УДК 004.43:004.415.5 **DOI: 10.15330/itee.2026.4.09**

Р. І. ЗАПУХЛЯК, канд. фіз.-мат. наук, В. В. ДОВГИЙ, канд. техн. наук,

В. О. КРАСНОПОЛЬСЬКИЙ

ЕМПІРИЧНИЙ АНАЛІЗ ВРАЗЛИВОСТЕЙ ПАМ'ЯТІ В C++ ІЗ ВИКОРИСТАННЯМ ЗАСОБІВ ДИНАМІЧНОГО ТА СТАТИЧНОГО АНАЛІЗУ

Вступ

У сучасних програмних системах питання захисту інформації тісно пов'язане з надійністю керування пам'яттю, особливо у мовах системного програмування, таких як C та C++. Незважаючи на високу продуктивність і гнучкість, ці мови залишають розробнику значну відповідальність за коректне управління ресурсами, що створює передумови для виникнення широкого спектра вразливостей. Значна частина критичних інцидентів кібербезпеки пов'язана саме з помилками роботи з пам'яттю [1, 2, 3], такими як переповнення буфера (buffer overflow), використання пам'яті після звільнення (use-after-free, UAF), подвійне звільнення виділеної пам'яті (double free), звернення за межі виділеного блоку, валідність вказівників та узгодженість часу життя об'єктів. Особливістю цих мов є відсутність вбудованих механізмів автоматичного контролю безпеки пам'яті, що обумовлено особливостями моделей керування пам'яттю в стандартах C та C++ [4 , 5].

Вразливості пам'яті займають важливе місце у класифікаціях вразливостей інформаційної безпеки, зокрема в рамках «Відкритого проєкту з безпеки веб застосунків» (Open Worldwide Application Security Project, OWASP) [6] та за класифікацією CWE (Common Weakness Enumeration) [7], і залишаються однією з основних причин компрометації програмних систем. У сучасних дослідженнях з кібербезпеки безпека пам'яті розглядається як один із базових принципів побудови безпечного програмного забезпечення та критично важлива складова сучасної інженерії безпеки програмного забезпечення [8]. Особливістю таких вразливостей є те, що вони можуть призводити як до аварійного завершення програм, так і до виконання довільного коду, що створює критичні ризики для конфіденційності, цілісності та доступності інформації [9], а також ці вразливості часто пов'язані з явищем невизначеної поведінки (Undefined behavior, UB), коли стандарт мови не визначає результат виконання програми, що може призводити до непередбачуваних наслідків [10].

У відповідь на ці виклики в сучасній практиці розробки застосовуються різні підходи до підвищення безпеки, зокрема використання динамічних інструментів аналізу (санітайзерів), статичного аналізу коду та методів фазинг-тестування. Аналізатор помилок роботи з пам'яттю (ASan) та аналізатор невизначеної поведінки (UBSan) дозволяють виявляти помилки під час виконання програми [11], тоді як статичні аналізатори дають змогу знаходити потенційно небезпечні конструкції ще на етапі компіляції [12]. Фазинг-тестування, у свою чергу, забезпечують автоматизоване генерування вхідних даних для виявлення неочікуваних сценаріїв виконання та широко застосовується у сучасних підходах до тестування безпеки [13].

Попри наявність великої кількості інструментів, питання їх практичної ефективності у реальних проєктах залишається відкритим. Зокрема, недостатньо дослідженим є співвідношення між результатами динамічного та статичного аналізу, а також їх здатність виявляти різні класи вразливостей у зрілих C++ проєктах з відкритим вихідним кодом [10, 14].

Частина попередніх результатів щодо ініціатив WG21 та базового санітайзер-аналізу була попередньо представлена авторами у [15]. У даній роботі дослідження суттєво розширено шляхом проведення порівняльного емпіричного аналізу GCC/Clang, нормалізації метрик та дослідження поведінки під час виконання санітайзер-конфігурацій.

