DOI: https://doi.org/10.36910/6775-2524-0560-2025-61-22
УДК 004.451
Мельник Василь Михайлович[1], к. ф-м. н., доцент
http://orcid.org/0000-0001-8282-6639
Багнюк Наталія Володимирівна[2], к.т.н., доцент
https://orcid.org/0000-0002-7120-5455
Корчук Оксана Ігорівна[1], викладач-методист
[1]Волинський фаховий коледж національного університету харчових технологій, Луцьк, Україна
[2] Луцький національний технічний університет, Луцьк, Україна

ПОШУК ПРИХОВАНИХ ПОМИЛОК ВИКОРИСТАННЯ ПАМ'ЯТІ ПРОГРАМНИМ КОДОМ C++ ЗА ДОПОМОГОЮ СТАТИЧНОГО АНАЛІЗУ DEEP LEARNING

Мельник В. М., Багнюк Н. В., Корчук О. І. Пошук прихованих помилок програмного використання пам'яті програмним кодом C++ за допомогою статичного аналізу Deep Learning. Використання пам'яті програмним кодом в наш час є однією з важливих проблем як зі сторони написання коду, так і зі сторони захисту застосунків. Економія ресурсів та безпечний код сьогодні відіграють важливу роль, коли ще перед промисловим впровадженням програм не можна виявити традиційними засобами помилок використання пам'яті. Невиявлені помилки використання пам'яті можуть спричинити як непотрібні затрати ресурсів, так і стати причиною вторгнень чи несанкціонованого заволодіння інформацією. В даній статті викладено основні переваги, налаштування, моделювання, спеціальні особливості вивчення коду та інші аспекти застосування статичного аналізу Deep learning на базі нейронних мереж для виявлення помилок використання пам'яті програмним кодом C++. Важливу роль у його використанні займає практичне вивчення проаналізованих кодових зразків з відомими проблемами розподілу пам'яті, які вже були виконані, збережені в базі і застосовуються як базові шаблони. На основі дослідження великої кількості коду з кількох програмних проєктів засобами Deep learning здійснено виявлення помилок, які не виявлялися іншими традиційними методами, а також проведено обговорення його продуктивності, візуалізації отриманих результатів та зменшення хибно-позитивних проявів.
Ключові слова: статичний аналіз коду, використання пам'яті, пошук помилок, хибно-позитивні результати, засоби Deep learning.

Melnyk V., Bahniuk N., Korchuk O. Hidden errors finding in memory programming usage for C++ code by means of static Deep Learning analysis. In our time, a memory usage by program code is one of the important problems both: from the side of code writing and from the side of application protection. Resource saving and secure code play an important role today, even before the industrial programs implementation, when memory usage errors cannot be detected by traditional means. Undetected memory usage errors can cause unnecessary resource consumption as well as become a source of intrusions or unauthorized information seizure. This article outlines the main advantages, settings, modeling, code study special features and other aspects of applying the Deep learning static analysis based on neural networks to detect memory usage errors in C++ program code. Some important role in its usage applies the practical study of have already been executed analyzed code samples with known memory allocation problems, saved in the database and used as basic templates. Based on the large amount of code study examples by Deep learning means from several software projects were used to detect errors that were undetected by other traditional methods, and its performance, obtained results visualization, and reduction of false positives were also discussed here.
Keywords: static code analysis, memory usage, error detection, false positives, Deep learning means.

Постановка наукової проблеми та аналіз останніх досліджень і публікацій. Помилки програмного використання пам'яті в застосунках C++ можуть спричиняти їх збої в роботі, відкриваючи перед зловмисником різноманітні вразливості безпеки [1-4]. Подібні проблеми також можуть обумовлювати і непередбачувану поведінку самого застосунку. Традиційні статичні аналізатори часто пропускають незначні помилки у використанні пам'яті, які можуть з'являтися лише за певних умов. Застосування Deep learning та його аналітичних засобів пропонує потужне вирішення цієї проблеми за допомогою статичного аналізу коду [5].

