

Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

”Допущено до захисту”
Завідувач кафедри кібербезпеки
та програмного забезпечення
д.т.н., професор
_____Олексій СМІРНОВ
“_____” _____20 ____ р.

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА
за першим (бакалаврським) рівнем вищої освіти
на тему

**“Програмне забезпечення системи кібербезпеки для виявлення та
запобігання SQL-ін'єкціям у веб-застосунках”**

Виконав здобувач вищої освіти
IV курсу, групи КБ-22-1
ОПП «Кібербезпека»
спеціальності 125 «Кібербезпека»
_____Штейніков М.С.
«_____» _____20 ____ р.

Керівник проекту
кандидат технічних наук, доцент
_____Улічев О.С.
«_____» _____20 ____ р.
Рецензент _____

Центральноукраїнський національний технічний університет
Факультет Механіко-технологічний
Кафедра Кібербезпеки та програмного забезпечення
Рівень вищої освіти бакалавр
Галузь знань . _12_ “Інформаційні технології”
Спеціальність 125 “Кібербезпека”
Освітньо-професійна (освітньо-наукова) програма “ Кібербезпека ”

ЗАТВЕРДЖУЮ

Завідувач кафедри

д.т.н., проф.

Олексій СМІРНОВ

“ _____ ” _____ 20__ року

**ЗАВДАННЯ НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ РОБОТУ ЗА
ПЕРШИМ (БАКАЛАВРСЬКИМ) РІВНЕМ ВИЩОЇ ОСВІТИ
ЗДОБУВАЧА ВИЩОЇ ОСВІТИ**

Штейнікова Микити Сергійовича

(прізвище, ім'я, по батькові)

1. Тема роботи Програмне забезпечення системи кібербезпеки для виявлення та запобігання SQL-ін'єкціям у веб-застосунках.

2. Керівник роботи Улічев Олександр Сергійович, канд. техн. наук, доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від “27” лютого 2026 року №134-02

3. Строк подання роботи до захисту 22.05.2026 р.

4. Мета та завдання випускної кваліфікаційної роботи Метою роботи є розробка програмного забезпечення системи кібербезпеки для виявлення та запобігання SQL-ін'єкціям у веб-застосунках

5. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) _____

1. Призначення та область використання.

2. Перегляд аналогічних існуючих систем.

3. Опис і обґрунтування проектних рішень.

4. Етапи програмування системи.

5. Впровадження системи в промислову експлуатацію

6. Висновки.

6. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Структурна схема системи 1 аркуш

Функціональна схема системи 1 аркуш

Діаграма процесів 1 аркуш

Блок-схема алгоритму роботи додатку 3 аркуша

7. Дата видачі завдання «27» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти	Строк виконання етапів випускної кваліфікаційної роботи за першим (бакалаврським) рівнем вищої освіти	Примітка
1.	Аналіз існуючих систем керування	10.03.2026 р.	
2.	Постановка задачі, оформлення ТЗ	15.03.2026 р.	
3.	Розробка моделі компонента	20.03.2026 р.	
4.	Розробка структур даних	25.03.2026 р.	
5.	Розробка алгоритмів зв'язку та відображення	30.03.2026 р.	
6.	Програмування алгоритмів	10.04.2026 р.	
7.	Оформлення ПЗ	17.05.2026 р.	
8.	Попередній захист роботи	22.05.2026 р.	

Дата видачі завдання
«27» лютого 2026 р.

Підпис керівника

(прізвище та ініціали)

Завдання прийнято до виконання
«27» лютого 2026 р.

Підпис здобувача

(прізвище та ініціали)

АНОТАЦІЯ

Штейніков М.С. Програмне забезпечення системи кібербезпеки для виявлення та запобігання SQL-ін'єкціям у веб-застосунках. 125 Кібербезпека. Центральноукраїнський національний технічний університет. Кропивницький. 2026.

В даній кваліфікаційній бакалаврській розроблено програмне забезпечення, яке призначено для виявлення та запобігання SQL-ін'єкційним атакам у веб-застосунках на платформі ASP.NET Core.

Метою розробки є програмне забезпечення системи кібербезпеки для виявлення та запобігання SQL-ін'єкціям у веб-застосунках.

Результат роботи – програмна реалізація системи кібербезпеки у вигляді middleware-компонента для ASP.NET Core 10 з гібридним механізмом детекції на основі сигнатурного аналізу, моделей машинного навчання та композитного скорингу.

В процесі роботи над програмною моделлю виконано аналіз існуючих апаратних та програмних засобів. В повній мірі описані всі компоненти розробленого програмного забезпечення.

Розроблено демонстраційний веб-застосунок для перевірки роботи системи. Наведені інструкції з підключення та налаштування програмних засобів.

Програма може використовуватися на ПЕОМ архітектури IBM PC з ОС Windows 10/11 та сумісних дистрибутивах Linux.

Програму розроблено в середовищі Visual Studio 2026 з використанням мови C# 14, платформи .NET 10 та СУБД PostgreSQL 18.

Ключові слова: кібербезпека, SQL-ін'єкція, веб-застосунок, машинне навчання, ASP.NET Core, WAF.

ABSTRACT

Shteinikov M.S. Cybersecurity system software for detecting and preventing SQL injections in web applications. 125 Cybersecurity. Central Ukrainian National Technical University. Kropyvnytskyi. 2026.

In this bachelor's qualification, software has been developed that is intended for detecting and preventing SQL injection attacks in web applications on the ASP.NET Core platform.

The purpose of the development is to develop cybersecurity system software for detecting and preventing SQL injections in web applications.

The result of the work is a software implementation of a cybersecurity system in the form of a middleware component for ASP.NET Core 10 with a hybrid detection mechanism based on signature analysis, machine learning models, and composite scoring.

In the process of working on the software model, an analysis of existing hardware and software was performed. All components of the developed software are fully described.

A demonstration web application has been developed to verify the operation of the system. Instructions for connecting and configuring the software are given.

The program can be used on an IBM PC with Windows 10/11 and compatible Linux distributions.

The program is developed in the Visual Studio 2026 environment using C# 14, the .NET 10 platform, and PostgreSQL 18 DBMS.

Keywords: cybersecurity, SQL injection, web application, machine learning, ASP.NET Core, WAF.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ І

ТЕРМІНІВ	2
ВСТУП.....	5
1. ПРИЗНАЧЕННЯ ТА ОБЛАСТЬ ВИКОРИСТАННЯ..	6
1.1 Призначення системи.....	6
1.2 Область застосування.	8
2. ПЕРЕГЛЯД АНАЛОГІЧНИХ ІСНУЮЧИХ СИСТЕМ	11
2.1 Огляд існуючих систем, технологій, архітектур та програмних рішень за профілем теми випускної кваліфікаційної роботи	11
2.2 Обґрунтування вибору засобів для побудови системи та мови програмування.	19
2.3 Розгорнута постановка завдання.	25
3 ОПИС І ОБҐРУНТУВАННЯ ПРОЕКТНИХ РІШЕНЬ	28
3.1 Опис функціонування системи	28
3.2 Розробка структурної схеми.....	36
3.3 Розробка функціональної схеми	43
3.4 Розробка діаграми процесів	48
4. РЕАЛІЗАЦІЯ ПРОЕКТУ. РОЗРАХУНКИ І ЕКСПЕРИМЕНТАЛЬНІ ДАНІ, ЩО ПІДТВЕРДЖУЮТЬ ПРАВИЛЬНІСТЬ ПРОЕКТНИХ РІШЕНЬ.....	54
4.1 Розробка блок–схем та опис алгоритмів функціонування системи... ..	54
4.2 Захист розробленого програмного забезпечення	71
5. МЕТОДИКА ВПРОВАДЖЕННЯ СИСТЕМИ В ПРОМИСЛОВУ ЕКСПЛУАТАЦІЮ	74
6. ОСНОВНІ ВИСНОВКИ	80
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	82

					<i>ВКРБ-125.26.0019.00.00.ПЗ</i>			
Зм.	Арк.	№ докум.	Підпис	Дата	Програмне забезпечення системи кібербезпеки для виявлення та запобігання SQL-ін'єкціям у веб-застосунках	Літера	Аркуш	Аркушів
Розроб.		Штейніков М.С.					1	86
Перевірів		Улічев О.С.				ЦНТУ гр. КБ-22-1		
Н. контр.		Коваленко А.С.						
Затв.		Смірнов О.А.						

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ І ТЕРМІНІВ

API – Application Programming Interface – програмний інтерфейс застосунку;

ASP.NET – Active Server Pages .NET – серверна платформа для розробки веб-застосунків від Microsoft;

BLOB – Binary Large Object – двійковий великий об'єкт;

CI/CD – Continuous Integration / Continuous Delivery – безперервна інтеграція та розгортання;

CNN – Convolutional Neural Network – згорткова нейронна мережа;

CRS – Core Rule Set – базовий набір правил (OWASP);

CRUD – Create, Read, Update, Delete – створення, читання, оновлення, видалення;

CSV – Comma-Separated Values – формат файлу з розділеними комами значеннями;

CWE – Common Weakness Enumeration – каталог типових слабкостей;

DBMS – Database Management System – система управління базами даних;

DI – Dependency Injection – впровадження залежностей;

DML – Data Manipulation Language – мова маніпулювання даними;

DDL – Data Definition Language – мова визначення даних;

DLL – Dynamic Link Library – динамічно-приєднувана бібліотека;

F1 – F1-score – гармонічне середнє точності та повноти;

FN – False Negative – хибно негативний результат;

FP – False Positive – хибно позитивний результат;

HTTP – HyperText Transfer Protocol – протокол передачі гіпертексту;

HTTPS – HTTP Secure – захищений HTTP;

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		2

IDS – Intrusion Detection System – система виявлення вторгнень;
 IPS – Intrusion Prevention System – система запобігання вторгненням;
 JSON – JavaScript Object Notation – формат обміну даними;
 JIT – Just-In-Time – компіляція «точно в строк»;
 LR – Logistic Regression – логістична регресія;
 ML – Machine Learning – машинне навчання;
 MVC – Model-View-Controller – модель-вид-контролер;
 NLP – Natural Language Processing – обробка природної мови;
 ONNX – Open Neural Network Exchange – відкритий формат нейронних мереж;
 ORM – Object-Relational Mapping – об'єктно-реляційне відображення;
 OWASP – Open Web Application Security Project – проект безпеки веб-застосунків;
 PII – Personally Identifiable Information – персональна ідентифікаційна інформація;
 RegEx – Regular Expression – регулярний вираз;
 REST – Representational State Transfer – архітектурний стиль веб-сервісів;
 ROC – Receiver Operating Characteristic – робоча характеристика приймача;
 SDK – Software Development Kit – комплект засобів розробки;
 SDLC – Software Development Life Cycle – життєвий цикл розробки ПЗ;
 SIEM – Security Information and Event Management – управління інформацією та подіями безпеки;
 SQL – Structured Query Language – структурована мова запитів;
 SQLi – SQL Injection – ін'єкція SQL;
 TF-IDF – Term Frequency – Inverse Document Frequency;
 TN – True Negative – істинно негативний результат;
 TP – True Positive – істинно позитивний результат;

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		3

URI – Uniform Resource Identifier – універсальний ідентифікатор ресурсу;

URL – Uniform Resource Locator – універсальний локатор ресурсу;

UTF – Unicode Transformation Format;

UUID – Universally Unique Identifier – універсально-унікальний ідентифікатор;

WAF – Web Application Firewall – брандмауер веб-застосунку;

XSS – Cross-Site Scripting – міжсайтовий скриптинг;

ВКР – Випускна кваліфікаційна робота;

ОС – Операційна система;

ПЗ – Програмне забезпечення;

СУБД – Система управління базами даних.