Метою даної роботи є проведення емпіричного аналізу вразливостей пам'яті в C++-проєктах із використанням динамічних інструментів аналізу (санітайзерів), статичного аналізу та

елементів фазинг-тестування, а також оцінка їх ефективності для виявлення різних типів помилок.

Для досягнення поставленої мети сформульовано такі завдання дослідження як: систематизувати основні класи вразливостей пам’яті у C++ та їх місце у сучасних моделях загроз, проаналізувати підходи до їх виявлення та запобігання, розробити відтворювану методологію експерименту із застосуванням ASan, UBSan та статичного аналізу, а також розглядом можливостей фазинг-тестування, провести експериментальне дослідження на реальних C++ проєктах з відкритим вихідним кодом та порівняти ефективність різних інструментів з інтерпретацією отриманих результатів.

Аналіз вразливостей роботи з пам'яттю за методологією OWASP

Вразливості пам’яті відіграють фундаментальну роль у реалізації багатьох атак, описаних у класифікації OWASP Top 10 [6], хоча вони не завжди виділяються як окрема категорія але їх вплив проявляється у кількох ключових класах загроз.

Зокрема атака, що полягає у переповненні буфера може бути використана для реалізації так званої атаки типу ін’єкція (Injection, A03:2021), коли зловмисник отримує можливість виконання довільного коду [16]. Крім того, відсутність безпечних механізмів роботи з пам’яттю відповідає категорії небезпечного проектування (Insecure Design, A04:2021), оскільки проблема закладена на рівні архітектури системи. Вразливості, що пов’язані з неправильним керуванням пам’яттю, також можуть бути наслідком неправильного налаштування засобів безпеки (Security Misconfiguration, A05:2021), наприклад, у випадку відсутності механізмів захисту (stack canaries, ASLR), або використання застарілих компонентів (A06:2021) з відомими дефектами, що пов’язані з пам’яттю [6]. Таким чином, вразливості пам’яті виступають як базовий технічний механізм, що лежить в основі значної частини сучасних атак.

Якщо наводити фрагменти вихідного коду на C++, які демонструють згадані вище вразливості, то вони можуть виглядати наступним чином.

Переповнення буфера:

```
char buffer[10];
strcpy(buffer, input); // небезпечно
```

У цьому випадку у нас явна відсутність перевірки розміру вхідних даних, що може призвести до перезапису пам’яті за межами буфера та виконання довільного коду [8, 16].

Використання пам’яті після звільнення:

```
int* ptr = new int(10);
delete ptr;
*ptr = 20; // UB
```

Тут після звільнення пам’яті вказівник стає невалідним, але все ще використовується, що створює серйозну загрозу безпеці [1, 2].

Подвійне звільнення:

```
int* ptr = new int;
delete ptr;
delete ptr; // помилка
```

Подвійне звільнення пам’яті може призводити до пошкодження внутрішніх структур менеджера пам’яті, що створює можливості для атак [2, 7].

Цілочисельне переповнення:

```
int size = input * sizeof(int);
int* arr = (int*)malloc(size);
```

Переповнення цілих чисел може призвести до неправильного розрахунку розміру буфера, що в подальшому спричиняє переповнення пам’яті [7, 14].

Узагальнені дані щодо типів вразливостей та методів їх виявлення представлені у табл. 1.

Таблиця 1. Узагальнення типів вразливостей

Тип вразливості	Опис	Потенційні наслідки	Відповідність OWASP	Методи виявлення
Переповнення буфера (Buffer overflow)	Вихід за межі буфера	Виконання коду, аварійне завершення	A03, A04	ASan
Використання пам’яті після звільнення (Use-after-free, UAF)	Доступ після звільнення	UAF-експлойт, аварійне завершення	A04	ASan
Подвійне звільнення (Double free)	Повторне звільнення	Пошкодження купи	A05	ASan
Вихід за межі (Out-of-bounds)	Некоректний доступ	Пошкодження пам’яті	A03	ASan
Висячий (невалідний) вказівник (Dangling pointer)	Невалідний вказівник	Нестабільність	A04	Статичний аналіз
Витік пам’яті (Memory leak)	Втрата пам’яті	Вичерпання ресурсів	A05	Статичний аналіз
Цілочисельне переповнення (Integer overflow)	Переповнення типів	Неправильні розміри	A03	UBSan