Слід відмітити, що виявлення витоків пам'яті на промисловому рівні ще досі недостатньо вивчене, хоча протягом останніх десятиліть для цього прикладаються величезні зусилля як зі сторони промислового впровадження застосунків, так і з наукової сторони. Відмічається [2], що високоточний аналіз обмежує масштабованість застосунку, а неточний – серйозно перешкоджає його точності або повноті роботи.

Статичний аналіз коду C++ у поєднанні з нейронними мережами може виявити приховані помилки використання пам'яті, виявляючи їх на мільйонних прикладах зразків коду [6]. Цей підхід виявляє основні закономірності та потенційно приховані проблеми, які пропускають системи тестування та аналізу коду застосунку, що базуються на традиційних аналітичних правилах.

Метою даної статті є опис можливості реалізації статичного аналізу на основі Deep learning для проєктів C++ з метою виявлення помилок програмного використання пам'яті перед тим, як вони

потраплять у виданий для публічного використання програмний застосунок. В даній статті також експериментально підтверджено, що застосування Deep learning значно покращує статичний аналіз коду C++, виявляючи помилки програмного використання пам'яті перед розгортанням застосунку, заощаджуючи час розробника та покращуючи безпеку коду [5, 7].

Відомо [8], що в ході проведення аналізу програмного коду помилки пам'яті в окремих випадках досить важко виявити. Зокрема відмічається, що керування пам'яттю також залишається одним із найскладніших завдань C++. Найбільш поширені проблеми з використанням і управлінням пам'яті включають багато важливих аспектів. Важливим є використання пам'яті після її звільнення і організація доступу до неї в наступних процесах (програмах). Наступна проблема з'являється під час переповнення буфера пам'яті, що призводить до запису даних за межі виділеної пам'яті. І ще одна проблема проявляється під час невдалого звільнення невикористаної пам'яті, яка відкриває можливості її витоку, а саме – помилкова спроба звільнити ту саму пам'ять двічі та доступ до пам'яті через розіменування нульових вказівників [7].

Сьогодні широко застосовуються традиційні статичні аналізатори, які зазвичай використовують попередньо визначені правила для виявлення подібних проблем [9]. Проте вони стикаються з такими проблемами, як складною арифметикою вказівників у програмному коді, шаблонами умовного розподілу пам'яті, багатопотоковими взаємодіями та використанням пам'яті в залежності від кодового контексту.

Виконання статичного аналізу Deep learning для C++

Deep learning вирішує вищеописані проблеми та недоліки традиційних методів, аналізуючи код як шаблони, а не дотримуючись жорстких правил. Його робота полягає в реалізації наступних кроків, описаних нижче. Даний метод використовує нейронні мережі для розуміння програмного коду. Однак сьогодні не тільки Deep learning, але й інші сучасні інструменти аналізу вихідного коду C++ використовують спеціалізовані нейронні мережі для його обробки. До них можна віднести такі як SonarQube, Sourcery AI, CodeClimate, CodeRabbit та інші, які для високоточного вирішення поставлених перед ними завдань використовують і засоби штучного інтелекту [10]. Спеціалізовані нейронні мережі спочатку перетворюють код у числове представлення або вбудовування. Наступним кроком проводиться аналіз зв'язків між змінними та функціями. Далі вони відстежують розподіл пам'яті за покроковим виконанням коду, відзначаючи підозрілі їм шаблони на основі вивчених даних.