К6ПЗ – 2026

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		4

ВСТУП

Актуальність теми. SQL-ін'єкційні атаки (CWE-89) залишаються в переліку OWASP Top Ten упродовж останнього десятиліття. Класичні засоби захисту (параметризовані запити, ORM) не покривають усіх векторів атак і часто не застосовуються в успадкованих системах, а існуючі WAF-рішення є або сигнатурними і уразливими до обфускації, або ML-орієнтованими з високим відсотком хибних спрацювань.

Об'єкт дослідження – процеси забезпечення безпеки веб-застосунків від атак типу впровадження SQL-коду.

Предмет дослідження – методи та програмні засоби виявлення SQL-ін'єкцій у запитах користувача на основі гібридного підходу.

Мета роботи – розробка програмного забезпечення системи кібербезпеки для виявлення та запобігання SQL-ін'єкційним атакам у веб-застосунках на основі гібридного підходу, що поєднує сигнатурний аналіз і машинне навчання та забезпечує пояснюваність прийнятих рішень.

Практичне значення одержаних результатів. Розроблене програмне забезпечення реалізоване як готовий до використання middleware-компонент для платформи ASP.NET Core та може бути інтегроване у новостворювані й успадковані веб-застосунки. За ступенем готовності розробка відповідає рівню робочого прототипу та придатна до впровадження в організаціях, що експлуатують веб-застосунки на платформі .NET.

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		5

1 ПРИЗНАЧЕННЯ ТА ОБЛАСТЬ ВИКОРИСТАННЯ

1.1 Призначення системи

Розроблене програмне забезпечення являє собою систему кібербезпеки спеціалізованого призначення, орієнтовану на захист веб-застосунків від атак типу SQL-ін'єкція. Система реалізована у формі проміжного програмного забезпечення (middleware) для платформи ASP.NET Core, що функціонує як вбудований брандмауер веб-застосунку (Web Application Firewall, далі WAF) на рівні обробки HTTP-запитів конкретного веб-застосунку. Архітектурно система розташовується між мережевим стеком ASP.NET Core та бізнес-логікою захищеного застосунку, що дозволяє аналізувати усі вхідні запити, прийняті основним конвеєром, до того, як вони досягнуть контролерів або обробників ендпоінтів застосунку.

Основне призначення системи полягає у попередженні успішної експлуатації SQL-ін'єкційних вразливостей у веб-застосунках шляхом виявлення підозрілих елементів у параметрах HTTP-запитів та своєчасного блокування таких запитів до моменту, коли потенційно небезпечні дані будуть передані до бази даних. Система виконує наступні цільові функції:

- виявлення SQL-ін'єкційних шаблонів у параметрах рядка запиту (query string), тілі запиту (request body), HTTP-заголовках та значеннях файлів cookie;
- нормалізація вхідних даних для нейтралізації типових технік обфускації, що використовуються зловмисниками для обходу засобів захисту;
- розрахунок композитного ризик-скорю на основі результатів сигнатурного аналізу та оцінки машинною моделлю;
- прийняття рішення про подальшу долю запиту (пропустити, заблокувати, передати у режимі тіньового моніторингу) на основі налаштованого порогу;

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		6

- детальне журналювання усіх подій детекції з фіксацією обставин рішення для подальшого розслідування адміністратором безпеки;

- забезпечення мінімального впливу на продуктивність захищеного застосунку за рахунок ефективної архітектури та паралельної обробки.

Система розроблена з урахуванням специфіки сучасних веб-застосунків. Зокрема, вона підтримує обробку запитів з форматом тіла application/json, application/x-www-form-urlencoded, multipart/form-data, що покриває переважну більшість сценаріїв роботи веб-API та класичних веб-форм. Окремо забезпечена підтримка моделі GraphQL шляхом аналізу вмісту JSON-параметрів запитів. Для застосунків, що використовують GraphQL або інші протоколи поверх HTTP, система функціонує прозоро без необхідності додаткового налаштування.

Важливою особливістю призначення системи є те, що вона не замінює, а доповнює інші засоби забезпечення безпеки на рівні застосунку. Зокрема, рекомендований сценарій застосування передбачає одночасне використання параметризованих запитів та об'єктно-реляційного відображення (ORM) на рівні коду застосунку, валідації вхідних даних на рівні моделей, належного управління правами доступу до бази даних. Розроблена система виступає додатковим, але не єдиним рубежем захисту згідно з принципом «глибинного захисту» (defense in depth).

Поряд з функцією активного захисту, система виконує функцію інформування адміністратора безпеки про спроби атак. Усі заблоковані та підозрілі запити фіксуються у базі даних PostgreSQL з повним контекстом, що дозволяє інтегрувати журнал детекції з системами управління інформацією про події безпеки (SIEM-системами) для проведення кореляційного аналізу та виявлення складних багатоетапних атак.

У контексті відповідних компетентностей фахівця з кібербезпеки, розроблена система реалізує практичну реалізацію наступних задач: забезпечення захисту інформації, що обробляється в інформаційно-

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		7

телекомунікаційних системах; виявлення, ідентифікація, аналіз та реагування на інциденти кібербезпеки; впровадження процесів виявлення атак та запобігання їм; підтримка функціонування програмних комплексів виявлення вторгнень.

1.2 Область застосування

Областю застосування розробленої системи є захист веб-застосунків та веб-API, розроблених на платформі ASP.NET Core версії 6 та вище, включно з поточним основним релізом ASP.NET Core 10 (.NET 10 LTS). Сама платформа .NET є кросплатформною, що дозволяє розгорнути захищені нею застосунки у середовищах операційних систем Microsoft Windows, дистрибутивів Linux (Ubuntu, Debian, RHEL, CentOS, Alpine та інших), Apple macOS, а також у контейнеризованих середовищах на основі Docker та оркестрованих кластерах Kubernetes.

Цільові сценарії розгортання системи можна класифікувати за наступними критеріями.

За масштабом захищеного застосунку:

- невеликі корпоративні застосунки внутрішнього призначення з кількістю користувачів до кількох сотень;
- середні застосунки інтернет-сервісів з навантаженням до тисяч запитів на секунду на один екземпляр;
- великі публічні веб-сервіси з горизонтальним масштабуванням за допомогою контейнерної оркестрації.

За критичністю захищених даних:

- фінансові застосунки (онлайн-банкінг, платіжні шлюзи, фінтех-сервіси);
- застосунки електронного урядування та надання державних послуг;

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		8

- медичні інформаційні системи та електронні системи охорони здоров'я;

- корпоративні портали з конфіденційною бізнес-інформацією;

- електронна комерція та системи онлайн-торгівлі;

- освітні платформи та системи управління навчанням.

За архітектурою захищеного застосунку:

- класичні монолітні веб-застосунки на ASP.NET Core MVC та Razor Pages;

- REST API сервіси, у тому числі побудовані за стандартом OpenAPI/Swagger;

- GraphQL-сервіси, реалізовані за допомогою бібліотек HotChocolate або GraphQL.NET;

- Blazor Server-застосунки;

- мікросервісні архітектури, у яких розроблена система розгортається перед кожним сервісом, що приймає зовнішній трафік;

- бек-енд-частини мобільних застосунків.

Особливо актуальним є застосування розробленої системи для захисту успадкованих (legacy) веб-застосунків, у яких неможливо швидко провести масштабний рефакторинг для усунення прямого формування SQL-запитів конкатенацією рядків. У таких випадках розгортання WAF-middleware дозволяє суттєво знизити ризики експлуатації наявних вразливостей без необхідності повного переписування коду.

Розроблена система також може використовуватися в освітніх цілях – як приклад практичної реалізації принципів захисту веб-застосунків для студентів спеціальності 125 «Кібербезпека» та інших суміжних спеціальностей. Завдяки відкритості архітектури та документованості коду, студенти можуть досліджувати на її прикладі такі питання як: проектування middleware-компонентів у ASP.NET Core; розробку та використання регулярних виразів для розпізнавання мов запитів; інтеграцію моделей

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		9

машинного навчання в .NET-застосунки через ONNX Runtime; принципи побудови систем виявлення вторгнень.

Окремою сферою застосування є дослідницькі задачі. Розроблена система може використовуватися фахівцями з кібербезпеки для збору датасетів реальних атак (через тривале спостереження за журналом подій), оцінки ефективності різних стратегій захисту, тестування нових моделей машинного навчання у задачах виявлення SQL-ін'єкцій (модель легко замінюється шляхом завантаження іншого ONNX-файлу без перекомпіляції основного коду системи).

Існує також ряд обмежень щодо області застосування. Зокрема, система не призначена для захисту від атак, що не пов'язані з рівнем застосунку – наприклад, DDoS-атак мережевого рівня, атак на DNS-інфраструктуру, фізичних атак тощо. Для протидії таким загрозам мають використовуватися інші засоби (мережеві брандмауери, системи захисту від DDoS, фізичні засоби безпеки). Окрім того, розроблена система спеціалізована саме на SQL-ін'єкціях; інші види ін'єкцій (NoSQL, XPath, LDAP, командний рядок) виходять за межі поточної реалізації, хоча архітектура передбачає розширення на ці випадки шляхом доповнення сигнатурної бази та перенавчання ML-моделі на відповідних датасетах.

Таким чином, область застосування розробленої системи охоплює широкий спектр сучасних веб-застосунків, для яких актуальним є посилення захисту від SQL-ін'єкційних атак, з пріоритетом на сферах з підвищеними вимогами до конфіденційності та цілісності даних.

