмування	4. Легка інтеграція з git репозиторіями

Таблиця 4.2 – Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	*Техніко-економічні характеристики ідеї*	*(потенційні) товари/концепції конкурентів*			*W (слабка сторона)*	*N (нейтральна сторона)*	*S (сильна сторона)*
		Мій проект	*Prophet*	*GenProg*			
1.	Швидкість роботи	Працює з проектами будь-якого розміру	Працює з малими проектами	Працює з середніми проектами	дуже залежить від розміру проекту	лінійна залежність	виконується за константний час
2.	Інтеграція з системами контролю версій	GitHub	тільки локальні проекти	тільки локальні проекти	-	тільки локальні проекти	локальні проекти та GitHub

Продовження таблиці 4.2

№ п/п	*Техніко-економічні характерис-тики ідеї*	*(потенційні) товари/концепції конкурентів*			*W (слабка сторона)*	*N (нейтральна сторона)*	*S (сильна сторона)*
		Мій проект	*Prophet*	*GenProg*			
3.	Необхідність в компіляції проекту	потрібен тільки початковий код	потрібна компіляція коду	потрібна компіляція коду	потрібна компіляція продукту	-	потрібен тільки початковий код
4.	Необхідність у створенні цілого ряду виправлень	Створює одне точне виправлення для даної помилки	Створює ряд виправлень, а потім обирає таке, що виправляє помилку безпеки	Створює ряд виправлень, а потім обирає таке, що виправляє помилку безпеки	Потрібно створювати ряд потенційних виправлень, а потім обирати правильне	-	Одразу створює одне правильне виправлення

Кінець таблиці 4.2

№ п/п	*Техніко-економічні характерис-тики ідеї*	*(потенційні) товари/концепції конкурентів*			*W (слабка сторона)*	*N (нейтральна сторона)*	*S (сильна сторона)*
		Мій проект	*Prophet*	*GenProg*			
6.	Кількість вхідних параметрів	Потрібен конфігураційний файл у якому вказані помилки безпеки та файли у яких вони знаходяться	Потрібен конфігураційний файл та набір тестів серед яких принаймні один виявляє помилку безпеки	Потрібен конфігураційний файл та набір тестів серед яких принаймні один виявляє помилку безпеки	Конфігураційний файл та додаткові параметри	-	Лише конфігураційний файл

4.2 Технологічний аудит ідеї проекту

В межах даного підрозділу був проведений аудит технології, за допомогою якої можна реалізувати ідею проекту та заповнена таблиця 4.3

Таблиця 4.3 – Технологічна здійсненність ідеї проекту

№ п/п	*Ідея проекту*	*Технології її реалізації*	*Наявність технологій*	*Доступність технологій*
1	Використання перетворення початкового коду	Використання перетворення початкового коду по типу абстрактного синтаксичного дерева.	Наявні деякі алгоритми для побудови абстрактного синтаксичного дерева.	Clang compiler tool знаходиться в відкритому доступі.
2	Впровадження програмної реалізації моделі машинного навчання для виправлення помилок безпеки	Використання сучасних фреймворків для побудови моделі нейронної мережі.	Нейронна мережа повинна бути побудована та натренована.	Фреймворки tenserflow, keras знаходяться в відкритому доступі.
3	Використання коду з систем контролю версій	Використання початкових кодів з віддалених репозиторіїв	Використання задокументованого API	Системи контролю версій, такі як GitHub, GitLab, BitBucket мають відкриті API

Продовження таблиці 4.3

№ п/п	*Ідея проекту*	*Технології її реалізації*	*Наявність технологій*	*Доступність технологій*
4	Взаємодія з іншими продуктами	Впровадження задокументованого API, що дозволяє автоматично виправляти помилки безпеки	Співробітництво з виробниками патчів, аналізу проектів.	Співробітництво можливе за умови двосторонньої вигідності партнерства
5	Використання напрацьованих змін початкового коду	Використання бази кодів, що були з вразливістю та без вразливості	Збирання вразливих та не вразливих початкових кодів	Існують загальновідомі бази даних з такими початковими кодами
Обрана технологія реалізації ідеї проекту: так як для реалізації ідеї проекту, всі технології є наявними та доступними, тому обираються всі вище описані технології.				

4.3 Аналіз ринкових можливостей запуску стартап-проекту

В межах даного підрозділу було визначено ринкові можливості які можна використати під час ринкового впровадження проекту, та ринкових загроз, які можуть перешкодити реалізації проекту, дозволяє спланувати напрями розвитку

проекту із урахуванням стану ринкового середовища, потреб потенційних клієнтів та пропозицій проектів-конкурентів (таблиця 4.4, таблиця 4.5).

Таблиця 4.4 – Попередня характеристика потенційного ринку стартап-проекту

№ п/п	*Показники стану ринку (найменування)*	*Характеристика*
1	Кількість головних гравців, од	20
2	Загальний обсяг продаж, грн/ум.од	20 млн ум. од.
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Потреба у базі даних початкових кодів, що мають помилки безпеки, та таких, що не мають помилки безпеки
5	Специфічні вимоги до стандартизації та сертифікації	Не потребує стандартизації та сертифікації
6	Середня норма рентабельності в галузі (або по ринку), %	Не менше 150