На відміну від традиційних аналізаторів, вивчення коду засобами Deep learning на сьогодні все більше утверджується за допомогою практичного вивчення вже мільйонів проаналізованих кодових зразків з відомими проблемами розподілу пам'яті. Такий підхід допомагає цьому аналізатору розпізнавати ледь помітні та приховані закономірності, які вказують безпосередньо на потенційні помилки.

```
/*  Приклад коду з незначним витоком пам'яті,
який можуть пропустити традиційні аналізатори. */

void processData(size_t size)
{
   char* buff = new char[size];

   if(size > MAX_SIZE)
   {
      /* Логічна помилка, обумовлена передчасним поверненням
      із функції без звільнення пам'яті. */
      return;
   }

   // Обробка даних...
   delete[] buff;
}
```

Аналізатор на основі нейронної мережі може дізнатися, що будь-який ранній шлях повернення з функції після виконання розподілу пам'яті створює потенційний витік. Це має місце навіть у складних потоках управління.

Наведемо перелік найкращих інструментів для виконання статичного аналізу Deep learning для коду C++. Першим засобом являється DeepCode, спрямований на безпеку використання пам'яті, який бере на себе відповідальність за розпізнавання зразків з відкритого вихідного коду програми, поєднуючись із засобами GitHub, GitLab та IDE-плагінами [11]. Другим засобом Deep learning є CodeGuru, який виконує управління ресурсами. Він підтримує процес вивчення ситуації з посиланням на кодову базу AWS, інтегруючись з CI/CD та direct API. Наступним засобом є Infer Neural, який проводить аналіз вказівників шляхом залучення графових нейронних мереж для їх відстеження, поєднуючи роботу з командним рядком терміналу операційної системи під час побудови застосунку. І останнім засобом є MemSafe AI, який виконує аналіз купи, залучаючи моделі трансформації для шаблонів розподілу та інтегруючи з IDE плагінами та Docker.

Налаштування статичного аналізу Deep Learning

Щоб інтегрувати виконання статичного аналізу на основі Deep Learning з проєктом C++, слід виконати наступні кроки:

1) вибрати правильний інструмент на основі необхідних для проєкту конкретних вимог, враховуючи його розмірність, потреби в інтеграції з ресурсами, конкретні проблеми з пам'яттю, які потрібно виявити, та доступність до обчислювальних ресурсів;

2) встановити необхідний ресурс. Для даного випадку необхідно використовувати Infer Neural, який поєднує Infer від Facebook з моделями нейронних мереж [11];

```
# Install prerequisites
sudo apt-get install cmake libboost-dev

# Download and install Infer Neural
git clone https://github.com/facebook/infer.git
cd infer
./build-infer.sh --neural-models

# Add to PATH
export PATH=$PATH:$(pwd)/infer/bin
```

3) виконати налаштування для обраного проєкту, тобто створити файл конфігурації, який буде визначати ті нейронні моделі, які будуть використані у вирішенні проблеми використання пам'яті [12];

```
{
  "memory_analysis": {
    "model_path": "/path/to/memory_model",
    "confidence_threshold": 0.85,
    "analyze_external_libs": false
  },
  "pointer_tracking": {
    "model_path": "/path/to/pointer_model",
    "enable_deep_paths": true
  }
}
```

4) виконати сам аналіз коду.

```
Integrate the analyzer into your build process:
# To CMake projects
infer-neural –config = infer - config.json –neural - mem --cmake  --build

# For Make projects
infer-neural –config = infer-config.json --neural-mem – make
```

Розглянемо, як статичний аналіз Deep Learning виявляє помилки, які традиційні інструменти в процесі перевірки просто пропускають. Наведемо приклад витоку пам'яті між функціями.

```
// файл memory_manager.cpp
void allocateBuff(char** buff, size_t size)
{
   *buff = new char[size];
```

```
}

// файл processor.cpp
void processData(size_t dataSize)
{
   char* dataBuff;
   allocateBuff(&dataBuff, dataSize);

   if(validateData(dataBuff, dataSize))
   {
      processValidData(dataBuff);
   }

   // Відсутній оператор delete[] dataBuff у вибраній директорії
}
```

Традиційний аналізатор виводить повідомлення:

No issues found in processor.cpp

Аналізатор Deep learning видає наступне повідомлення:

HIGH CONFIDENCE (94%): Potential memory leak in processor.cpp: 10

- dataBuff allocated in allocateBuff() is not freed in all execution paths;
- Learned pattern: cross-file allocation with incomplete deallocation paths;
- Suggested fix: Add delete[] dataBuff before function exit.