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		10

2 ПЕРЕГЛЯД АНАЛОГІЧНИХ ІСНУЮЧИХ СИСТЕМ

2.1 Огляд існуючих систем, технологій, архітектур та програмних рішень за профілем теми випускної кваліфікаційної роботи

Перед початком проектування власної системи необхідно провести аналіз існуючих рішень у досліджуваній предметній області. Це дозволяє: по-перше, врахувати накопичений у галузі досвід та уникнути повторення помилок попередників; по-друге, чітко визначити нішу для пропонованого рішення з точки зору його відмінностей від наявних аналогів; по-третє, обґрунтувати доцільність розробки шляхом виявлення непокритих або недостатньо вирішених наявними системами питань.

Усю сукупність існуючих систем захисту веб-застосунків від SQL-ін'єкційних атак можна умовно поділити на чотири класи: брандмауери веб-застосунків загального призначення (ModSecurity, NAXSI, Cloudflare WAF, AWS WAF); спеціалізовані бібліотеки виявлення SQLi на рівні коду (libinjection, SQLMap-detection); системи виявлення вторгнень з підтримкою аналізу HTTP-трафіку (Snort, Suricata з відповідними правилами); академічні рішення на основі машинного навчання, описані у науковій літературі. Розглянемо детальніше представників кожного з класів.

ModSecurity з набором правил OWASP CRS

ModSecurity – це найвідоміше відкрите рішення WAF, яке розгортається у вигляді модуля до веб-серверів Apache HTTP Server, Nginx та IIS. Початково проект був розроблений Іваном Ристичем у 2002 році, наразі активна розробка ведеться у форматі ModSecurity v3 (бібліотека libmodsecurity), яка є переписаним C++ ядром, що відокремило логіку від конкретного веб-сервера. ModSecurity сам по собі не містить правил детекції – він являє собою рушій виконання правил, написаних мовою SecLang. Найпоширенішим набором

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		11

правил для нього є OWASP Core Rule Set (CRS) – відкрита база правил, що покриває основні класи веб-атак (SQLi, XSS, LFI, RFI, серверні template injection, command injection тощо).

Архітектурно ModSecurity з CRS реалізує суто сигнатурний підхід. Кожне правило є регулярним виразом або складовою умовою, що перевіряє вміст HTTP-запиту на наявність шкідливих фрагментів. Для зменшення кількості хибно-позитивних спрацювань CRS використовує модель оцінювання аномалій (anomaly scoring), у якій кожне правило підвищує деякий лічильник на свою вагу, а блокування здійснюється при досягненні налаштованого порога. Це частково розв’язує проблему точкових спрацювань, але у цілому не виходить за рамки сигнатурної парадигми.

Сильні сторони ModSecurity з CRS: висока швидкодія (сотні тисяч запитів на секунду на одне ядро); широка база перевірених у промисловому використанні правил; розгалужена документація та активна спільнота; відкритий код під ліцензією Apache 2.0; підтримка усіх популярних веб-серверів.

Слабкі сторони ModSecurity з CRS: складність налаштування та підтримки великої кількості правил; традиційно високий рівень хибно-позитивних спрацювань (за різними оцінками – до 1-3% на типовому продуктивному трафіку), що вимагає тривалого тюнінгу для конкретного застосунку; принципова вразливість до невідомих варіантів обфускації, які не передбачені авторами правил; обмежена пояснюваність – для розуміння причин блокування адміністратор має самотійно зіставити спрацювали правила з контентом запиту; необхідність розгортання разом з веб-сервером (Apache/Nginx/IIS), що не завжди відповідає сучасним архітектурам мікросервісів та хмарних застосунків.

У контексті розроблюваної у даній роботі системи, важливо відзначити, що ModSecurity не використовує машинне навчання у класичному сенсі, не виконує системної багатоетапної нормалізації payload’у (хоча має набір

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		12

transformation-функцій), та не пропонує вбудованої інтеграції з .NET-застосунками без додаткового веб-сервера-проксі.

NAXSI

NAXSI (Nginx Anti XSS & SQL Injection) – це WAF-модуль для веб-сервера Nginx, розроблений компанією NBS System (Франція) у 2011 році. На відміну від ModSecurity, NAXSI використовує підхід «білого списку»: система не містить заздалегідь визначеного списку шкідливих шаблонів, а навпаки – описує правила того, які символи та послідовності допустимі у різних частинах HTTP-запиту. Будь-яке відхилення від білого списку фіксується та оцінюється.

Сильні сторони NAXSI: простіший за ModSecurity синтаксис правил; принциповий підхід «за замовчуванням заборонено» зменшує ймовірність пропуску невідомих атак; помірне споживання ресурсів. Слабкі сторони: вимагає тривалого етапу «навчання» (LearningMode) для збору legitimate-патернів конкретного застосунку; вузька прив'язка до Nginx; обмежена робота з нестандартними форматами тіла запитів; вкрай мала підтримка нових технік обфускації.

Хмарні WAF-сервіси (Cloudflare, AWS, Azure)

У сегменті комерційних рішень переважають хмарні сервіси WAF – Cloudflare WAF, AWS WAF, Azure Web Application Firewall, Google Cloud Armor, Imperva, F5 BIG-IP ASM. Ці рішення розгортаються у моделі SaaS: трафік до захищеного застосунку направляється через інфраструктуру провайдера, де піддається аналізу. Перевагами такої моделі є: відсутність необхідності встановлювати та підтримувати ПЗ на власних серверах; масштабованість; інтеграція із засобами захисту від DDoS-атак; регулярні оновлення сигнатурних баз.

Технологічно хмарні WAF використовують гібридний підхід: сигнатурна детекція класичних атак доповнюється системами поведінкового аналізу та машинного навчання, які функціонують на агрегованих даних від

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		13

мільйонів клієнтів провайдера. Це дає таким сервісам перевагу у виявленні нових атак, але одночасно створює відомі обмеження: висока вартість підписки (від кількох сотень доларів на місяць для базових тарифів до десятків тисяч для корпоративних); неможливість роботи у режимі ізольованої внутрішньої мережі (on-premise); залежність від інфраструктури зовнішнього провайдера, що неприйнятно для систем з вимогами щодо обробки персональних даних у межах юрисдикції; повна «чорна скринька» логіки детекції з боку клієнта – рішення приймаються без можливості детального аудиту.

Для контексту України, де питання локалізації обробки персональних даних, відповідності національному законодавству про захист інформації та функціонування у режимах часткової ізоляції від глобальної інфраструктури мають особливе значення, рішення на основі повністю локального розгортання WAF мають об'єктивні переваги перед хмарними сервісами.

Спеціалізовані бібліотеки: libinjection

libinjection – це бібліотека з відкритим кодом, розроблена Ніком Гелбретом (Nick Galbreath) у 2012 році, що реалізує специфічний підхід до виявлення SQL-ін'єкцій – токенизацію та аналіз «відбитків» (fingerprints) рядка на предмет його структурної подібності до коректного SQL-виразу. Бібліотека написана на C, має зручні обгортки для багатьох мов, та є де-факто стандартом для багатьох WAF-рішень (зокрема, ModSecurity інтегрує libinjection як одну з опцій). Підхід libinjection відрізняється від класичних регулярних виразів: вона не шукає конкретні шаблони, а намагається лексично розібрати рядок та визначити, чи являє він собою SQL-вираз або його фрагмент.

Сильні сторони: висока швидкодія; відносно низький рівень хибно-позитивних спрацювань для природних текстів; стійкість до простих варіантів обфускації. Слабкі сторони: повне ігнорування семантики запиту (так звані blind SQLi через тимчасові затримки можуть мати валідний синтаксис); обмежена ефективність на нових мовах SQL-діалектів (NoSQL-варіанти,

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		14

наприклад); відсутність механізму навчання – для виявлення нових векторів атак потрібен апдейт самої бібліотеки.

Академічні рішення на основі машинного навчання

У науковій літературі останніх років опубліковано велику кількість досліджень, у яких пропонуються різноманітні підходи до виявлення SQL-ін'єкцій з використанням методів машинного навчання та глибокого навчання. Виокремимо найбільш репрезентативні напрямки.

Класифікація на основі статистичних ознак. У ранніх роботах (приблизно 2008-2015 років) автори зазвичай витягали з payload'у набір ручно сконструйованих ознак: довжина, кількість лапок, наявність ключових слів SQL, частоти різних класів символів, відношення кількості цифр до букв тощо. На цих ознаках навчалися класифікатори – наївний Баєс, дерева рішень, метод опорних векторів, випадкові ліси. Такий підхід має проблему – якість залежить від якості сконструйованих ознак (feature engineering), а зломисник може обходити модель, спеціально підлаштовуючи ознаки своєї атаки під легітимні.

Класифікація на основі мішка n-грамів (bag of n-grams). У роботах 2014-2018 років широкого розповсюдження набув підхід з представленням payload'у як мішка символічних або словесних n-грамів з подальшим перетворенням TF-IDF та класифікацією моделями типу логістичної регресії, лінійного SVM, градієнтного бустингу. Цей підхід дозволяє моделі автоматично виявити інформативні фрагменти без ручної інженерії ознак. На стандартних бенчмарках (CSIC 2010, HTTP DATASET CSIC) такі моделі показують точність на рівні 0.95-0.99 за метрикою F1-score. Однак на штучно обфускованих наборах метрики падають.

Глибокі нейронні мережі. Починаючи з 2017-2018 років, активно розвивається напрямок з використанням глибоких архітектур: рекурентні нейронні мережі (RNN, LSTM, GRU), згорткові мережі на символічному рівні (character-level CNN), трансформери. У низці робіт показано, що такі моделі здатні виявляти SQL-ін'єкції з обфускацією краще, ніж класичні моделі на n-

грамах. Платою за це є більший розмір моделей, більший час інференсу та значно більші вимоги до обчислювальних ресурсів на етапі навчання.

Велкі мовні моделі (LLM). Найновіші роботи (2023-2025) пропонують використовувати fine-tuning попередньо навчених трансформерних моделей (BERT, RoBERTa, спеціалізовані code-моделі) для задачі класифікації SQL-ін'єкцій. Такі моделі демонструють найвищі метрики на дослідницьких наборах, проте їх практичне розгортання у real-time WAF ускладнюється значним розміром (від сотень мегабайт) та повільним інференсом (десятки мілісекунд на запит), що неприйнятно для високонавантажених систем.

Загальним недоліком академічних рішень є те, що вони, як правило, є дослідницькими прототипами, не доведеними до рівня промислової інтеграції. У публікаціях рідко розглядаються питання інтеграції моделі у конкретний веб-фреймворк, продуктивності у режимі реального часу, журналювання, обробки нових форматів запитів. Тому пряме перенесення академічних результатів у промислову експлуатацію вимагає значного інженерного зусилля.