Таким чином, вразливості пам’яті є системною проблемою мов низького рівня, що обумовлена поєднанням високої гнучкості та відсутності автоматичних гарантій безпеки [1, 2]. Вони залишаються однією з основних причин порушення безпеки програмних систем і широко представлені у міжнародних класифікаціях вразливостей та загроз [6, 7].

Складність їх виявлення пов’язана з тим, що багато з них проявляються тільки за певних умов виконання або залежать від конкретної реалізації компілятора та середовища виконання [10]. Це зумовлює необхідність використання комплексного підходу до їх виявлення, що включає як динамічні, так і статичні методи аналізу, які ми розглянемо нижче.

Методи та інструментарій захисту від вразливостей пам’яті

Захист від вразливостей пам’яті у C/C++ базується на поєднанні безпечного програмування, статичного та динамічного аналізу, а також системних механізмів захисту [1, 2]. Такий багаторівневий підхід дозволяє підвищити стійкість програмного забезпечення до вразливостей, пов'язаних із пошкодженням пам'яті.

До базових практик належать використання RAII, розумних вказівників, контейнерів STL та уникнення небезпечних функцій роботи з пам’яттю [2, 17].

У дослідженні використовувалися компілятори GCC та Clang, які забезпечують підтримку розширених компіляторних діагностик, санітайзерів та засобів статичного аналізу. GCC забезпечує широкий набір механізмів попереджень, тоді як Clang вирізняється більш деталізованою системою діагностики та інтеграцією зі статичними інструментами аналізу, зокрема clang-tidy.

Для динамічного аналізу використовувалися ASan та UBSan [11, 18, 19], які дозволяють виявляти дефекти пам'яті та прояви невизначеної поведінки під час виконання програми шляхом інструментування коду додатковими перевірками.

Перевагами цього підходу є висока точність та можливість аналізу реальних сценаріїв виконання, тоді як основним обмеженням виступає залежність від тестового покриття.

Фазинг-тестування є ефективним методом автоматизованого виявлення нетривіальних сценаріїв виконання та широко використовується у дослідженнях безпеки програмного забезпечення [13]. У сучасних системах фазинг часто комбінується із санітайзерами для підвищення ефективності виявлення дефектів пам'яті. У даній роботі фазинг-тестування розглядається як перспективний додатковий підхід до виявлення дефектів пам'яті, однак повноцінний фазинг-експеримент не входив до практичної частини дослідження.

Додатковий рівень захисту забезпечують механізми механізми захисту від експлуатації переповнень буфера [8].

Порівняльний аналіз цих інструментів узагальнено та представлено у табл. 2.

Таблиця 2. Узагальнені дані порівняння методів дослідження вразливостей пам'яті

Інструмент	Тип аналізу	Переваги	Обмеження
GCC	Статичний (warnings)	Швидкість, доступність	Обмежений аналіз
Clang	Статичний + розширений	Детальна діагностика	Потребує налаштування
clang-tidy	Статичний	Глибокий аналіз	Можливі хибнопозитивні результати
ASan	Динамічний	Висока точність	Потребує виконання
UBSan	Динамічний	Виявлення UB	Часткове покриття
AFL	Фазинг	Виявлення граничних помилок	Ресурсоємність

Таким чином, ефективне виявлення вразливостей пам'яті у C++ потребує комбінування статичних і динамічних методів аналізу. У роботі основна увага приділяється порівняльному аналізу GCC і Clang, а також практичній ефективності ASan та UBSan у реальних проєктах з відкритим кодом.