В ході виконання аналізу нейронна мережа чітко розпізнає зв'язок між виділенням та звільненням динамічної пам'яті через функціональні межі.

Наведемо найбільш поширені закономірності помилок пам'яті, які були виявлені за допомогою Deep learning. Одним із поширених витоків пам'яті є витоки через помилкове застосування *умовного виразу* в коді програми. Deep learning чудово справляється з пошуком подібних витоків в програмних умовних розгалуженнях. Наведемо приклад виявлення такої ситуації:

```
if(condition)
{
   resource = allocate();
}
else
{
   // Немає розподілення ресурсу!
}
// Використання ресурсу без перевірки його розподілення.
```

Наступною є проблема безпеки потоків, яку нейронні мережі можуть ідентифікувати як операцію із пам'яттю, небезпечну для них. Наведемо типовий код помилкового застосування потоків:

```
// Thread 1
if(!initialized)
{
   buff = new char[SIZE];
   initialized = true;
}

// Thread 2 – умова зростаючої проблеми
if(!initialized)
{
   buff = new char[SIZE];
   initialized = true;
}
```

Також наведемо приклад нижче складних передавань права власності, які моделі Deep learning відстежують як володіння пам'яттю в складних потоках коду:

```
void transferOwnship(Resource** dest)
{
   // Право власності передане, але не є зрозумілим
   // для статичного аналізу!
   *dest = source;
   source = nullPtr;
}
```

Визначення моделей для виявлення помилок, характерних для проєкту

Вивчення нейронних моделей використання для великих кодових баз може покращити точність виявлення помилок використання пам'яті, поява яких має прояв під час створення проєкту. Для вивчення їх прояву необхідно виконувати наступні дії:

1) збір навчальних даних на основі збору та аналізу історії помилок та виправлень з проєкту розробки;

2) підготовка зразків коду, яка включає його перетворення з відомими помилками на матеріал вивчення та навчальні зразки;

3) удосконалення існуючих попередньо вивчених моделей з їх адаптацією до конкретних діючих шаблонів, задіяних в проєкті;

4) перевірка відомих проблем та проведення тестування моделей на наявність раніше виправлених помилок.

Відмітимо, що моделі користувача найкраще працюють для випадків, коли у проєкті розробки є наявними певні шаблони або вимоги до управління пам'яттю.

Інтеграція з конвеєром CI/CD

CI/CD конвеєр – це автоматизований від початку до кінця робочий конвеєрний процес, який проводить зміни коду через повний цикл доставки програмного забезпечення. Він також зв'язує етапи безперервної інтеграції, тестування та розгортання для безперервної розробки, подальшого тестування та випуску оновлень застосунків. Для того, щоб виявити помилки пам'яті на ранній стадії розробки, необхідно інтегрувати статичний аналіз Deep learning у свій конвеєр CI/CD [11]:

```
# Приклад робочого процесу дій
name: C++ Static Analysis

on: [push, pull_request]

jobs:
  static-analysis:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up C++ environment
        uses: actions/setup-cpp@v1

      - name: Install Infer Neural
        run: |
          wget https://github.com/facebook/infer/releases/ download/v1.2.0/infer-neural-linux.tar.gz
          tar -xf infer-neural-linux.tar.gz
          echo "$(pwd)/infer-neural/bin" >> $GITHUB_PATH

      - name: Run deep learning static analysis
        run: infer-neural --neural-mem --output-json=results.json -- make

      - name: Check for high-confidence issues
        run: python .github/scripts/check_memory_issues.py results.json
```

```
# Приклад робочого процесу дій
name: C++ Static Analysis
on: [push, pull_request]

jobs:
  static-analysis:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up C++ environment
        uses: actions/setup-cpp@v1

      - name: Install Infer Neural
        run: |
          wget https://github.com/facebook/infer/releases/download/v1.2.0/infer-neural-linux.tar.gz
          tar -xf infer-neural-linux.tar.gz
          echo "$(pwd)/infer-neural/bin" >> $GITHUB_PATH

      - name: Run deep learning static analysis
        run: infer-neural --neural-mem --output-json = results.json -- make

      - name: Check for high-confidence issues
        run: python .github/scripts/check_memory_issues.py results.json
```