Таблиця 4.5 – Характеристика потенційних клієнтів стартап-проекту

Потреба, що формує ринок	*Цільова аудиторія (цільові сегменти ринку)*	*Відмінності у поведінці різних потенційних цільових груп клієнтів*	*Вимоги споживачів до товару*
Виправлення помилок безпеки, що можуть призвести до втрат даних або відмові в обслуговуванні цільового програмного забезпечення	1. IT компанії, що займаються розробкою ПЗ 2. IT компанії, що займаються тестуванням ПЗ 3. Програмісти 4. Люди, що вивчають програмування	Для кожної з категорій існують окремі цінності пропозицій. Пропозиція відрізняється для кожної цільової групи.	• точність виправлення помилок безпеки • швидкість роботи • покриття усіх помилок безпеки • зручний користувацький інтерфейс

Після визначення потенційних груп клієнтів проводиться аналіз ринкового середовища: складаються таблиці факторів, що сприяють ринковому впровадженню проекту, та факторів, що йому перешкоджають (таблиці 4.6-4.7)

Таблиця 4.6 – Фактори загроз

№ п/п	*Фактор*	*Зміст загрози*	*Можлива реакція компанії*
1	Цінова конкуренція	Деякі продукти наявні в вільному доступі	Покращення якості виправлення та розширення бази помилок безпеки відносно відкритих продуктів
2	Низька точність виправлення помилок безпеки	Деякі помилки виправляються хибно	Покращення методу, зниження помилок першого та другого роду
3	Низька швидкість роботи	Програма працює дуже довго та займає багато ресурсів	Оптимізація програмного коду, використання більших ресурсів комп’ютера
4	Мала кількість виправлених помилок	Деякі помилки безпеки не виправляються	Збирання або знаходження бази початкових кодів з помилками безпеки та без них

Таблиця 4.7 – Фактори можливостей

№ п/п	*Фактор*	*Зміст можливості*	*Можлива реакція компанії*
1	Поява нових топологій нейронних мереж	Підвищення результатів точності виявлення та часу роботи	Впровадження нової топології нейронної мережі для генерації послідовності коду, розширення функціоналу, проведення маркетингової компанії
2	Поява нового методу представлення початкового коду	Збільшення можливих точок входу, взяття до уваги додаткових факторів	Підвищення результатів виправлення помилок безпеки
3	Поява відкритої бази даних з початковими кодами, що містять помилки безпеки та відповідно не містять	Збільшення точності виправлення помилок безпеки, виправлення нових типів помилок безпеки	Розширення типів помилок безпеки, що можуть бути виправлені
4	Залучення нових відомих компаній у якості клієнтів чи постачальників	Підвищення фінансування	Підвищення якості програмного коду, оптимізації, шляхом найму додаткових спеціалістів

Продовження таблиці 4.7

№ п/п	*Фактор*	*Зміст можливості*	*Можлива реакція компанії*
5	Поява нової мови програмування	Вийти на ринок виправлення помилок безпеки першими	Вивчення особливостей мови та запровадження даного методу для іншої мови програмування

Надалі проводиться аналіз пропозиції: визначаються загальні риси конкуренції на ринку (таблиця 4.8).

Таблиця 4.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	*В чому проявляється дана характеристика*	*Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)*
Олігополістична конкуренція	Інструменти засновані на шаблонах дуже дорогі для розробки, оскільки потребується багато ресурсів для опису кожного правила виправлення, тому існуючі рішення є тільки у великих IT компаніях	Вдосконалення рішення для підвищення якості надаваних послуг для клієнтів при невеликих затратах та невеликій ціні продукту

Продовження таблиці 4.8

Особливості конкурентного середовища	*В чому проявляється дана характеристика*	*Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)*
За рівнем конкурентної боротьби: національний	Надання послуг для клієнтів в Україні та за її межами	Застосування новітніх технологій машинного навчання для підвищення конкурентоспроможності
За галузевою ознакою: внутрішньогалузева	Рішення можуть використовуватися тільки компаніям, що займаються розробкою або тестування програмного забезпечення	Застосування маркетингових методів, які позволять користуватися продуктом невеликим компаніям, або клієнтам, що навчаються програмуванню
Конкуренція за видами товарів: товарно-видова	Рішення, що використовується для задоволення потреб клієнтів, але істотно відрізняються від рішень конкурентів	Надання послуг з виправлення помилок безпеки на основі машинного навчання
За характером конкурентних переваг: цінова	Ціна запропонованого рішення є меншою за рахунок використання вже готових доступних результатів наукових досліджень та розробок	Надання порівняного рівня послуг, проте з наголосом на інформаційну безпеку та за меншу ціну

Кінець таблиці 4.8

Особливості конкурентного середовища	*В чому проявляється дана характеристика*	*Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)*
За інтенсивністю: марочна	Сукупність характеристик та властивостей рішення	Підвищення якості роботи програмного рішення для клієнтів

Після аналізу конкуренції проводиться більш детальний аналіз умов конкуренції в галузі (за моделлю 5 сил М. Портера) (таблиця 4.9).

Таблиця 4.9 – Аналіз конкуренції в галузі за М. Портером

Складові аналізу	*Прямі конкуренти в галузі*	*Потенційні конкуренти*	*Постачаль-ники*	*Клієнти*	*Товари-замінники*
	GenProg	Prophet	Досконалі продукти, підтримка зі сторони розробки, рішення перевірене часом	Великі IT компанії мають великі фінанси та підтримують високу якість розроблюваних продуктів	Товари замінники відсутні
Висновки:	Є можливості розширення на ринку, але над іншими помилками уже можуть працювати більші компанії	Продукт є дослідницьким, тому не є конкурентом	Постачальники диктують умови роботи на ринку; компанії, які мають найбільше ресурсів, мають найбільш якісний продукт.	Клієнти диктують умови на ринку; при організації вибору послуги, відчутний рівень відношення до вартості рішення та його якості.	Обмежень на ринку через товари замінники немає.