Порівняльна характеристика існуючих рішень

Усю розглянуту різноманітність систем захисту від SQL-ін'єкцій можна структурувати у вигляді ієрархічної класифікації за основою архітектурного підходу та принципом детекції. Така класифікація представлена на рисунку 2.1; вона також показує, яку нішу займає розроблювана у даній випускній кваліфікаційній роботі система. На верхньому рівні виділено чотири основні категорії: брандмауери веб-застосунків загального призначення, спеціалізовані бібліотеки виявлення SQLi на рівні коду, системи виявлення вторгнень з аналізом HTTP-трафіку та академічні рішення на основі машинного навчання. Кожна категорія представлена характерними інструментами або підходами, що дозволяє наочно зіставити їх із позиціонуванням AntiSQLi-WAF як спеціалізованого WAF для .NET-екосистеми з гібридною детекцією.

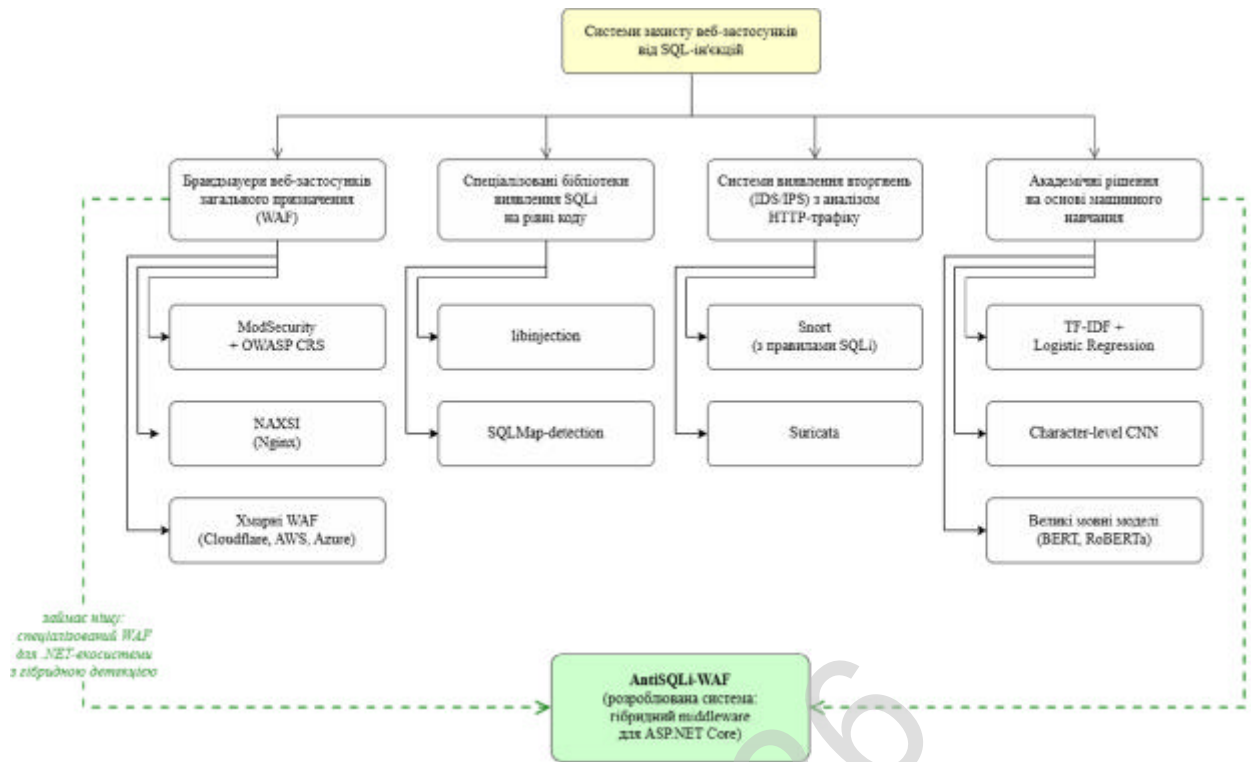


Рисунок 2.1 – Класифікація існуючих систем захисту веб-застосунків від SQL-ін'єкцій та позиціонування розроблюваної системи

Для систематизації результатів проведеного огляду наведемо у вигляді таблиці порівняння ключових характеристик розглянутих систем за низкою критеріїв, що мають практичне значення.

Таблиця 2.1 – Порівняльна характеристика існуючих рішень захисту від SQL-ін'єкцій

Критерій	Mod-Security	NAXSI	Cloud-flare WAF	libinjection	Розроблювана система
Сигнатурний аналіз	Так	Частково	Так	Так	Так
Машинне навчання	Ні	Ні	Так	Ні	Так

Продовження таблиці 2.1

Критерій	Mod-Security	NAXSI	Cloud-flare WAF	libinjection	Розроблювана система
Багатоетапна нормалізація payload	Частково	Ні	Так	Ні	Так
Пояснюваність рішень	Низька	Середня	Низька	Низька	Висока
Локальне розгортання	Так	Так	Ні	Так	Так
Інтеграція з .NET без проксі	Ні	Ні	Ні	Частково	Так
Вартість використання	Безкошт.	Безкошт.	Висока	Безкошт.	Безкошт.
Стійкість до обфускації	Низька	Середня	Висока	Середня	Висока
Налаштовуваний порог скорингу	Частково	Так	Так	Ні	Так

З аналізу таблиці видно, що жодне з розглянутих готових рішень не покриває одночасно усіх практично важливих критеріїв. Зокрема, відкриті рішення (ModSecurity, NAXSI, libinjection) суттєво поступаються у стійкості до обфускації та не використовують машинне навчання. Хмарні комерційні WAF (Cloudflare, AWS) технологічно більш досконалі, але непридатні для локального розгортання та мають високу вартість. Власне рішення, що пропонується у даній випускній кваліфікаційній роботі, заповнює виявлену

нішу: відкритий гібридний WAF з машинним навчанням та пояснюваним скорингом, який нативно інтегрується у .NET-застосунки.

Окрім переліченого, варто зазначити такий аспект: реальні промислові WAF (особливо комерційні) є системами загального призначення для захисту від багатьох класів атак. Розроблювана у даній роботі система спеціалізована саме на SQLi, що дозволяє приділити значно більше уваги особливостям саме цього класу атак – зокрема, побудувати багатоетапну нормалізацію, специфічну для SQL-діалектів, та навчити ML-модель саме на специфічних SQLi-датасетах.

2.2 Обґрунтування вибору засобів для побудови системи та мови програмування

Після проведеного аналізу існуючих рішень та виявлення цільової ніші для розроблюваної системи постає задача обґрунтованого вибору технологічного стеку реалізації. Цей вибір має враховувати декілька взаємопов'язаних аспектів: продуктивність системи, доступність необхідних бібліотек та інструментів, узгодженість з типовим стеком веб-застосунків, які будуть захищатися, перспективи розвитку та підтримки технологій, а також відповідність вимогам предметної області.

Обґрунтування вибору мови програмування

Серед сучасних мов програмування, що використовуються для розробки серверних компонентів та засобів кібербезпеки, провідні позиції займають C/C++, Java, C#, Python, Go та Rust. Кожна має свої сильні сторони: C/C++ – максимальна продуктивність та контроль над пам'яттю; Java – розвинена корпоративна екосистема; Python – лідерство у машинному навчанні; Go та Rust – сучасний підхід до конкурентності. Проте у контексті задачі захисту .NET-веб-застосунків саме C# / .NET має ключові переваги.

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		19

Обрана мова реалізації – C# 14 з використанням .NET 10 LTS. Платформа .NET 10 є поточною версією з тривалою підтримкою (до листопада 2028 року), що відповідає вимогам промислової експлуатації. Перевагами C# у контексті розроблюваної системи є: нативна підтримка ASP.NET Core, у якому розгортається переважна більшість сучасних .NET веб-застосунків; зріла модель middleware; вбудована підтримка асинхронного програмування на основі моделі async/await; офіційно підтримуваний пакет Microsoft.ML.OnnxRuntime для виконання ONNX-моделей; розвинена екосистема пакетів NuGet; статичний аналіз та засоби безпечного кодування.

Для навчання моделей машинного навчання використовується Python 3.14 з бібліотеками TensorFlow 2.21 / Keras 3 (для CNN-моделі) та scikit-learn 1.6 (для TF-IDF + LogReg). Експорт навчених моделей у формат ONNX виконується через tf2onnx та skl2onnx. Такий розподіл відповідає поширеній індустріальній практиці: ML-команда працює у звичному середовищі Python, а інженерна команда інтегрує готову модель у production-середовище через стандартизований формат ONNX.

Обґрунтування вибору платформи ASP.NET Core

Платформа ASP.NET Core є офіційним веб-фреймворком .NET від компанії Microsoft. Версія ASP.NET Core 10 забезпечує одні з найвищих показників продуктивності серед усіх веб-фреймворків. За даними щорічного бенчмарку TechEmpower, ASP.NET Core входить до десятки найшвидших веб-фреймворків у світі.

Ключовою архітектурною концепцією ASP.NET Core, яку безпосередньо використовує розроблювана система, є конвеєр обробки запитів (request processing pipeline), побудований на патерні middleware. Middleware – це компонент, який отримує HTTP-контекст, виконує деяку логіку та передає управління наступному компоненту в конвеєрі або повертає відповідь без подальшої передачі. Така архітектура ідеально підходить для

					ВКРБ-125.26.0019.00.00.ПЗ	Арх
Зм	Лист	№ докум.	Підпис	Дата		20

реалізації WAF. Місце розроблюваного WAF у конвеєрі обробки запитів ASP.NET Core схематично показано на рисунку 2.2.

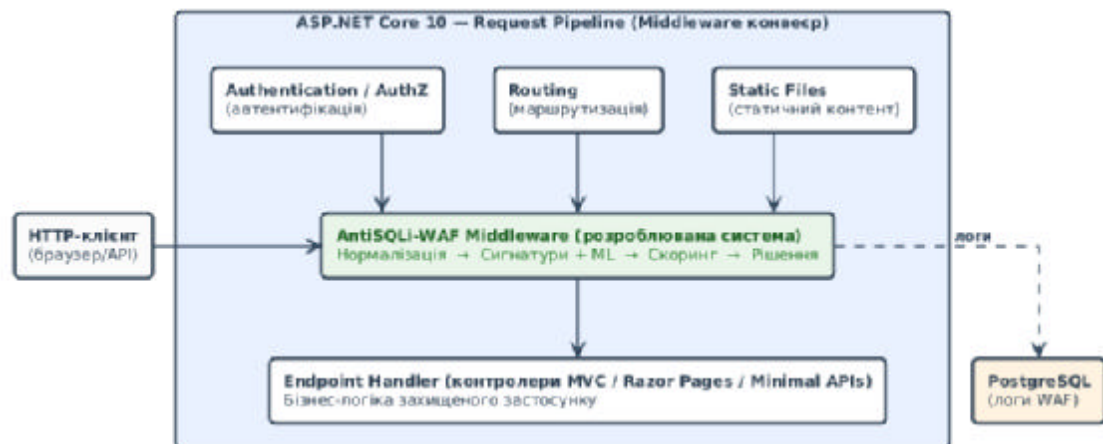


Рисунок 2.2 – Позиція розроблюваного middleware у конвеєрі обробки запитів ASP.NET Core 10

Інші переваги ASP.NET Core: вбудована підтримка впровадження залежностей (DI), що дозволяє чисто реалізувати декомпозицію компонентів; вбудована система логування; підтримка конфігурації через файли appsettings.json та змінні оточення; багатий вибір засобів інтеграції з базами даних (Entity Framework Core, Dapper тощо).