Обговорення виконання статичного аналізу Deep learning

Необхідно відмітити, що аналіз Deep learning вимагає більше обчислювальних ресурсів, ніж традиційний статичний аналіз, що може обумовлювати деякі додаткові затрати. Нижче в таблиці наведено окремі показники ресурсів для реалізації його виконання.

Таблиця 1. Показники ресурсів для реалізації аналізу Deep learning

Ресурс	Типова вимога	Метод оптимізації
Пам'ять	8-16 GB оперативної пам'яті	Поступовоо виконується аналізу коду
CPU	Чотири і більше ядер	Запуск збірок на сервері, а не на локальному комп'ютері
Диск	2-5 GB - для реалізації моделей	Використання спільного репозиторію моделей
Час	2-5х довше, ніж традиційний аналіз	Запуск аналізу лише на критичних шляхах коду

Для великих проєктів необхідно врахувати ще і наступні вимоги:

– використання і запуск аналізу лише для змінених файлів;

– використання функцій інкрементального аналізу;

– планування повного аналізу у вигляді щодобових тестових збірок.

Для візуалізації результатів статичного аналізу Deep learning наведемо типовий приклад його практичного застосування, використавши наведений вище код з динамічним об'єктом у файлі processor.cpp. Наперед відмітимо, що в наведеному прикладі C++ коду виявлено помилку використання пам'яті, яка, відповідно, відзначена на наведеному нижче рисунку 1 відсутністю оператора delete[] для її звільнення.

Сучасні інструменти забезпечують візуалізацію виявлених проблем використання пам'яті, висвітлюючи при цьому такі дані, як графіки викликів із виділеними шляхами виникнення проблем, точки розподілу та звільнення пам'яті, рівні достовірності нейронної мережі для кожного виявлення та історичні закономірності подібних помилок

Моделі Deep learning можуть також генерувати хибно-позитивні результати, кількість яких можна значно зменшувати наступним шляхом:

1) застосовувати встановлення порогів достовірності і повідомляти тільки про проблеми, які перевищують певні рівні достовірності;

2) використовувати вивчення проблематики, орієнтованої на *конкретний проєкт розробки*, добиваючись точного налаштування робочих моделей перевірки у напрацьованій кодовій базі;

3) використовувати включення зворотного зв'язку, відтворюючи позначення хибно-позитивних результатів для підвищення точності робочої моделі перевірки на помилки;

4) поєднувати методи перевірки, які базуються на використанні як нейронних мереж, так і традиційного аналізу.