На основі проведеної роботи проводиться обґрунтування факторів конкурентоспроможності (таблиця 4.10)

Таблиця 4.10 – Обґрунтування факторів конкурентоспроможності

№ п/п	*Фактор конкурентоспроможності*	*Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)*
1	Якість продукту	Продукт працює швидко та показує гарні показники виправлення, без великої кількості помилок першого та другого роду та з мінімально необхідною кількістю ресурсів.
2	Рівень ціни	Ціна розробки програмного продукту не велика, оскільки використовуються новітні технології машинного навчання, які здатні замінити деяку працю людини.
3	Наявність конкурентів	Конкуренти – великі IT компанії, або відкриті продукти, тому продукт або комерційний або недостатньої якості
4	Масштабованість продукту	Продукт не потребує масштабованості, оскільки одна й та ж копія може бути проданою без змін

За визначеними факторами конкурентоспроможності (таблиця 4.10) проводиться аналіз сильних та слабких сторін стартап-проекту (таблиця 4.11).

Таблиця 4.11 – Порівняльний аналіз сильних та слабких сторін

Фактор конкурентоспроможності	*Бали 1-20*	*Рейтинг товарів-конкурентів у порівнянні з GenProg, Prophet*						
		3	*2*	*1*		*1*	*2*	*3*
Якість продукту	20			+				
Рівень ціни	10	+						
Масштабованість продукту	10							+

Фінальним етапом ринкового аналізу можливостей впровадження проекту є складання SWOT-аналізу (матриці аналізу сильних (Strength) та слабких (Weak) сторін, загроз (Troubles) та можливостей (Opportunities) (таблиця 4.12)

Таблиця 4.12 – SWOT- аналіз стартап-проекту

Сильні сторони: використання машинного навчання для питання створення виправлення помилки безпеки, мала кількість людської роботи для реалізації підходу, швидкість роботи, низька ціна розробки	Слабкі сторони: низький рівень маркетингу, наявність конкурентів у даній сфері
Можливості: залучення великих IT компаній в якості постачальників чи клієнтів	Загрози: цінова конкуренція; зниження доходів потенційних споживачів; конкуренція в якості продукту та підтримці.

На основі SWOT-аналізу розробляються альтернативи ринкової поведінки (перелік заходів) для виведення стартап-проекту на ринок та орієнтовний оптимальний час їх ринкової реалізації (таблиця 4.13).

Таблиця 4.13 – Альтернативи ринкового впровадження стартап-проекту

№ п/п	*Альтернатива (орієнтовний комплекс заходів) ринкової поведінки*	*Ймовірність отримання ресурсів*	*Строки реалізації*
1	Створення угоди з певною IT компанією	Висока ймовірність отримання ресурсів	до 1 року
2	Впровадження рекламних засобів для таргетингової категорії	Середня ймовірність отримання ресурсів	до 6 місяців
3	Участь в IT виставках, публікації в журналах	Середня ймовірність отримання ресурсів	до 1 року
4	Організування патенту	Мала ймовірність отримання ресурсів	до 1 року
5	Додання до функціоналу специфічних функцій	Мала ймовірність отримання ресурсів	до 3 місяців

4.4 Розроблення ринкової стратегії проекту

Розроблення ринкової стратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів (таблиця 4.14)

Таблиця 4.14 – Вибір цільових груп потенційних споживачів

№ п/п	*Опис профілю цільової групи потенційних клієнтів*	*Готовність споживачів сприйняти продукт*	*Орієнтовний попит в межах цільової групи (сегменту)*	*Інтенсивність конкуренції в сегменті*	*Простота входу у сегмент*
1	IT компанії, що займаються розробкою ПЗ	Клієнти потребують продукт такого типу та готові ним користуватись	Високій рівень попиту	Існує конкуренція з подібними функціями	Існує складність входу, оскільки великі компанії мають власнорозроблені продукти
2	IT компанії, що займаються тестуванням ПЗ	Клієнти потребують продукт такого типу та готові ним користуватись	Високій рівень попиту		Існують конкуренти з подібним функціоналом

Продовження таблиці 4.14

<table>
<tr><th>№ п/п</th><th>Опис профілю цільової групи потенційних клієнтів</th><th>Готовність споживачів сприйняти продукт</th><th>Орієнтовний попит в межах цільової групи (сегменту)</th><th>Інтенсивність конкуренції в сегменті</th><th>Простота входу у сегмент</th></tr>
<tr><td>3</td><td>Програмісти</td><td>Низька зацікавленість</td><td>Середній рівень попиту</td><td rowspan="2">Існує конкуренція з подібними функціями</td><td>Не велика зацікавленість, оскільки перевіркою займається сама компанія</td></tr>
<tr><td>4</td><td>Люди, що вивчають програмування</td><td>Низька зацікавленість</td><td>Низький попит, пов’язаний з невеликою необхідністю вивчати вразливості тієї або іншої функції</td><td>Існують безкоштовні та відкриті продукти, але з меншим функціоналом</td></tr>
<tr><td colspan="6">Які цільові групи обрано: вибрані групи з високим рівнем попиту, а саме IT компанії, що займаються розробкою ПЗ та IT компанії, що займаються тестуванням ПЗ</td></tr>
</table>

Для роботи в обраних сегментах ринку необхідно сформувати базову стратегію розвитку (таблиця 4.15).