Обґрунтування вибору ONNX Runtime для виконання ML-моделей

ONNX (Open Neural Network Exchange) – це відкритий формат представлення моделей машинного навчання, розроблений альянсом провідних технологічних компаній. ONNX Runtime – високопродуктивний рушій виконання моделей у форматі ONNX, доступний у вигляді бібліотек для C++, C#, Java, Python та інших мов.

Переваги ONNX Runtime у контексті поставленої задачі: гнучкість у виборі ML-фреймворку для навчання (модель може бути навчена у TensorFlow, PyTorch, scikit-learn та потім завантажена у .NET-застосунок); висока продуктивність інференсу за рахунок graph fusion та апаратного прискорення; простота розгортання через єдиний NuGet-пакет

Microsoft.ML.OnnxRuntime; стійкість API завдяки відкритому стандарту. Обраний ONNX Runtime версії 1.26 – поточна стабільна гілка на момент розробки.

Обґрунтування вибору моделей машинного навчання

Як обговорено у пункті 2.1, у літературі описано широкий спектр моделей машинного навчання для виявлення SQL-ін'єкцій. У даній роботі обрано порівняти дві моделі різних класів складності: TF-IDF + Logistic Regression (класична схема з ваговим коефіцієнтом TF-IDF на символічних n-грамах та логістичною регресією) та символічну Character-level CNN з архітектурою, інспірованою роботою Yoan Kim «Convolutional Neural Networks for Sentence Classification» (2014). Очікувано, що CNN покаже кращі результати на обфускованих payload'ах, тоді як LR забезпечить більш швидко та просту інтеграцію. Експериментальне порівняння обох моделей виконано у розділі 4 з остаточним обґрунтуванням вибору.

Обґрунтування вибору PostgreSQL для журналювання

Для зберігання журналу подій виявлення обрано систему управління базами даних PostgreSQL 18 (поточна стабільна гілка станом на момент розробки, з мінорним релізом 18.4 від травня 2026 року). Вибір обумовлений наступними чинниками:

- PostgreSQL є однією з найпопулярніших СУБД з відкритим кодом, з широким розповсюдженням у промисловому використанні;
- повна підтримка стандарту SQL та розширень для роботи з JSON, що зручно для зберігання структурованих метаданих про події;
- висока продуктивність та надійність, перевірена тривалим використанням у системах будь-якого масштабу;
- розвинена екосистема .NET-драйверів (Npgsql) та ORM-засобів (Entity Framework Core з провайдером Npgsql.EntityFrameworkCore.PostgreSQL);
- повна крос-платформність та доступність у будь-яких сучасних дистрибутивах ОС, контейнерах Docker та керованих хмарних сервісах;

– зрозуміла модель захисту самої СУБД (вбудовані ролі, ACL, TLS-з'єднання), що відповідає вимогам захисту чутливої інформації про події безпеки.

Для взаємодії з PostgreSQL з C# обрано бібліотеку Npgsql 10 – нативний драйвер з підтримкою найновіших функцій PostgreSQL 18, у тому числі віртуальних обчислюваних стовпців та UUIDv7. Для абстрагування від ручної роботи з SQL-запитами на рівні WAF-логування використовується Dapper – легковагова мікро-ORM, що балансує між простотою та продуктивністю.

Обґрунтування вибору засобів логуювання та конфігурування

Для системного логуювання (рівень діагностичних повідомлень, помилок, продуктивності) використано бібліотеку Serilog – фактичний стандарт для структурованого логуювання у .NET-екосистемі. Serilog підтримує множинні «приймачі» (sinks) для пересилання логів у файли, консоль, бази даних, ELK-стек, Seq, Splunk тощо.

Для конфігурування системи використано вбудовану у ASP.NET Core інфраструктуру конфігурування на основі шаблону Options. Параметри (порог скорингу, ваги джерел детекції, файл моделі, рядок підключення до PostgreSQL тощо) зчитуються з файлу appsettings.json з можливістю перевизначення через змінні оточення, що зручно для контейнеризованого розгортання.

Зведений технологічний стек

У результаті проведеного аналізу та обґрунтування, технологічний стек розроблюваної системи виглядає наступним чином:

Таблиця 2.2 – Технологічний стек розроблюваної системи

Компонент	Технологія	Версія / зауваги
Мова програмування основного коду	C#	14 (C# 14)

Продовження таблиці 2.2

Компонент	Технологія	Версія / зауваги
Платформа виконання	.NET	10.0 LTS (до 11.2028)
Веб-фреймворк	ASP.NET Core	10.0
Інференс ML-моделей	Microsoft.ML.OnnxRuntime	1.26
База даних логування	PostgreSQL	18.4
Драйвер БД	Npgsql + Dapper	10.0 / 2.1
Структуроване логування	Serilog	4.0
Мова навчання ML-моделей	Python	3.14
ML-фреймворк (CNN)	TensorFlow / Keras	2.21 / 3.x
Експорт моделей в ONNX	tf2onnx, skl2onnx	1.16 / 1.16
Контейнеризація	Docker	27+
IDE	Visual Studio 2026 / Rider	18.4 / 2025.3

Обраний стек є сучасним, добре документованим, підтримуваним великими спільнотами та промисловими гравцями. Він повністю задовольняє вимоги до розроблюваної системи. Поєднання .NET 10 LTS з ONNX Runtime для інференсу та Python-екосистеми для навчання моделей дозволяє відокремити обчислювально витратний етап тренування від продуктивного середовища виконання, що відповідає сучасним практикам розгортання MLOps-рішень.

2.3 Розгорнута постановка завдання

На основі результатів аналізу предметної області, огляду існуючих систем та обґрунтування технологічного стеку, сформулюємо розгорнуту постановку задачі для подальшого проектування та реалізації.

Необхідно розробити програмне забезпечення системи кібербезпеки для виявлення та запобігання SQL-ін'єкційним атакам у веб-застосунках на платформі ASP.NET Core. Система має задовольняти наступним вимогам.

Функціональні вимоги

- Система повинна функціонувати як middleware-компонент ASP.NET Core, який вбудовується у конвеєр обробки запитів за допомогою одного методу розширення (наприклад, `app.UseAntiSqliWaf()`).

- Система повинна аналізувати усі компоненти HTTP-запиту, потенційно здатні нести SQL-ін'єкційний payload: параметри рядка запиту (query string); параметри маршруту (route values); HTTP-заголовки (за винятком службових); значення файлів cookie; тіло запиту у форматах application/json, application/x-www-form-urlencoded, multipart/form-data.

- Система повинна виконувати багатоетапну нормалізацію виявлених строкових значень перед детекцією: рекурсивне URL-декодування (до 3 ітерацій); декодування hex-послідовностей; декодування Unicode escape-послідовностей; видалення SQL-коментарів усіх типів (--, /*...*/, #, /*!...*/); згортка пробільних символів; перетворення регістру до нижнього.

- Сигнатурний аналізатор повинен оцінювати кожне значення проти бази правил у форматі регулярних виразів, розділених на дві категорії (High Severity та Low Severity) з власними вагами.

- ML-канал повинен використовувати модель у форматі ONNX, завантажену з файлу, шлях до якого налаштовується через конфігурацію.

- Композитний скоринг повинен обчислювати фінальний ризик-скор як зважену суму нормованих оцінок сигнатурного аналізу та ML-моделі, а також

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		25

враховувати додаткові контекстні фактори (тип цільового endpoint'у, попередня поведінка IP-адреси).

– Система повинна підтримувати три режими прийняття рішення: Detect Only (тільки логувати, не блокувати), Shadow (логувати з позначкою «було б заблоковано»), Block (блокувати з поверненням HTTP 403).

– Усі рішення та підстави повинні детально журналюватися у базу даних PostgreSQL з фіксацією: ідентифікатора події (UUID); часової мітки; IP-адреси клієнта; URI та методу запиту; нормалізованого вмісту, що викликав підозру; списку спрацьованих сигнатур; впевненості ML-моделі; підсумкового ризик-скорю; ужитого рішення.

– Система повинна забезпечувати можливість виключення певних URI або параметрів зі сканування за допомогою конфігураційних списків allowlist.

Нефункціональні вимоги

– Продуктивність: середній час обробки запиту middleware не повинен перевищувати 10 мс на стандартному обладнанні (CPU 2.5+ ГГц).

– Пропускна здатність: одне ядро повинно витримувати не менше 500 запитів на секунду з повним аналізом.

– Споживання пам'яті: до 200 МБ резидентної пам'яті при роботі з типовою CNN-моделлю.

– Відмовостійкість: відмова логування у PostgreSQL не повинна блокувати основну функцію детекції – логи буферизуються та повторно надсилаються.

– Кросплатформність: підтримка ОС Windows 10/11, основних дистрибутивів Linux (Ubuntu 22.04+, Debian 12+, RHEL/Rocky 9+), macOS 13+.

– Документованість: розгорнутий README, документація API-розширень, журнал змін.

Вимоги до інтерфейсу

Розроблювана система не має графічного інтерфейсу. Її конфігурування виконується через стандартний для ASP.NET Core механізм Options та секції

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		26

файлу appsettings.json. Перегляд журналу детекції здійснюється через стандартні засоби роботи з PostgreSQL (psql, DBeaver, pgAdmin) або через зовнішні системи аналізу логів (SIEM, ELK тощо). Підтримка веб-інтерфейсу адміністратора у даній версії системи не передбачається.

Вимоги до даних

- Конфігурація системи зберігається у файлі appsettings.json у форматі JSON.
- Сигнатурна база зберігається у вигляді JSON-файлу з масивом правил, кожне з яких містить id, regex, severity, weight, description.
- ML-модель зберігається у файлі формату ONNX.
- Журнал детекції зберігається у таблиці PostgreSQL detection_events з відповідною схемою.