Методологія експерименту та результати досліджень

Метою експериментального дослідження було емпіричне оцінювання ефективності сучасних засобів виявлення помилок пам'яті та потенційно небезпечних конструкцій у C++-проєктах із використанням компіляторних діагностик, санітайзерів часу виконання та порівняльного аналізу різних конфігурацій інструментарію. Дослідження проводилося в реальному Linux-середовищі розробки з використанням відкритих бібліотек промислового рівня на C++.

Усі експерименти виконувалися на персональній робочій станції під керуванням операційної системи Ubuntu 24.04.4 LTS. Конфігурація середовища наведена у табл. 3.

Для автоматизації збірки та тестування використовувалися системи CMake та GNU Make. Паралельна збірка виконувалася із використанням усіх доступних потоків процесора (make -j$(nproc)).

Таблиця 3. Конфігурація експериментального середовища

Компонент	Характеристика
Операційна система	Ubuntu 24.04.4 LTS
Ядро Linux	6.8.0-117-generic
Архітектура	x86_64
Кількість потоків CPU	16
GCC	13.3.0
Clang/LLVM	18.1.3
CMake	3.28.3
GNU Make	4.3
Інструмент підрахунку LOC	cloc 1.98

Для експерименту були обрані три популярні бібліотеки відкритого коду мовою C++, які активно використовуються у сучасному програмному забезпеченні, які наведені у табл. 4.

Таблиця 4. Експериментальні бібліотеки відкритого коду мовою C++

Бібліотека	Призначення	Версія
fmt	Форматування рядків	12.1.1
spdlog	Логування	1.17.0
nlohmann/json	Серіалізація та десеріалізація JSON	актуальна master-версія

Обрані проєкти характеризувалися високою популярністю у C++-екосистемі, активною підтримкою, промисловим рівнем якості коду, наявністю автоматизованих тестових наборів, використанням сучасних практик розробки.

На першому етапі дослідження виконувалося статичне оцінювання кодових баз за допомогою компіляторних діагностик GCC та Clang. Для GCC використовувався такий набір warning-прапорів:

```
-Wall -Wextra -Wpedantic -Wshadow
-Wconversion -Wsign-conversion
-Wnull-dereference
-Wdouble-promotion
-Wformat=2
-Wduplicated-cond
-Wduplicated-branches
-Wlogical-op
-Wuseless-cast
```

Для Clang використовувався набір:

```
-Wall -Wextra -Wpedantic
-Wshadow
-Wconversion
```

Під час аналізу фіксувалися загальна кількість попереджень, типи діагностик, належність попереджень до промислового коду або тестової інфраструктури, характер попереджень (стилістичні, сумісності, логічні тощо).

На другому етапі дослідження виконувалося тестування під час виконання із використанням санітайзерів. Зокрема для GCC застосовувався ASan, а для Clang застосовувалися і ASan, і UBSan.

Конфігурація санітайзера для GCC мала вигляд:

```
-fsanitize=address
-g
-fno-omit-frame-pointer
```

а для Clang:

```
-fsanitize=address,undefined
-g
-fno-omit-frame-pointer
```

Під час виконання тестових наборів аналізувалися переповнення буфера в купі, переповнення буфера в стеку, використання пам’яті після звільнення, повторне використання пам’яті, некоректне звільнення пам'яті, невизначена поведінка під час виконання, аварійні завершення тестів, конфлікти середовища виконання санітайзерів.

Для кожного проєкту виконувалася така послідовність дій:

1. Конфігурація CMake-проєкту.
2. Збірка з використанням GCC попереджень.
3. Збірка з використанням Clang попереджень.
4. Збірка GCC + ASan.
5. Збірка Clang + ASan/UBSan.
6. Запуск автоматизованих тестових наборів через ctest.
7. Аналіз логів санітайзерів.
8. Підрахунок кількості рядків промислового коду.

Для деяких проєктів тестові цілі потребували явного ввімкнення через CMake-прапорів (SPDLOG_BUILD_TESTS=ON).

Для забезпечення коректного порівняння кодових баз різного розміру використовувалася нормалізація попереджень на 10 000 рядків промислового коду.