Таблиця 4.15 – Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	*Стратегія охоплення ринку*	*Ключові конкурентоспроможні позиції відповідно до обраної альтернативи*	*Базова стратегія розвитку**
Флангова атака	Стратегія концентрованого маркетингу	Сильні сторони та можливості рішення	Стратегія спеціалізації

Наступним кроком є вибір стратегії конкурентної поведінки (таблиця 4.16)

Таблиця 4.16 – Стратегії конкурентної поведінки

№ п/п	*Чи є проект «першопроходьцем» на ринку?*	*Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?*	*Чи буде компанія копіювати основні характеристики товару конкурента, і які?*	*Стратегія конкурентної поведінки**
1	Частково	Так	Так, загалом це будуть взаємодія товару з інтерфейсом, можливості конфігурації, тощо.	Стратегія лідера

На основі вимог споживачів з обраних сегментів до постачальника (стартап-компанії) та до продукту (див. таблицю 4.5), а також в залежності від обраної базової стратегії розвитку (таблиця 4.15) та стратегії конкурентної поведінки (таблиця 4.16) розробляється стратегія позиціонування (таблиця 4.17), що полягає у формуванні ринкової позиції (комплексу асоціацій), за яким споживачі мають ідентифікувати торгівельну марку/проект.

Таблиця 4.17 – Визначення стратегії позиціонування

Вимоги до товару цільової аудиторії	*Базова стратегія розвитку*	*Ключові конкурентоспроможні позиції власного стартап-проекту*	*Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)*
Отримання готового продукту зі зручним користувацьким інтерфейсом, що легко налаштовується на будь-який продукт, можливість працювати з віддаленими репозиторіями, якість створеного виправлення	Стратегія спеціалізації	Сильні сторони та можливості рішення	Швидкість, якість виправлення, доступна ціна

4.5 Розроблення маркетингової програми стартап-проекту

Першим кроком є формування маркетингової концепції товару, який отримає споживач. Для цього у таблиці 4.18 потрібно підсумувати результати попереднього аналізу конкурентоспроможності товару.

Таблиця 4.18 – Визначення ключових переваг концепції потенційного товару

№ п/п	*Потреба*	*Вигода, яку пропонує товар*	*Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити*
1	Виправлення можливих вразливостей програмного забезпечення	Висока якість та швидкість виправлення	Отримання комплексного продукту, який буде точно та швидко виправляти можливі вразливості використання функцій
2	Легкість у використанні	Максимальна автоматичність продукту, мінімальна кількість вхідних параметрів	На вхід приймається лише файл конфігурації, після запуску програми клієнт отримує створене виправлення, та не робить жодних дій на проміжних етапах роботи продукту

Продовження таблиці 4.18

№ п/п	*Потреба*	*Вигода, яку пропонує товар*	*Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити*
3	Якість створеного виправлення	Клієнту пропонується лише правильне виправлення	Клієнту пропонується лише правильне виправлення
4	Можливість конфігурації відносно проекту	Під різні проекти можна налаштувати виправлення помилок безпеки	Можливість налаштувати систему виправлення відповідно до кожного проекту, з врахуванням його специфікацій

Надалі розробляється трирівнева маркетингова модель товару: уточнюється ідея продукту та/або послуги, його фізичні складові, особливості процесу його надання (таблиця 4.19).

Таблиця 4.19 – Опис трьох рівнів моделі товару

Рівні товару	*Сутність та складові*		
I. Товар за задумом	Система виправлення помилок безпеки в початковому коді.		
II. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх /Тл/Е/Ор
	1. Використовує глибинне навчання для створення виправлення, що призводить до високої швидкості роботи	М	Тх
	2. Створює якісні виправлення	М	Тх
	3. Легкий у використанні	М	Тх
	4. Працює с системами контролю версій	М	Тх
	Якість: виправляються помилки безпеки відносно CWE-119, CWE-369, CWE-399		
	Пакування: Надання користувачу електронної ліцензії (ключа доступу) до хмарного сервісу (якщо побудова системи відбувається в хмарному середовищі		
	Марка: Patch Generator		
III. Товар із підкріпленням	До продажу: початок рекламної компанії, аналіз open source кодів		
	Після продажу: продовження рекламної компанії		
За рахунок чого потенційний товар буде захищено від копіювання: користувач не має початкового коду, а має лише доступ до веб інтерфейсу			

Наступним кроком є визначення цінових меж, якими необхідно керуватись при встановленні ціни на потенційний товар (остаточне визначення ціни відбувається під час фінансово-економічного аналізу проекту), яке передбачає аналіз ціни на товари-аналоги або товари субститути, а також аналіз рівня доходів цільової групи споживачів (таблиця 4.20). Аналіз проводиться експертним методом.

Таблиця 4.20 – Визначення меж встановлення ціни

Рівень цін на товари-замінники	*Рівень цін на товари-аналоги*	*Рівень доходів цільової групи споживачів*	*Верхня та нижня межі встановлення ціни на товар/послугу*
11000	10000	13000	2500-4000

Наступним кроком є визначення оптимальної системи збуту, в межах якого приймається рішення (таблиця 4.21)

Таблиця 4.21 – Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	*Функції збуту, які має виконувати постачальник товару*	*Глибина каналу збуту*	*Оптимальна система збуту*
мінімальна кількість посередників	організувати широку мережу збуту товару	3	не пряма

Останньою складової маркетингової програми є розроблення концепції маркетингових комунікацій, що спирається на попередньо обрану основу для позиціонування, визначену специфіку поведінки клієнтів (табл. 4.22).