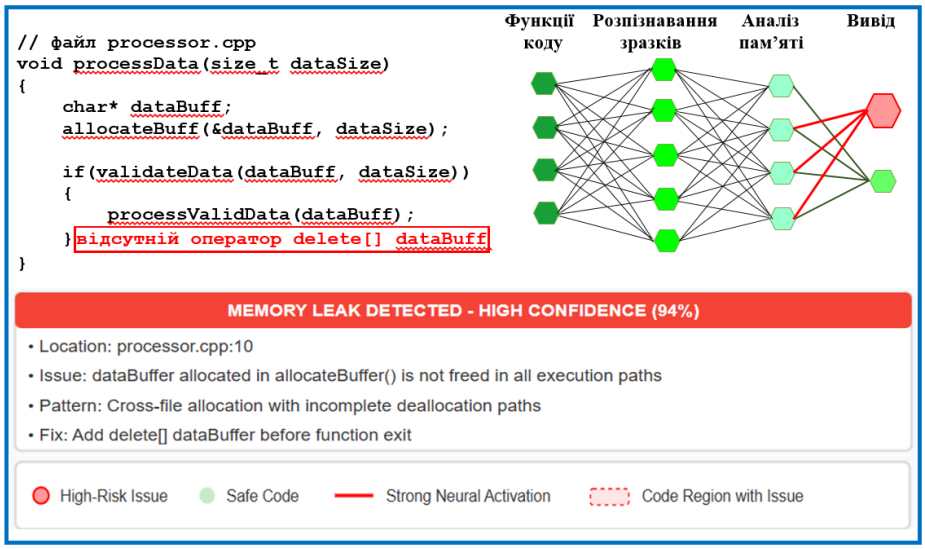

Рис.1. Виявлення помилок пам'яті за допомогою Deep Learning

Використовуючи статичний аналіз Deep learning, на протязі тривалого часу проведено тематичне дослідження зменшення помилок програмного використання пам'яті у порівняно великих масштабах.

Для кількох великих проєктів спорідненої тематики, які були створені програмними засобами C++ і в цілому становили понад 1,9 мільйонів рядків коду, реалізовано статичний аналіз Deep learning. Після детального аналізу та перегляду результатів отримано наступні показники поліпшення використання пам'яті, що використовувалася даними проєктами: зменшення кількості збоїв роботи проєктів, пов'язаних з використанням пам'яті, досягло понад 71 %; на 89 % зросла точність виявлення реальних помилок використання пам'яті програмним кодом. При цьому порівняно з ручною перевіркою коду більше ніж у 3 рази пришвидшилося виявлення помилок, а хибних спрацьовувань поменшало більше ніж на 40 %, порівнюючи цей показник для традиційних інструментів.

Для отримання вищевикладених показників і проведення якісного статичного аналізу Deep learning були витримані й інші ключові фактори успіху. До них відносились: налаштування моделей відповідно до шаблонів застосування, характерних для кожного проєкту, постійне їх перевизначення та вивчення з використанням нових даних про помилки, поступовий аналіз коду, який зазнав змін, та інші фактори впливу.

Висновок. Статичний аналіз Deep learning значно покращує використання пам'яті кодом C++ за рахунок знаходження прихованих помилок, які традиційні інструменти не виявляють. Даний підхід реалізується на основі застосування реальних шаблонів коду та їх аналітичного співставлення, з метою виявлення складних проблеми з використанням пам'яті застосунками, перш ніж вони спричинять проблеми в ході їх випуску та публічному використанні.

За рахунок поєднання статичного аналізу Deep learning та нейронних мереж у робочий процес розробки проєкту можна виявляти помилки використання пам'яті значно раніше, скорочуючи час налагодження та покращуючи якість C++ коду. Оскільки дані інструменти зазнають постійного розвитку, вони можуть відігравати важливу роль для будь-якого проєкту C++, надаючи

йому пріоритетності в надійності та безпеці. Отже, використання статичного аналізу Deep learning в наш час набирає високого розмаху для знаходження невловимих помилок пам'яті, а інколи і таких, які досить приховані у коді проєкту C++.