Підрахунок LOC виконувався за допомогою інструмента cloc лише для промислового коду C/C++ (include, src) без врахування документації, markdown-файлів, YAML-файлів, скриптів збирання, сторонніх тестових framework-компонентів.

Нормалізована метрика обчислювалася за формулою (1):

$$Warnings_{10kLOC} = \frac{Warnings}{LOC} x10000 \quad (1)$$

Експеримент мав низку обмежень, оскільки аналізувалися лише три проєкти відкритого коду, не проводилося символьне виконання, дослідження проводилося тільки на Linux-платформі, частина попереджень належала тестовій інфраструктурі, а не промисловому коду, відсутність зауважень від санітайзерів (sanitizer findings) не гарантувала повної відсутності вразливостей пам'яті.

Однак, попри зазначені обмеження, проведений експеримент дозволяє оцінити практичну ефективність сучасних підходів до виявлення дефектів пов’язаних з пам’яттю у реальних C++-проєктах.

На першому етапі експерименту виконувалося статичне оцінювання кодових баз за допомогою діагностик компіляторів GCC 13.3 та Clang 18.1.3, результати якого наведено у табл. 5.

Таблиця 5. Результати діагностик компіляторами

Проєкт	Кількість рядків промислового коду (LOC)	GCC попередження	Clang попередження	GCC попередження / 10k LOC	Clang попередження / 10k LOC
fmt	13 494	18	2	13.34	1.48
spdlog	20 730	0	0	0.00	0.00
nlohmann/json	18 421	0	0	0.00	0.00

Результати демонструють суттєву різницю між профілями GCC та Clang. Найбільшу кількість попереджень було зафіксовано у бібліотеці fmt, причому GCC згенерував у дев'ять разів більше порівняно з Clang. Аналіз попереджень показав, що більшість із них належали до неоднозначності конструкції потоку керування, дубльованих розгалужень, зайвих приведення типів, діагностиками макросів, які пов'язані з переносністю між архітектурами, стилістичних конструкцій. При цьому значна частина попереджень була локалізована у тестовій інфраструктурі, а не у промисловому коді бібліотеки.

Для spdlog та nlohmann/json компіляторні діагностики не зафіксували жодного попередження навіть при використанні розширених конфігурацій попереджень. Отримані результати узгоджуються із сучасними підходами до безпечного програмування (secure coding) та орієнтованої на якість розробки програмного забезпечення (quality-oriented software engineering) [2, 12, 17].

На другому етапі дослідження виконувалося тестування під час виконання із використанням ASan [11, 18]. Результати досліджень наведено у табл. 6.

У жодному з досліджуваних проєктів ASan не виявив переповнення буфера в купі, переповнення буфера в стеку, використання пам'яті після звільнення, повторного використання пам'яті, некоректне звільнення пам'яті.

Отримані результати свідчать про високий рівень стабільності під час виконання досліджуваних бібліотек та відсутність очевидних дефектів, що спричиняють пошкодження пам'яті у тестових сценаріях.

Таблиця 6. Результати ASan-тестування

Проєкт	Тестів	GCC + ASan	Clang + ASan	Виявлені дефекти пам'яті
fmt	21	20 із 21 тестів успішно виконано	20 із 21 тестів успішно виконано	0
spdlog	2	2 із 2 тести успішно виконано	2 із 2 тести успішно виконано	0
nlohmann/json	102	102 із 102 тестів успішно виконано	102 із 102 тестів успішно виконано	0

Особливо показовими є результати nlohmann/json, де повний тестовий набір із 102 тестів був успішно виконаний як під GCC + ASan, так і під Clang + ASan без жодного зауваження від санітайзерів.

Водночас результати fmt показали, що попередження компіляторів не обов'язково корелюють із фактичними дефектами пам'яті під час виконання. Незважаючи на наявність 18 GCC попереджень, ASan не виявив жодної помилки роботи з пам'яттю під час виконання.

Отримані результати узгоджуються з дослідженнями ефективності рантайм-аналізу на основі санітайзерів [1, 11, 18].