Таблиця 4.22 – Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	*Канали комунікацій, якими користуються цільові клієнти*	*Ключові позиції, обрані для позиціонування*	*Завдання рекламного повідомлення*
Дослідження переваг та властивостей продукту для якісного та швидкого виправлення помилок безпеки своїх продуктів	Інформаційні канали	Виправлення помилок безпеки в початковому коді	Донести переваги та доступність до можливих клієнтів

Висновки до розділу 4

У розділі було проведено маркетинговий аналіз стартап проекту задля визначення принципової можливості його ринкового впровадження та можливих напрямів реалізації цього впровадження. А саме, було описано ідеї стартап-проекту, визначено сильні та слабкі характеристики ідей проекту, наведено попередню характеристику потенційного ринку для стартап-проекту, надано характеристику потенційних клієнтів. Також вказано фактори потенційних загроз та можливосте. Було проведено аналіз конкуренції на ринку та в галузі, обґрунтовано фактори конкурентоспроможності. Наведено порівняльний аналіз сильних та слабких сторін продукту у порівнянні з конкурентами. Визначено базову стратегію розвитку, стратегію конкурентної поведінки та стратегію позиціонування. Описано ключові переваги концепції потенційного товару.

ВИСНОВКИ

Помилки безпеки допущені на етапі розробки програмного забезпечення пов'язані з небезпечним використанням функцій або операторів стають головною загрозою для безпеки багатьох систем. Написання виправлень цих помилок вручну є рутинною та дуже клопіткою роботою. Для полегшення цієї проблеми постає питання в автоматизації процесу створення виправлень помилок безпеки в програмному забезпеченні. А враховуючи велику кількість продуктів з відкритим початковим кодом у яких було зроблено відповідні виправлення, можна накопичити базу даних для алгоритмів глибинного навчання, які б могли розв'язати проблему автоматичного створення виправлень.

На сьогоднішній день серед існуючих підходів та рішень проблеми автоматичного виправлення помилок безпеки у програмному забезпеченні можна виділити два основних класи. Один з класів базується на узагальнені правил, що стосуються конкретної помилки, та написанню відповідного шаблона виправлення. Інший клас – це спроби використання алгоритмів машинного навчання для виправлення найбільш поширених помилок для яких уже було накопичено бази даних з типовими виправленнями. Більшість таких підходів виправляють лише стилістичні помилки або помилки компіляції для різних мов програмування.

Після дослідження існуючих підходів до автоматичного виправлення помилок безпеки у програмному забезпеченні було вирішено розвивати метод виправлення помилок безпеки в C/C++ коді. У даній роботі було запропоновано дві технології, що виправляють помилки безпеки у програмному забезпеченні. Одна з технологій заснована на детермінованих шаблонах, що дозволяють автоматично трансформувати абстрактне синтаксичне дерево файлу програми, що містить помилку безпеки, таким чином виправляючи помилку. Даний підхід має один недолік, а саме, необхідність у рутинному написані шаблонів виправлення. Тому наступним кроком було розроблення технології на основі

глибинного навчання, яка може створювати виправлення автоматично навчившись на вже існуючих виправленнях впровадженими розробниками у своїх програмах. Для цієї цілі було обрано sequence-to-sequence модель машинного навчання, яка може приймати на вхід послідовність токенів (код-гаджет), та генерувати нову вихідну послідовність з виправленою помилкою безпеки.

Після розробки програмних реалізацій, їх було випробувано на проектах з відкритим кодом таких корпорацій, як Google та Microsoft. У деяких проектах було знайдено та успішно виправлено помилки безпеки, що відносяться до класів помилок CWE – 119, CWE – 369 та CWE – 399.

У ході дослідження було проведено маркетинговий аналіз можливості створення стартап проекту і його впровадження на ринку та можливих напрямів реалізації кінцевого програмного рішення.

Результати роботи можуть бути використані компаніями, що розробляють програмне забезпечення та програмістами для швидкого і якісного виправлення специфічних помилок безпеки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Stroud F. Security Vulnerability [Електронний ресурс] F. Stroud – Режим доступу: https://www.webopedia.com/TERM/S/security_vulnerability.html
2. European Union Agency for Cybersecurity Vulnerability [Електронний ресурс] – Режим доступу: https://www.enisa.europa.eu/topics/threat-risk-management/risk-management/current-risk/risk-management-inventory/glossary#G52
3. Avizienis A. Basic Concepts and Taxonomy of Dependable and Secure Computing [Текст] / A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr // Ieee transactions on dependable and secure computing 1.1 – 2004. – с. 11–33.
4. Monperrus M. Automatic Software Repair: a Bibliography [Текст] / M. Monperrus // ACM Computing Surveys, Association for Computing Machinery – 2017. – с. 1-24.
5. Staats M. Programs, Tests, and Oracles: the Foundations of Testing Revisited [Текст] / M. Staats, M. W. Whalen, M. P. E. Heimdahl // Proceedings of the International Conference on Software Engineering – 2011. - с. 391–400.
6. Monperrus M. A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair [Текст] / M. Monperrus // Proceedings of the international conference on software engineering – 2014. – с. 234–242.
7. Tsipenyuk K. Seven Pernicious Kingdoms: a Taxonomy of Software Security Errors [Текст] / K. Tsipenyuk, B. Chess, G. McGraw // Security & privacy 3.6 – 2005. – с. 81–84.
8. Duraes J. Emulation of Software Faults: a Field Data Study and a Practical Approach [Текст] / J. Duraes, H. Madeira // Ieee transactions on software engineering 32.11 - 2006. - с. 849–867.