Вимоги до тестування

- Розробка модульних тестів для кожного компонента системи (нормалізатор, сигнатурний аналізатор, скоринг);
- Інтеграційні тести middleware на тестовому ASP.NET Core застосунку.
- Експериментальне оцінювання ефективності на тестовій вибірці payload'ів з обчисленням метрик: accuracy, precision, recall, F1-score, ROC-AUC.
- Навантажувальне тестування для перевірки нефункціональних вимог щодо продуктивності.

Така постановка задачі забезпечує конкретні, перевіряємі критерії успіху для розроблюваної системи та задає рамки наступної проектної роботи у Розділі 3.

3 ОПИС І ОБҐРУНТУВАННЯ ПРОЕКТНИХ РІШЕНЬ

Згідно з результатами огляду літератури, аналізу аналогічних рішень та обґрунтування технологічного стеку, проведених у Розділі 2, у цьому розділі формулюються основні проектні рішення, що визначають архітектуру та принципи функціонування розроблюваної системи. Структура розділу відповідає рекомендованій структурі проектної частини випускної кваліфікаційної роботи: підрозділ 3.1 присвячено опису функціонування системи у цілому, у підрозділі 3.2 наведено структурну схему (повну) системи з детальним описом кожного компонента, у підрозділі 3.3 – функціональну схему з описом інформаційних потоків, у підрозділі 3.4 – діаграму процесів, що відбуваються в системі під час обробки HTTP-запиту.

3.1 Опис функціонування системи

Розроблювана система являє собою програмне забезпечення, яке функціонує як один з компонентів конвеєра обробки HTTP-запитів веб-застосунку на платформі ASP.NET Core 10. У цьому пункті описується принцип роботи системи на верхньому рівні абстракції – як послідовність етапів, через які проходить кожен вхідний HTTP-запит, та як відбувається прийняття рішення про допустимість запиту.

Загальна логіка обробки запиту

Обробка кожного вхідного HTTP-запиту розроблюваною системою складається з шести логічно послідовних етапів. На рисунку 3.1 наведено високорівневу схему цього процесу.

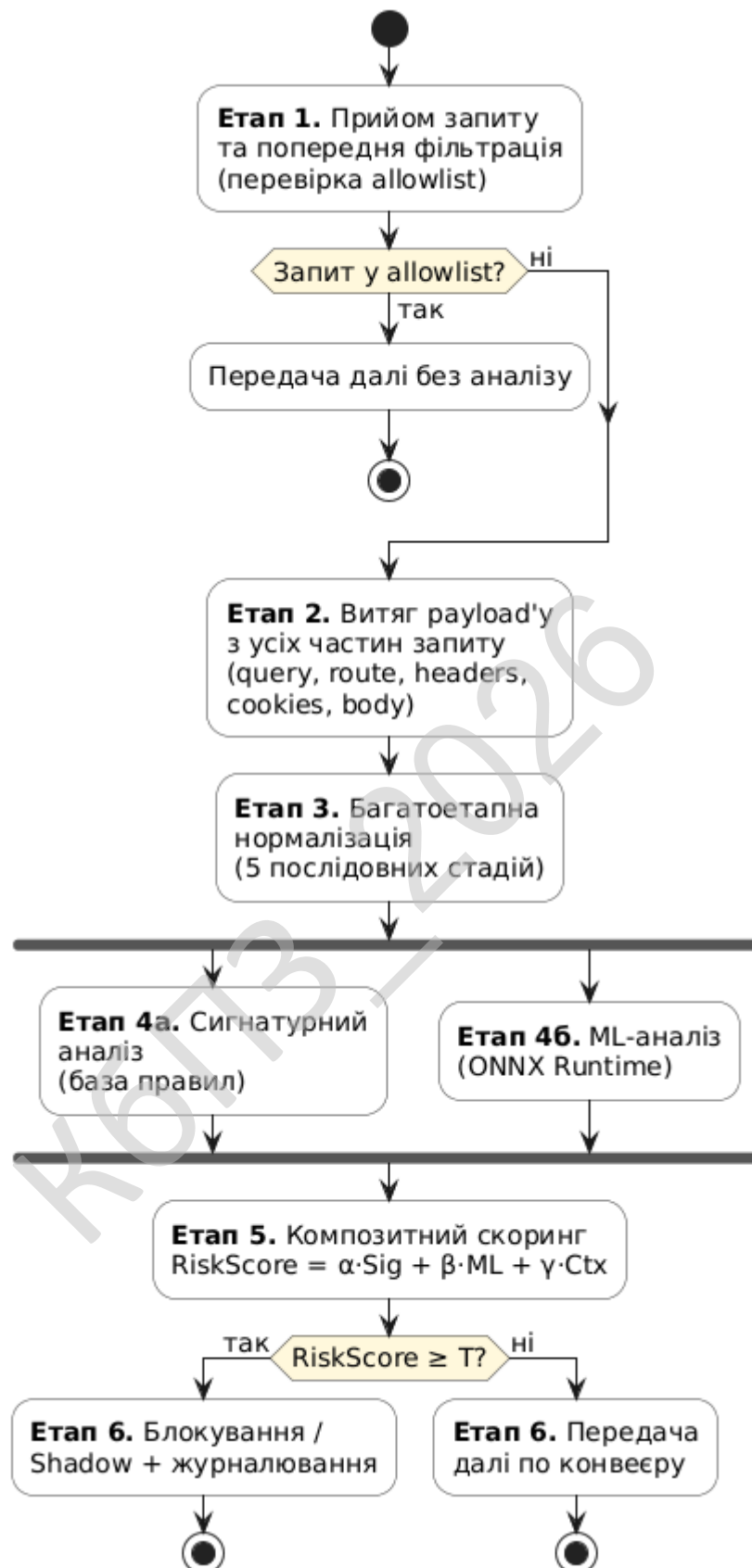


Рисунок 3.1 – Високорівнева схема обробки HTTP-запиту розроблюваною системою

Етап 1. Прийом запиту та попередня фільтрація. На цьому етапі middleware отримує HTTP-контекст від попередніх компонентів конвеєра ASP.NET Core. Виконується перевірка списку виключень (allowlist): якщо цільовий URI або метод запиту знаходиться у налаштовуваному списку безпечних маршрутів, аналіз не виконується, і запит одразу передається далі. Це дозволяє виключити зі сканування службові маршрути (наприклад, healthcheck-ендпоінти, ендпоінти метрик), для яких сканування економічно недоцільне.

Етап 2. Витяг payload'у з усіх частин запиту. Система формує список усіх строкових значень, які потенційно можуть нести SQL-ін'єкційний payload. Джерелами таких значень є: параметри рядка запиту (query string), значення маршруту (route values), HTTP-заголовки (за винятком службових та аутентифікаційних), значення файлів cookie, тіло запиту у відповідних форматах. Кожне витягнуте значення супроводжується метаданими: ідентифікатор джерела (наприклад, «query:username», «header:X-Custom-Header», «body:json:user.email»), що використовується пізніше для пояснюваності рішень.

Етап 3. Багатоетапна нормалізація. Кожне витягнуте значення проходить через конвеєр нормалізації, що складається з п'яти послідовних трансформацій. Метою нормалізації є нейтралізація типових методів обфускації payload'у, що використовуються зловмисниками для обходу засобів захисту. Детальний опис нормалізатора наведено у підпункті 3.1.

Етап 4. Виявлення двома паралельними каналами. Нормалізоване значення подається одночасно на два незалежних аналізатори: сигнатурний (швидкий, на основі регулярних виразів) та ML (на основі попередньо навченої нейронної мережі). Робота двох каналів виконується паралельно, що мінімізує загальну затримку. Сигнатурний канал повертає список спрацьованих правил з їх вагами, ML-канал – оцінку ймовірності того, що

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		30

значення є SQL-ін'єкцією. Детальний опис каналів наведено у підпунктах 3.1 та 3.2.

Етап 5. Композитний скоринг та прийняття рішення. На цьому етапі обчислюється фінальний ризик-скор як зважена сума нормованих оцінок з обох каналів та контекстного коефіцієнта. Якщо отриманий скор перевищує налаштовуваний поріг, приймається рішення про блокування запиту. Залежно від поточного режиму роботи (Block, Shadow, Detect Only) виконується одна з трьох дій: повернення HTTP 403 Forbidden (Block); пропуск запиту з логуванням як заблокованого «у тінь» (Shadow); пропуск запиту з логуванням як інформаційного повідомлення (Detect Only).

Етап 6. Журналювання. Усі рішення, що передбачають реагування (тобто рішення про блокування або фіксацію Shadow-події), фіксуються у журналі подій. Журнал містить повний контекст: ідентифікатор події (UUID), часова мітка з мілісекундною точністю, IP-адреса клієнта (з урахуванням заголовків X-Forwarded-For та X-Real-IP), URI та метод запиту, ідентифікатор користувача (якщо застосунок виконав автентифікацію), нормалізований вміст підозрілого значення, ідентифікатор поля-джерела, перелік спрацьованих сигнатур, оцінка ML-моделі, фінальний композитний скор, прийняте рішення. Запис у журнал виконується асинхронно, щоб не блокувати основний потік обробки запиту.

Принцип роботи багатоетапного нормалізатора payload'у

Нормалізатор payload'у є ключовим елементом запропонованої архітектури, що відрізняє розроблювану систему від більшості сигнатурних аналогів. Його завдання – привести вхідне значення до канонічної форми, яка відображає семантику payload'у, незалежно від конкретного способу його кодування у HTTP-запиті. Конвеєр нормалізації складається з п'яти послідовних стадій, кожна з яких виконує одне чітко обмежене перетворення.

Стадія 1. Рекурсивне URL-декодування. Зловмисники часто застосовують подвійне (а іноді й потрійне) URL-кодування корисного

навантаження для обходу простих фільтрів. Наприклад, символ «'» може бути закодований як %27, або як %2527 (де перша лапка кодується ще раз), або як %252527. Стадія 1 виконує URL-декодування у циклі, поки декодоване значення відрізняється від попереднього, з обмеженням на 3 ітерації для запобігання нескінченному циклу на сконструйованих payload'ах.

Стадія 2. Декодування hex та Unicode escape-послідовностей. SQL-діалекти більшості СУБД підтримують конструкції на кшталт CHAR(0x27) або CONCAT(N'a', X'62'). У контексті payload'у атакуючий може використовувати такі конструкції для маскування ключових слів атаки. Стадія 2 розпізнає типові форми (\xHH, %uHHHH, \uHHHH, &#xHHHH; – для HTML-сутностей) та декодує їх у відповідні Unicode-символи.