Список бібліографічного опису

1. Коваленко А.А., Куценко Т.Г., Шаповал А.С., Ситник О.В., Ні Я.С. Огляд методів аналізу безпечного програмного забезпечення. Control, Navigation and Communication Systems. Харків, ХНУР. 2025. №1. с.156-161.
2. G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, C. Zhang, "SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code", International Conference on Software Engineering (ICSE), 2019. URL: https://gangfan.github.io/assets/papers/ gang_smoke_icse2019_preprint.pdf (date of access: 12.06.2025).
3. Openssl. URL: https://www.openssl.org (date of access: 27.06.2025).
4. OSS-Fuzz: Continuous Fuzzing for Open Source Software. URL: https://github.com/google/oss-fuzz (date of access: 11.07.2025).
5. Heinrichs F., Heim M., Weber C. Functional Neural Networks: Shift invariant models for functional data with applications to EEG classification. 2023. URL: https://arxiv.org/abs/2301.05869 (date of access: 12.07.2025).
6. Гапон А.О., Федорченко В.М., Сєвєрінов О.В. Методи та засоби статичного та динамічного аналізу коду. Радіотехніка : Всеукр. міжвід. наук.-техн. зб. 2023. № 212. с. 7-13.
7. CWE-762: Mismatched Memory Management Routines. 2025. URL: https://cwe.mitre.org/data/definitions/762.html (date of access: 12.07.2025).
8. How do you guys deal with memory bugs? URL: https://www.reddit.com/r/cpp/comments/r5dkfc/how_do_you_guys_deal _with_memory_bugs/ (date of access: 24.07.2025).
9. Лукас Беннетт. 8 Найкращих інструментів статичного аналізу коду. 2025. URL: https://www.guru99.com/uk/best-static-code-analysis-tools.html (date of access: 24.07.2025).
10. 12 найкращих аналізаторів коду ШІ. URL: https://www.morningdough.com/uk/ai-tools/best-ai-code-analyzers/ (date of access: 17.08.2025).
11. GitHub: Infer Neural. 2025. URL: https://github.com/facebook/infer/tree/main (date of access: 17.08.2025).
12. C++ Memory Model Documentation Memory model. cppreference.com. 2025. URL: https://en.cppreference.com/w/cpp/language/memory_model.html (date of access: 12.09.2025).

References

1. Kovalenko A., Kutsenko T., Shapoval A., Sytnyk O., Ni J. Secure software analysis methods overview. Control, Navigation and Communication Systems. Kharkiv. KhNUR. 2025. №1. p.156-161.
2. G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, C. Zhang, "SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code", International Conference on Software Engineering (ICSE), 2019. URL: https://gangfan.github.io/assets/papers/ gang_smoke_icse2019_preprint.pdf (date of access: 12.06.2025).
3. Openssl. URL: https://www.openssl.org (date of access: 27.06.2025).
4. OSS-Fuzz: Continuous Fuzzing for Open Source Software. URL: https://github.com/google/oss-fuzz (date of access: 11.07.2025).
5. Heinrichs F., Heim M., Weber C. Functional Neural Networks: Shift invariant models for functional data with applications to EEG classification. 2023. URL: https://arxiv.org/abs/2301.05869 (date of access: 12.07.2025).
6. Hapon A., Fedorchenko V., Severinov O., Methods and tools for static and dynamic code analysis. Radiotechnic : All-Ukrainian Interdepю Scie. and Tech. Collection. 2023. № 212. p. 7–13.
7. CWE-762: Mismatched Memory Management Routines. 2025. URL: https://cwe.mitre.org/data/definitions/762.html (date of access: 12.07.2025).
8. How do you guys deal with memory bugs? URL: https://www.reddit.com/r/cpp/comments/r5dkfc/how_do_you_guys_deal _with_memory_bugs/ (date of access: 24.07.2025).
9. Lukas B. 8 Best Static Code Analysis Tools. 2025. URL: https://www.guru99.com/uk/best-static-code-analysis-tools.html (date of access: 24.07.2025).
10. 12 Best Code Analyzers AI. URL: https://www.morningdough.com/uk/ai-tools/best-ai-code-analyzers/ (date of access: 17.08.2025).
11. GitHub: Infer Neural. 2025. URL: https://github.com/facebook/infer/tree/main (date of access: 17.08.2025).
12. C++ Memory Model Documentation Memory model. cppreference.com. 2025. URL: https://en.cppreference.com/w/cpp/language/memory_model.html (date of access: 12.09.2025).