Для Clang-конфігурацій додатково використовувався UBSan [19]. У процесі виконання тестових наборів не було зафіксовано переповнення знакових цілих чисел, некоректних операцій зсуву, розіменування нульових вказівників, арифметичних операцій за межами допустимого діапазону, інших проявів невизначеної поведінки. Команда аналізу grep -oE "runtime error: .*" не повернула жодного динамічно виявленого дефекту для всіх досліджуваних проєктів. Отримані результати свідчать про високий рівень коректності під час виконання досліджуваних бібліотек та підтверджують ефективність сучасних практик безпечної розробки C/C++-програмного забезпечення [2, 17, 19].

У процесі Clang + ASan тестування бібліотеки fmt було зафіксовано помилку часу виконання:

Your application is linked against incompatible ASan runtimes

Аналіз показав, що проблема виникла через конфлікт компонентів середовища виконання санітайзерів GCC та Clang у змішаній конфігурації компілятора та середовища виконання. Важливо відзначити, що ця помилка не є помилкою вразливості пам'яті, вона не пов’язана з пошкодженням пам’яті, ASan не зафіксував рантайм-дефекту пам’яті. Проте цей результат демонструє важливу практичну проблему екосистеми санітайзерів — потенційну несумісність змішаних конфігурацій інструментарію [11, 20]. На відміну від Clang-конфігурації, зв’язка GCC + ASan не продемонструвала подібних проблем із сумісністю під час виконання.

Проведене експериментальне дослідження показало, що результати експерименту узгоджуються із сучасними дослідженнями вразливостей пам’яті та практичних методів їх виявлення у C/C++-екосистемі [1, 10, 14].

Оскільки експеримент проводився виключно на Linux-платформі, результати можуть відрізнятися для Windows, macOS, вбудованих систем та ARM-архітектур.

Висновки

У роботі проведено емпіричне дослідження ефективності сучасних засобів забезпечення безпеки програмного забезпечення та виявлення пов'язаних із пам'яттю вразливостей у C++-проєктах із використанням діагностики компілятора, аналізатора помилок роботи з пам'яттю та аналізатора невизначеної поведінки. Актуальність дослідження обумовлена тим, що вразливості пам’яті залишаються одним із найнебезпечніших класів загроз у сфері кібербезпеки та захисту інформації. Помилки типу переповнення буфера, використання пам'яті після звільнення, висячі вказівники та невизначена поведінка можуть призводити до несанкціонованого виконання коду, витоку конфіденційних даних, компрометації програмних систем та порушення цілісності інформаційних ресурсів.

У межах дослідження було проаналізовано три відкриті бібліотеки мовою C++ промислового рівня: fmt, spdlog та nlohmann/json. Експеримент проводився у реальному Linux-середовищі із використанням компіляторів GCC 13.3 та Clang 18.1.3, що забезпечило високу відтворюваність отриманих результатів та практичну наближеність дослідження до сучасних умов розробки програмного забезпечення.

На основі узагальнення результатів проведеного аналізу сформульовано такі основні висновки:

1. Встановлено, що діагностики компіляторів GCC та Clang демонструють суттєво різний рівень строгості та специфіки перевірки кодових баз. Компілятор GCC виявився схильним до генерації значно більшої кількості попереджень порівняно з Clang. Проте детальний розбір показав, що ці застереження мають переважно стилістичний характер або пов'язані з переносністю коду між архітектурами, не вказуючи на реальні дефекти безпеки. Для кодових баз із високим рівнем промислової зрілості такі попередження можуть бути повністю відсутні.

2. Динамічне тестування із застосуванням інструментів ASan та UBSan підтвердило високий рівень коректності та стабільності досліджуваних бібліотек під час виконання. У межах наявних тестових сценаріїв не було зафіксовано жодних випадків критичних дефектів (зокрема переповнення буферів, використання пам'яті після звільнення чи некоректного її оперування), а також ознак невизначеної поведінки.