Стадія 3. Видалення SQL-коментарів. Більшість СУБД підтримує декілька форм коментарів у запитах: дворядкові ANSI-коментарі (-- до кінця рядка), C-style блокові коментарі (/* ... */), MySQL-специфічні коментарі з потенційним виконанням коду (/*!50000 SELECT */), решітки в MySQL (#). Зловмисники використовують ці форми для розбиття ключових слів атаки на фрагменти, які не розпізнаються прямими сигнатурами (наприклад, UNI/**/ON SE/**/LECT замість UNION SELECT). Стадія 3 видаляє всі форми коментарів, повертаючи payload до неперервної форми.

Стадія 4. Згортка пробільних символів та нормалізація регістру. Численні послідовні пробіли, табуляції, символи нового рядка, нерозривні пробіли (NBSP) у payload'і замінюються на одиничні звичайні пробіли. Це усуває можливість маскування фрагментів атаки шляхом нестандартного пробільного форматування. Після цього весь payload приводиться до нижнього регістру (за винятком значень у рядкових літералах, якщо такі залишаються). Це уніфікує payload'и виду «Union seLecT» та «union select», які функціонально еквівалентні.

Стадія 5. Заміна рядкових літералів на токен <STR>. SQL-рядкові літерали у лапках можуть містити довільний користувацький контент, не

пов'язаний з логікою атаки. У нормалізованій формі такий вміст замінюється єдиним токеном <STR>, що зменшує «шум» для ML-моделі та робить сигнатурні правила менш чутливими до конкретних значень. Наприклад, рядки `SELECT * FROM users WHERE name='admin' OR '1'='1'` та `SELECT * FROM users WHERE name='foo' OR '2'='2'` після стадії 5 матимуть однакову нормалізовану форму: `SELECT * FROM users WHERE name=<STR> OR <STR>=<STR>`.

Слід наголосити, що нормалізатор НЕ виконує токенизацію SQL у строгому лексичному сенсі (це не парсер), а лише послідовність текстових перетворень, які зберігають однозначну відповідність до оригінального payload'у. Усі трансформації є детермінованими та зворотно документованими у журналі, що забезпечує прозорість обробки.

Принцип роботи сигнатурного аналізатора

Сигнатурний аналізатор виконує перевірку нормалізованого значення проти бази правил, кожне з яких являє собою регулярний вираз з мета-інформацією. Структура одного правила:

- id – унікальний ідентифікатор правила (наприклад, «SQLI-001»);
- category – категорія атаки («union-based», «boolean-based», «time-based», «stacked-queries», «error-based», «out-of-band»);
- severity – рівень серйозності («high» або «low»);
- pattern – компільований .NET Regex з опціями IgnoreCase та CultureInfoInvariant;
- weight – вага правила, що сумується у фінальний сигнатурний скор;
- description – людино-читаний опис правила для журналу.

Поділ правил на дві категорії за серйозністю критично важливий: правила категорії «high» (наприклад, точне співпадіння з UNION SELECT, OR 1=1, або з вбудованими функціями типу SLEEP, BENCHMARK, WAITFOR) є практично однозначними індикаторами атаки і мають велику вагу, що часто самотійно перевищує поріг блокування. Правила категорії «low» (наприклад,

наявність окремих ключових слів SQL у непідозрілому контексті, ймовірно втручання знаків лапки) самі по собі не є достатньою підставою для блокування, але їх сумарний внесок з декількох спрацювань може підняти скор до значущого рівня.

Для запобігання повного відмови у сервісі через помилку у складному регулярному виразі (ReDoS – Regular Expression Denial of Service), кожне правило виконується з жорстким timeout у 100 мс. Якщо timeout перевищено, спрацювання правила вважається таким, що не відбулося, з фіксацією у журналі діагностичної інформації. Окрім того, при ініціалізації системи виконується автоматична перевірка кожного правила на синтетичних патологічних рядках великої довжини; правила, що відмовляють пройти цю перевірку, вилучаються з активної бази.

Сигнатурний скор обчислюється як проста сума ваг усіх спрацьованих правил, обмежена зверху значенням 100:

$$\text{SigScore} = \min(100, \sum \text{weight}(r_i)) \quad (3.1)$$

Це дозволяє безпосередньо інтерпретувати сигнатурний скор як шкалу 0–100 та поєднувати його з ML-оцінкою на тому самому масштабі.

Принцип роботи ML-каналу

ML-канал реалізує виявлення SQL-ін'єкцій на основі моделі машинного навчання, попередньо натренованої на змішаному датасеті (опис датасету та процедури навчання – у Розділі 4). Модель експортована у формат ONNX та завантажується через бібліотеку Microsoft.ML.OnnxRuntime версії 1.26.

При запуску middleware екземпляр InferenceSession ONNX-моделі створюється один раз і використовується протягом усього часу роботи застосунку, що уникає накладних витрат на повторну ініціалізацію моделі. InferenceSession є потокобезпечним об'єктом, тому одночасна обробка запитів декількома потоками не вимагає синхронізації.

Підготовка вхідних даних для моделі залежить від конкретної архітектури (TF-IDF + LR або CNN на рівні символів). У випадку CNN-моделі, нормалізований рядок перетворюється у послідовність цілих чисел згідно зі словником символів (фіксованої довжини 256 з паддингом нулями), що потім подається у вхідний тензор моделі. У випадку моделі TF-IDF + LR, рядок проходить векторизацію через попередньо побудований TF-IDF-екстрактор та подається у класифікатор. У обох випадках вихідом моделі є скаляр у діапазоні $[0, 1]$, що представляє ймовірність того, що значення є SQL-ін'єкцією.

Для зведення ML-оцінки до того ж масштабу 0–100, що й сигнатурний скор, виконується множення на 100:

$$\text{MLScore} = \text{round}(\text{prob_sqli} * 100) \quad (3.2)$$

Час інференсу CNN-моделі на CPU становить, за результатами навантажувального тестування, у середньому 2.4 мс на запит для одиничного значення довжиною до 256 символів. Це є прийнятним для більшості сценаріїв веб-обслуговування. Якщо ML-канал часом перевищує налаштовуваний поріг (за замовчуванням 50 мс), результат вважається тимчасово недоступним, та фінальне рішення приймається лише на основі сигнатурного скору (з відповідним записом у журнал).

Принцип роботи композитного скорингу

Композитний ризик-скор обчислюється як зважена сума трьох складових:

$$\text{RiskScore} = \alpha \cdot \text{SigScore} + \beta \cdot \text{MLScore} + \gamma \cdot \text{ContextWeight} \quad (3.3)$$

Де α , β , γ – налаштовувані коефіцієнти, що мають у сумі дорівнювати 1 (за замовчуванням $\alpha=0.45$, $\beta=0.45$, $\gamma=0.10$). Налаштування коефіцієнтів дозволяє адміністратору схилити рішення системи у бік сигнатурного або ML-аналізу залежно від специфіки захищеного застосунку.

Контекстний коефіцієнт ContextWeight враховує додаткові ознаки, що не пов'язані безпосередньо з вмістом payload'у, але опосередковано впливають на ризик-оцінку. Зокрема:

- частота запитів з даної IP-адреси за останні 60 секунд (висока частота
- індикатор автоматизованого сканування);
- наявність попередніх заблокованих запитів з цієї ж IP-адреси за останні 24 години;
- тип цільового endpoint'у (наприклад, ендпоінти автентифікації мають вищу базову вагу, ніж ендпоінти отримання публічного контенту);
- наявність відомих сигнатур у User-Agent (sqlmap, nessus, nikto тощо).

Рішення про дію приймається порівнянням RiskScore з налаштовуваним порогом (за замовчуванням 60):

- RiskScore < threshold: запит пропускається, журналювання відсутнє (за винятком випадків Verbose-режиму).
- RiskScore $\geq$ threshold: запит блокується (або фіксується у тінь, залежно від режиму).

Запропонована схема композитного скорингу забезпечує найкращий компроміс між повнотою виявлення та контролем хибно-позитивних спрацювань: жодне з джерел свідчень самостійно не приймає рішення про блокування, проте їх збалансоване поєднання дає змогу впевнено реагувати на більшість реальних атак, водночас не порушуючи легітимний трафік.

3.2 Розробка структурної схеми

Структурна схема системи (повна) являє собою декомпозицію розроблюваного програмного забезпечення на елементарні ланки (компоненти) з відображенням усіх внутрішніх зв'язків між ними та зовнішніх зв'язків з середовищем (захищеним застосунком, базою даних, файловою системою). Структурна схема дозволяє оцінити масштаб системи, чітко

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		36

визначити відповідальність кожного компонента та слугує основою для подальшого програмного проектування.

На рисунку 3.2 наведено повну структурну схему розроблюваної системи AntiSQLi-WAF.