9. Martinez M. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing [Текст] / M. Martinez, M. Monperrus // Empirical Software Engineering 20.1 – 2013. – с. 176–205.
10. Weimer W. Automatically Finding Patches Using Genetic Programming [Текст] / W. Weimer, T. Nguyen, C. Le Goues, S. Forrest // Proceedings of the International Conference on Software Engineering - 2009. – с. 364-374.
11. Weimer W. Automatic Program Repair with Evolutionary Computation [Текст] / W. Weimer, S. Forrest, C. Le Goues, T. Nguyen // Communications of the acm 53.5 - 2010. - с. 109.
12. Forrest S. A Genetic Programming Approach to Automated Software Repair [Текст] / S. Forrest, T. Nguyen, W. Weimer, C. Le Goues // Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation – 2009. – с. 947–954.
13. GenProg [Електронний ресурс] – Режим доступу: https://squareslab.github.io/genprog-code/
14. Le Goues C. A Systematic Study of Automated Program Repair: Fixing 55 Out of 105 Bugs for $8 Each [Текст] / C. Le Goues, M. Dewey-Vogt, S. Forrest, W. Weimer // Proceedings of the International Conference on Software Engineering – 2012. – с. 3–13.
15. Schulte E. Automated Program Repair Through the Evolution of Assembly Code [Текст] / E. Schulte, S. Forrest, W. Weimer // Proceedings of the IEEE/ACM International Conference on Automated Software Engineering – 2010. – с. 313–316.
16. Le Goues C. GenProg: a Generic Method for Automatic Software Repair [Текст] / C. Le Goues, T. Nguyen, S. Forrest, W. Weimer // Ieee transactions on software engineering 38 – 2012 – с. 54–72.
17. Qi Y. Efficient Automated Program Repair Through FaultRecorded Testing Prioritization [Текст] / Y. Qi, X. Mao, Y. Lei // Proceedings of ICSM – 2013. – с. 180-189.

18. Qi Y. The Strength of Random Search on Automated Program Repair [Текст] / Y. Qi, X. Mao, Y. Lei, Z. Dai, C. Wang // Proceedings of the 36th International Conference on Software Engineering – 2014. – c. 254–265.
19. V. Oliveira Improved Crossover Operators for Genetic Programming for Program Repair [Текст] / V. Oliveira, E. Souza, C. Le Goues, C. G. Camilo // Proceedings of the 8th international symposium on search based software engineering – 2016. – c. 112-127.
20. M. Stumptner A Model-based Approach to Software Debugging [Текст] / M. Stumptner, F. Wotawa // Proceedings on the Seventh International Workshop on Principles of Diagnosis – 1996. – c.35 -51.
21. A. Arcuri A Novel Co-evolutionary Approach to Automatic Software Bug Fixing [Текст] / A. Arcuri, X. Yao // Proceedings of the IEEE Congress on Evolutionary Computation – 2008 – c. 162–168.
22. A. Arcuri Automatic Software Generation and Improvement Through Search Based Techniques [Текст]: PhD thesis. The University of Birmingham - 2009.
23. A. Arcuri Evolutionary Repair of Faulty Software [Текст] / A. Arcuri // Applied soft computing 11.4 – 2011. -c. 3494–3514.
24. V. Debroy Using Mutation to Automatically Suggest Fixes for Faulty Programs [Текст] / V. Debroy, W. Wong // Proceedings of the International Conference on Software Testing, Verification and Validation – 2010. – c. 65–74.
25. M. Nica On the Use of Mutations and Testing for Debugging [Текст] / M. Nica, S. Nica, F. Wotawa // Software: practice and experience 43.9 – 2013. – c. 1121–1142.
26. C. Kern Automatic Error Correction of Java Programs [Текст] / C. Kern, J. Esparza // Formal Methods for Industrial Critical Systems – 2010. – c. 67–81.
27. H. D. T. Nguyen SemFix: Program Repair via Semantic Analysis [Текст] / H. D. T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra // Proceedings of the International Conference on Software Engineering – 2013. – c. 772-781.

28. S. Chandra Angelic Debugging [Текст] / S. Chandra, E. Torlak, S. Barman, R. Bodik // Proceeding of the International Conference on Software Engineering – 2011. – с. 121–130.
29. S. Jha Oracle-guided Component-based Program Synthesis [Текст] / S. Jha, S. Gulwani, S. A. Seshia, A. Tiwari // Proceedings of the International Conference on Software Engineering. Vol. 1. – 2010. - с. 215–224.
30. S. Mechtaev Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis [Текст] / S. Mechtaev, J. Yi, A. Roychoudhury // Proceedings of the 38th international conference on software engineering – 2016. – с. 691–701.
31. D. Kim Automatic Patch Generation Learned From Human-Written Patches [Текст] / D. Kim, J. Nam, J. Song, S. Kim // Proceedings of ICSE – 2013. – с. 802-811.
32. F. DeMarco Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT [Текст] / F. DeMarco, J. Xuan, D. Le Berre, M. Monperrus // Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis – 2014.
33. X. Zhang Locating Faults Through Automated Predicate Switching [Текст] / X. Zhang, N. Gupta, R. Gupta // Proceedings of the 28th International Conference on Software Engineering – 2006. – с. 272–281.
34. S. L. Marcote Automatic Repair of Infinite Loops [Електронний ресурс] S. L. Marcote, M. Monperrus – Режим доступу: https://arxiv.org/pdf/1504.05078.pdf
35. S. H. Tan Relifix: Automated Repair of Software Regressions [Текст] / S. H. Tan, A. Roychoudhury // Proceedings of ICSE – 2015. – с. 471-482.
36. S. Mechtaev DirectFix: Looking for Simple Program Repairs [Текст] / S. Mechtaev, J. Yi, A. Roychoudhury // Proceedings of the 37th International Conference on Software Engineering – 2015. – с. 448-458.
37. A. Griesmayer Repair of Boolean Programs with An Application to C [Текст] / A. Griesmayer, R. Bloem, B. Cook // Computer Aided Verification – 2006. – с. 358–371.