3. Експериментально доведено відсутність прямої кореляції між наявністю статичних попереджень компілятора та фактичними вразливостями керування пам'яттю під час виконання програми. Це підкреслює, що статичні компіляторні діагностики не є достатнім ізольованим засобом контролю безпеки, тоді як використання рантайм-санітайзерів виступає ефективним і необхідним механізмом для верифікації цілісності пам'яті в сучасних проєктах на C++.

4. Виявлено практичне обмеження сучасної екосистеми динамічного аналізу, яке полягає у потенційній несумісності компонентів виконання (runtime incompatibility) між санітайзерами GCC та Clang у разі використання змішаних конфігурацій інструментаріїв збірки. Цей результат акцентує на необхідності суворої уніфікації та коректної інтеграції інфраструктури збирання й аналізу під час побудови безпечних конвеєрів CI/CD і впровадження практик DevSecOps.

Наукова новизна отриманих результатів полягає у комплексному емпіричному порівнянні ефективності діагностичних профілів компіляторів GCC та Clang у контексті безпеки пам'яті, застосуванні нормалізованих метрик (кількість попереджень на 10 000 рядків коду / LOC) для об’єктивного зіставлення різнорозмірних кодових баз, а також у систематизації рантайм-поведінки програм за одночасного використання ASan і UBSan та ідентифікації прихованих конфліктів сумісності змішаних інструментаріїв у реальному Linux-середовищі розробки.

Практичне значення роботи полягає у демонстрації високої ефективності гібридного (комбінованого) підходу, що інтегрує статичні діагностики компілятора, глибоке динамічне інструментування коду та автоматизовані набори тестів для превентивного усунення помилок роботи з пам'яттю у сучасних системах на мові C++. Запропонована методологія та аналітичні висновки можуть бути безпосередньо використані під час проектування життєвого циклу безпечної розробки (SDLC), створення захищених конвеєрів автоматизованої збірки (security pipelines) та впровадження DevSecOps-практик у промислових проєктах. Отримані результати також можуть бути використані для побудови більш стійкої та безпечної цифрової інфраструктури, що узгоджується з Ціллю сталого розвитку ООН №9.

Перспективами подальших досліджень у цьому напрямі є розширення експериментальної бази на ширший пул C++-проєктів (зокрема вбудованих та високопродуктивних систем), інтеграція засобів автоматизованого фазинг-тестування на основі покриття коду (coverage-guided fuzzing), а також вивчення специфіки безпеки пам'яті в межах новітніх стандартів C++23 та C++26.

Список літератури:

1. V. E. Moghadam, “Memory integrity techniques for memory-unsafe languages: A survey,” Scuola Superiore Sant'Anna, Pisa, Italy, Tech. Rep., 2024. [Online]. URL: https://iris.santannapisa.it/retrieve/9ebf4c27-86fb-401c-827b-6fad063f1d08/Memory_Integrity_Techniques_for_Memory-Unsafe_Languages_A_Survey.pdf
2. R. C. Seacord, Secure Coding in C and C++, 2nd ed. Boston, MA, USA: Addison-Wesley, 2013..
3. “The case for memory safe roadmaps,” Cybersecurity and Infrastructure Security Agency (CISA), CISA, Washington, DC, USA, Tech. Rep., 2023. [Online]. URL: https://www.cisa.gov/resources-tools/resources/case-memory-safe-roadmaps
4. “Programming Languages — C,” ISO/IEC 9899:2018, International Organization for Standardization, Geneva, Switzerland, 2018. [Online]. URL: https://www.iso.org/standard/74528.html .
5. “Programming Languages — C++,” ISO/IEC 14882:2024, International Organization for Standardization, Geneva, Switzerland, 2024. [Online]. URL: https://www.iso.org/standard/83626.html .
6. “OWASP Top 10: The ten most critical web application security risks,” OWASP Foundation, 2021. [Online]. URL: https://owasp.org/www-project-top-ten/
7. “CWE-119: Improper restriction of operations within the bounds of a memory buffer,” MITRE, Common Weakness Enumeration, 2024. [Online]. URL: https://cwe.mitre.org/data/definitions/119.html

8. D. S. Wallach and B. Lord, “A viewpoint: A memory safety manifesto,” *IEEE Secur. Privacy*, vol. 22, no. 4, pp. 18–21, Jul.–Aug. 2024. doi: https://doi.org/10.1109/MSEC.2024.3409934.
9. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, “Buffer overflows: Attacks and defenses for the vulnerability of the decade,” in *Proc. DARPA Inf. Survivability Conf. Expo. (DISCEX)*, Hilton Head, SC, USA, 2000, pp. 119–129. doi: https://doi.org/10.1109/FITS.2003.1264935 .
10. X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in C compilers,” in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI)*, San Jose, CA, USA, 2011, pp. 283–294. doi: https://doi.org/10.1145/1993498.1993532.
11. “AddressSanitizer documentation,” LLVM Project, Clang Documentation, 2024. [Online]. URL: https://clang.llvm.org/docs/AddressSanitizer.html .
12. A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kalgren, C. Kaner, F. Samsky, and D. Engler, “A few billion lines of code later: Using static analysis to find bugs in the real world,” *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010. doi: https://doi.org/10.1145/1646353.1646374.
13. “American fuzzy lop (AFL) fuzzer,” M. Zalewski, 2023. [Online]. URL: https://lcamtuf.coredump.cx/afl/
14. M. Verdi, D. C. D’Elia, and C. Demetrescu, “An empirical study of C++ vulnerabilities in crowd-sourced code examples,” arXiv:1910.01321, 2019. doi: https://doi.org/10.48550/arXiv.1910.01321.
15. R. I. Zapukhlyak and V. V. Dovhyi, “Memory safety in C++: WG21 initiatives (lifetime, contracts, safety profiles) and empirical analysis of open-source projects,” (in Ukrainian), Inf. Technol. Electron. Eng., no. 2, pp. 44–50, Jun. 2025. doi: https://doi.org/10.15330/itee.2025.2.05.
16. “Buffer overflow attack,” OWASP Foundation, OWASP Community, 2023. [Online]. URL: https://owasp.org/www-community/attacks/Buffer_overflow_attack
17. “C++ Core Guidelines,” B. Stroustrup and H. Sutter, Eds., Standard C++ Foundation, 2024. [Online]. URL: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
18. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *Proc. 2012 USENIX Annu. Tech. Conf. (USENIX ATC 12)*, Boston, MA, USA, 2012, pp. 309–318.
19. “UndefinedBehaviorSanitizer documentation,” LLVM Project, Clang Documentation, 2024. [Online]. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
20. N. S. P. K. Yadati, “Integrating dynamic security testing tools into CI/CD pipelines: A continuous security testing case study,” *Int. J. Sci. Res. (IJSR)*, vol. 10, no. 6, pp. 1403–1405, 2021. doi: https://doi.org/10.21275/SR24615152732.

Надійшла до редколегії 09.06.2026

Відомості про авторів:

Запухляк Руслан Ігорович – кандидат фізико-математичних наук, професор кафедри комп’ютерної інженерії та електроніки, Карпатський національний університети імені Василя Стефаника / Vasyl Stefanyk Carpathian National University, Україна; email: ruslan.zapukhlyak@cnu.edu.ua, ORCID: https://orcid.org/0000-0002-4204-8235

Довгий Віктор Володимирович – кандидат технічних наук, старший викладач кафедри комп’ютерної інженерії та електроніки, Карпатський національний університети імені Василя Стефаника / Vasyl Stefanyk Carpathian National University, Україна; email: viktor.dovhyi@cnu.edu.ua, ORCID: https://orcid.org/0009-0009-7158-6938

Краснопольський Вадим Олександрович – студент кафедри комп’ютерної інженерії та електроніки, Карпатський національний університети імені Василя Стефаника / Vasyl Stefanyk Carpathian National University, Україна; email: vadym.krasnopolskyi.25@cnu.edu.ua.