38. F. Long Staged Program Repair with Condition Synthesis [Текст]/ F. Long, M. C. Rinard // Proceedings of ESEC/FSE – 2015. – c. 166-178.
39. S. Sidiroglou-Douskos Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications [Текст]/ S. Sidiroglou-Douskos, E. Lahtinen, F. Long, M. Rinard // Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation – 2015. – c. 43–54.
40. Y. Ke Repairing Programs with Semantic Code Search [Текст]/ Y. Ke, K. T. Stolee, C. Le Goues, Y. Brun // Proceedings of the International Conference on Automated Software Engineering – 2015. – c. 298-309.
41. F. Long Prophet: Automatic Patch Generation via Learning From Successful Patches [Текст] / F. Long, M. C. Rinard // Proceedings of the Symposium on Principles of Programming Languages -2016.
42. E. Kneuss Deductive program repair [Текст] / E. Kneuss, M. Koukoutos, V. Kuncak // Computer-Aided Verification (CAV) – 2015. – c. 217-233.
43. H. D. T. Nguyen SemFix: Program Repair via Semantic Analysis [Текст] / H. D. T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra // Proceedings of the International Conference on Software Engineering – 2013. – c. 772-781.
44. R. Samanta Cost-aware automatic program repair [Текст] / R. Samanta, O. Olivo, E. A. Emerson // In Static Analysis - 21st International Symposium – 2014. – c. 268–284.
45. H. Samimi Automated repair of HTML generation errors in PHP applications using string constraint solving [Текст] / H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, L. J. Hendren // ICSE – 2012. – c. 277–287.
46. F. Long Staged program repair with condition synthesis [Текст] / F. Long and M. Rinard // Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering – 2015. – c. 166–178.
47. J. H. Perkins Automatically patching errors in deployed software [Текст] / J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan,W.-F. Wong, Y. Zibin, M. D. Ernst, M.

Rinard // Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles – 2009. – с. 87–102.

48. X. B. D. Le History Driven Program Repair [Текст] / X. B. D. Le, D. Lo, C. L. Goues // Proceedings of the 23rd international conference on software analysis, evolution, and reengineering (saner) – 2016. – с. 213–224.
49. H. He Automated Debugging Using Path-Based Weakest Preconditions [Текст] / H. He, N. Gupta // FASE – 2004. – с. 267–280.
50. Y. Wei Automated Fixing of Programs with Contracts / Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, A. Zeller // Proceedings of the International Symposium on Software Testing and Analysis – 2010. – с. 26-50.
51. A. Zeller Automated Fixing of Programs with Contracts [Текст] / A. Zeller, Y. Wei, B. Meyer, M. Nordio, C. A. Furia, Y. Pei // Ieee transactions on software engineering 40.5 – 2014. – с. 427–449.
52. D. Gopinath Specification-based Program Repair Using SAT [Текст] / D. Gopinath, M. Z. Malik, S. Khurshid // Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems – 2011. – с. 173-188.
53. D. Jackson Finding Bugs with a Constraint Solver [Текст] / D. Jackson, M. Vaziri // Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis – 2000. – с. 14–25.
54. R. Könighofer Automated Error Localization and Correction for Imperative Programs [Текст] / R. Könighofer R. Bloem // Formal Methods in Computer-Aided Design (FMCAD) – 2011. – с. 91–100.
55. W. Weimer Patches As Better Bug Reports [Текст] / W. Weimer // Proceedings of the International Conference on Generative Programming and Component Engineering – 2006. – с.181-190.
56. V. Dallmeier Generating Fixes From Object Behavior Anomalies [Текст] / V. Dallmeier, A. Zeller, B. Meyer // Proceedings of the International Conference on Automated Software Engineering – 2009. – с.550-554.

57.F. Logozzo Modular and Verified Automatic Program Repair [Текст] / F. Logozzo, T. Ball // Proceedings of the 27th ACM International Conference on Object Oriented Programming Systems Languages and Applications – 2012. – c.133-146.

58.F. Logozzo Automatic Repair of Overflowing Expressions with Abstract Interpretation [Текст] / F. Logozzo, M. Martel // Semantics, Abstract Interpretation, and Reasoning About Programs: Essays Dedicated to David A. Schmidt on the Occasion of His Sixtieth Birthday. Vol. 129. – 2013. – c. 341–357.

59.Z. Coker Program Transformations to Fix C Integers [Текст] / Z. Coker, M. Hafiz // Proceedings of the International Conference on Software Engineering – 2013. – c. 792–801.

60.Q. Gao Safe Memory-leak Fixing for C Programs [Текст] / Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, H. Mei // Proceedings of the 37th International Conference on Software Engineering – 2015. – c. 459–470.

61.R. Gupta DeepFix: Fixing Common C Language Errors by Deep Learning [Текст] / R. Gupta, S. Pal, A. Kanade, S. Shevade // Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence – 2017. – c. 1345-1351

62.P. Muntean Automated Generation of Buffer Overflows Quick Fixes Using Symbolic Execution and SMT [Текст] / P. Muntean, V. K. Kommanapalli, A. Ibing, C. Eckert // International Conference on Computer Safety, Reliability & Security (SAFECOMP'15) – 2015. – c. 441-456.

63.D. Hovemeyer Finding Bugs Is Easy [Текст] / D. Hovemeyer, W. Pugh // Acm sigplan notices 39.12 -2004. – c. 92-106.

64.T. Azim Towards Self-healing Smartphone Software via Automated Patching [Текст] / T. Azim, I. Neamtiu, L. Marvel // Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering – 2014. – c. 623–628.

65. S. Sidiroglou Countering Network Worms Through Automatic Patch Generation [Текст] / S. Sidiroglou, A. Keromytis // Security & privacy 3.6 - 2005. – с. 41–49.
66. H. ETO Propolice: Improved Stacksmashing Attack Detection [Текст] / H. ETO, K. Yoda // Ipsj sig notes 75 – 2001. – с. 181–188.
67. Z. Lin AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair [Текст] / Z. Lin, X. Jiang, D. Xu, B. Mao, L. Xie // Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security – 2007. – с. 329–340.
68. T. Wang Diagnosis and Emergency Patch Generation for Integer Overflow Exploits [Текст] / T. Wang, C. Song, W. Lee // Detection of Intrusions and Malware, and Vulnerability Assessment – 2014. – с. 255–275.
69. G. Jin Automated Atomicity-violation Fixing [Текст] / G. Jin, L. Song, W. Zhang, S. Lu, B. Liblit // Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation – 2011. – с. 389–400.
70. P. Liu Axis: Automatically Fixing Atomicity Violations Through Solving Control Constraints [Текст] / P. Liu, C. Zhang // Proceedings of the 2012 International Conference on Software Engineering - 2012. – с. 299–309.
71. H. Liu Understanding and Generating High Quality Patches for Concurrency bugs [Текст] / H. Liu, Y. Chen, S. Lu // Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering. ACM – 2016. – с. 715–726.
72. H. Samimi Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving [Текст] / H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, L. J. Hendren // Proceedings of ICSE – 2012. – с. 277–287.
73. X. Wang Automating presentation changes in dynamic web applications via collaborative hybrid analysis [Текст] / X. Wang, L. Zhang, T. Xie, Y. Xiong, H. Mei // Proceedings of the acm sigsoft international symposium on the foundations of software engineering. ACM – 2012. – с. 16.

74. I. Medeiros Automatic detection and correction of web application vulnerabilities using data mining to predict false positive [Текст] / I. Medeiros, N. F. Neves, M. Correia // Proceedings of the 23rd international conference on world wide web. ACM – 2014. – c. 63– 74.
75. C. Liu R2Fix: Automatically Generating Bug Fixes From Bug Reports [Текст] / C. Liu, J. Yang, L. Tan, and M. Hafiz // Proceedings of the International Conference on Software Testing, Verification and Validation (ICST) – 2013. – c. 282–291.
76. L. A. Dennis Proof-directed Debugging and Repair [Текст] / L. A. Dennis, R. Monroy, P. Nogueira // Seventh Symposium on Trends in Functional Programming – 2006. – c. 131–140.
77. M. Jiang A Metamorphic Testing Approach for Supporting Program Repair without the Need for a Test Oracle [Текст] / M. Jiang, T. Y. Chen, F.-C. Kuo, D. Towey, Z. Ding. // Journal of systems and software – 2016. – c. 127-140.
78. Abstract Syntax Tree (AST) [Електронний ресурс] – Режим доступу: https://www.techopedia.com/definition/22431/abstract-syntax-tree-ast
79. Common Weakness Enumeration (CWE) [Електронний ресурс] – Режим доступу: https://cwe.mitre.org/index.html
80. Neural Network [Електронний ресурс] – Режим доступу: https://deepai.org/machine-learning-glossary-and-terms/neural-network
81. Introduction to Recurrent Neural Network [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/
82. A. Moawad The magic of LSTM neural networks [Електронний ресурс] A. Moawad - Режим доступу: https://medium.com/datathings/the-magic-of-lstm-neural-networks-6775e8b540cd
83. Gated Recurrent Unit Networks [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/gated-recurrent-unit-networks/
84. S. Kostadinov Understanding Encoder-Decoder Sequence to Sequence Model [Електронний ресурс] S. Kostadinov – Режим доступу:

https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346

85. D. Bahdanau Neural machine translation by jointly learning to align and translate [Електронний ресурс] D. Bahdanau, K. H. Cho, Y. Bengio – Режим доступу: https://arxiv.org/pdf/1409.0473.pdf.
86. Neural machine translation with attention [Електронний ресурс] – Режим доступу: https://www.tensorflow.org/tutorials/text/nmt_with_attention#write_the_encoder_and_decoder_model
87. A. Chernousov Deep learning based automatic software defects detection framework [Текст] / A. Chernousov, A. Savchenko, S. Osadchyi, Y. Kubiuk, Y. Kostenko, D. Likhomanov // THEORETICAL AND APPLIED CYBERSECURITY v.1.1 – 2019. – c. 68-74.
88. GitHub Microsoft Terminal, Check null pointer before fclose [Електронний ресурс] – Режим доступу: https://github.com/microsoft/Terminal/commit/99555ef9e9ba89b03bbeedf238b7e65375775b56