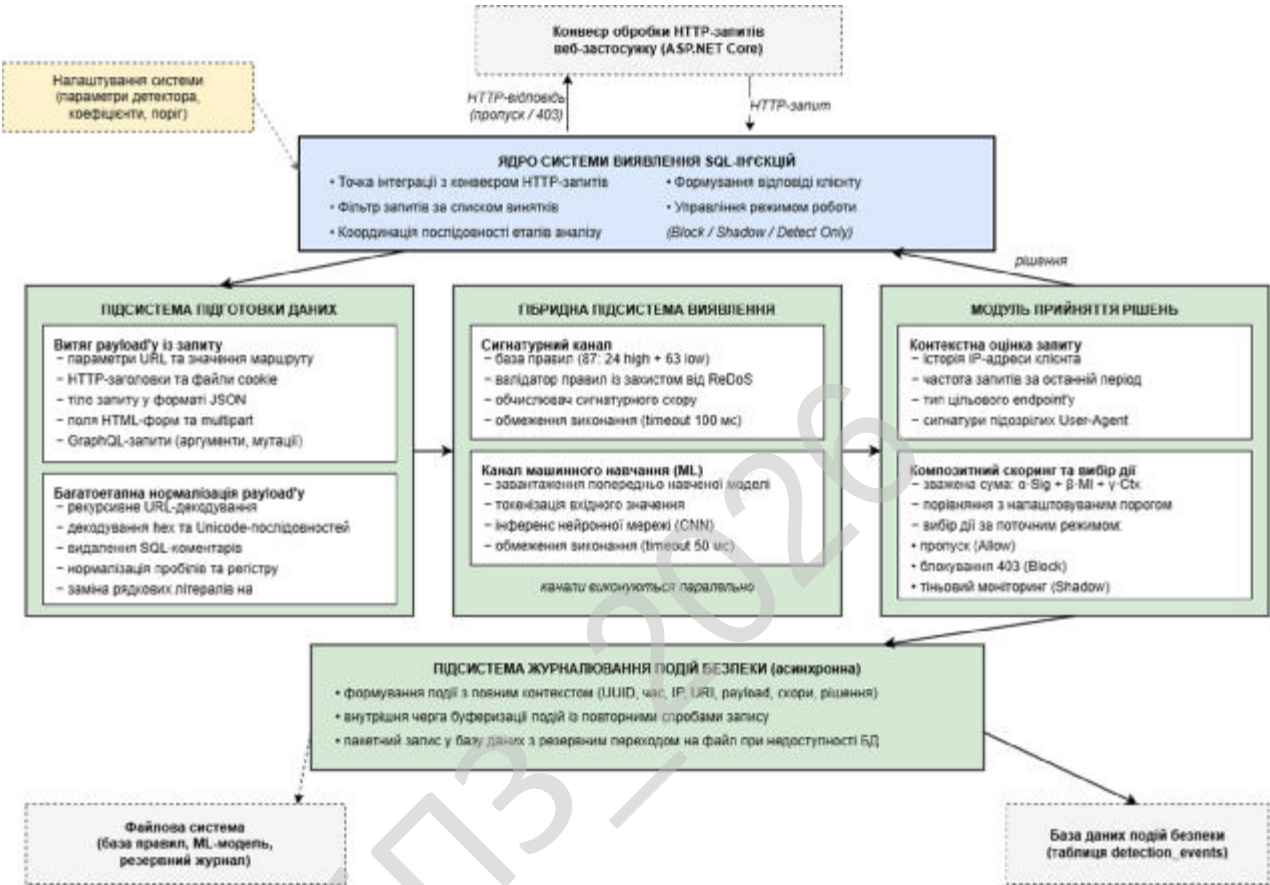


Рисунок 3.2 – Структурна схема системи AntiSQLi-WAF (повна)

Розглянемо детальніше функціональне призначення та внутрішню структуру кожного з відображених на схемі компонентів.

Компонент AntiSqliMiddleware

AntiSqliMiddleware є основним компонентом системи та входить у конвеєр обробки запитів ASP.NET Core. Реалізує інтерфейс IMiddleware платформи. Має одну публічну точку входу – асинхронний метод InvokeAsync(HttpContext context, RequestDelegate next), який викликається фреймворком для кожного вхідного запиту. Внутрішня логіка цього методу реалізує описаний у пункті 3.1 шестиетапний процес обробки. Компонент є

stateless-об'єктом, проте отримує посилання на інші компоненти системи через впровадження залежностей (DI):

- IPayloadExtractor – для отримання payload'у з різних частин запиту;
- IPayloadNormalizer – для нормалізації;
- ISignatureAnalyzer – для сигнатурного аналізу;
- IMlDetector – для ML-аналізу;
- IRiskScorer – для обчислення композитного скору;
- IDetectionLogger – для запису подій;
- IOptions<AntiSqliOptions> – для доступу до налаштувань.

Така декомпозиція дозволяє: повністю тестувати middleware у ізоляції шляхом підстановки тестових реалізацій інтерфейсів; перевизначати окремі компоненти у конкретних розгортаннях без зміни ядра (наприклад, замінити дефолтний логер на адаптований до конкретного SIEM); виконувати модульні оновлення системи.

Компонент PayloadExtractor

PayloadExtractor відповідає за витяг усіх потенційно-небезпечних значень з HTTP-запиту. Внутрішньо складається з низки спеціалізованих субкомпонентів, кожен з яких відповідає за один тип джерела:

- QueryStringExtractor – параметри рядка запиту;
- RouteValuesExtractor – значення маршруту;
- HeadersExtractor – HTTP-заголовки (з фільтром службових);
- CookiesExtractor – значення файлів cookie;
- JsonBodyExtractor – рекурсивний витяг строкових значень з JSON-тіла;
- FormBodyExtractor – поля форм application/x-www-form-urlencoded та multipart/form-data;
- GraphQLExtractor – спеціалізована логіка для GraphQL-запитів (parse операції, аргументи).

Кожен субкомпонент повертає колекцію об'єктів PayloadEntry, кожен з яких містить три поля: SourceId (рядок-ідентифікатор джерела, наприклад

«query:userId» або «body:json:user.name»), Value (саме значення для аналізу), Index (порядковий номер для впорядкування результатів). Загальний інтерфейс IPayloadExtractor агрегує результати усіх субкомпонентів та повертає єдину колекцію.

Важливою архітектурною особливістю PayloadExtractor є те, що він НЕ модифікує оригінальний HTTP-контекст – лише читає з нього інформацію. Це гарантує, що подальші компоненти конвеєра ASP.NET Core отримають незмінений запит. Для уникнення повторного парсингу тіла запиту, після першого читання тіло перемотується (Request.Body.Seek(0, SeekOrigin.Begin)), що вимагає попереднього виклику Request.EnableBuffering() для активації буферизованого потоку.

Компонент PayloadNormalizer

PayloadNormalizer реалізує описаний у пункті 3.1 п'ятиетапний конвеєр нормалізації. Кожна стадія представлена окремим класом, що реалізує спільний інтерфейс INormalizationStage:

- RecursiveUrlDecoder – рекурсивне URL-декодування.
- HexUnicodeDecoder – декодування hex та Unicode escape-послідовностей.
- SqlCommentStripper – видалення SQL-коментарів усіх форм.
- WhitespaceCaseNormalizer – згортка пробілів та нормалізація регістру.
- StringLiteralReplacer – заміна рядкових літералів на <STR>.

Стадії послідовно застосовуються до payload'у у тому порядку, у якому вони зареєстровані у компонувавальному кореневі (composition root) застосунку. Це дозволяє адміністратору перевизначити порядок або вимкнути окремі стадії через конфігурацію без зміни коду. Кожна стадія додатково повертає діагностичну інформацію про здійснені перетворення (наприклад, «stripped 2 comments», «decoded 1 URL level»), що зберігається в об'єкті NormalizationTrace та може бути використана для пояснюваності.

Компонент SignatureAnalyzer

SignatureAnalyzer оцінює нормалізований payload проти бази сигнатурних правил. Внутрішня структура:

- RuleRepository – інкапсулює завантаження правил з JSON-файлу (шлях до якого налаштовується), кешує компільовані Regex у пам'яті, забезпечує гаряче перерачитування файлу при його зміні без перезапуску системи (через FileSystemWatcher).

- RuleValidator – на старті проганяє кожне правило проти набору тестових патогенних рядків, виявляє правила з потенційною уразливістю до ReDoS, заносить інформацію у журнал.

- RuleEvaluator – основний обчислювач, який послідовно (або паралельно, при великій кількості правил) застосовує усі активні правила до вхідного значення; кожне правило виконується з timeout 100 мс.

Базовий набір правил, що постачається разом з системою, охоплює основні категорії SQL-ін'єкційних атак: UNION-based, Boolean-based blind, Time-based blind, Error-based, Stacked queries, Out-of-band. Загальна кількість правил у базовому наборі – 87, з яких 24 належать до категорії High Severity та 63 – Low Severity. Адміністратор може доповнити, відключити або модифікувати будь-які правила через редагування JSON-файлу.

Компонент MlDetector

MlDetector інкапсулює інтеграцію з ML-моделлю через ONNX Runtime. Внутрішня структура:

- OnnxSessionProvider – лінива ініціалізація екземпляра InferenceSession; шлях до файлу моделі завантажується з конфігурації; підтримується механізм гарячої заміни моделі (нова версія моделі може бути завантажена без перезапуску застосунку).

- InputTokenizer – підготовка вхідного тензора; для CNN-моделі реалізує отримання індексів символів з фіксованої довжини 256 (з паддингом нулями); для TF-IDF моделі – векторизацію через попередньо побудований словник.

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		40

– InferenceExecutor – виконання моделі з timeout-захистом (50 мс); декодування виходу; повернення значення MlScore.

Розмежування цих субкомпонентів дозволяє замінювати модель та її токенизатор одночасно (через одночасне постачання нового онх-файлу та json-словника), без необхідності зміни логіки middleware або компонента-обгортки. Це робить систему придатною для дослідницької роботи: достатньо натренувати у Python нову модель, експортувати у ONNX та відповідний файл словника, після чого система автоматично переключиться на нову версію детектора.

Компонент RiskScorer

RiskScorer обчислює фінальний композитний ризик-скор за описаною у пункті 3.1 формулою. Внутрішня структура:

– CoefficientProvider – забезпечує доступ до налаштовуваних коефіцієнтів α , β , γ та порога з конфігурації.

– ContextWeighter – обчислює ContextWeight з урахуванням метаданих запиту, історії IP-адреси та цільового endpoint'у.

– ScoreCombiner – фактичне обчислення зваженої суми.

– DecisionMaker – порівнює фінальний скор з порогом, обирає одну з трьох дій (Allow, Block, Shadow) залежно від поточного режиму.

Компонент DetectionLogger

DetectionLogger відповідає за асинхронний запис подій детекції у журнал. Особливості:

– LogEventBuilder – формує повноструктурний об'єкт DetectionEvent (UUID, часова мітка, ip, uri, method, нормалізований payload, спрацьовані правила, ml-score, score, decision, додаткові метадані).

– PostgreSQLSink – використовує Npgsql 10 та Dapper для запису у таблицю detection_events; підтримує batch-запис до 100 подій за одну операцію для зменшення накладних витрат на мережеву комунікацію з базою.

– `BufferingQueue` – внутрішня черга на основі `System.Threading.Channels`, що дозволяє при тимчасовій недоступності БД зберігати події у пам'яті та повторювати спроби з експоненціальним backoff.

– `FallbackFileSink` – резервний приймач: якщо БД недоступна понад налаштовуваний час (за замовчуванням 5 хвилин), події починають дублюватися в JSON-файл на диску для подальшого імпорту після відновлення БД.

Така архітектура журналювання гарантує, що тимчасові проблеми з базою даних (мережеві затримки, перезавантаження СУБД, конфлікти блокувань) НЕ блокують основну функцію детекції та не призводять до втрати подій безпеки.

Зовнішні компоненти та інтерфейси

Окрім внутрішніх компонентів, на структурній схемі відображено три зовнішні елементи:

– `ASP.NET Core Pipeline` – у який вбудовується `AntiSqliMiddleware`; зв'язок реалізується через спеціальний метод-розширення `IApplicationBuilder.UseAntiSqliWaf()`.

– `PostgreSQL Database` – зовнішня СУБД, що зберігає журнал детекції; зв'язок через `Npgsql`-конект через рядок підключення, що зчитується з конфігурації.

– `File System` – використовується для зчитування JSON-файлу правил, ONNX-файлу моделі та (резервно) для скидання журналу при недоступності БД.

Усі зовнішні взаємодії абстраговані через інтерфейси (`IDbConnection`, `IFileSystem`), що уможливорює модульне тестування системи без реальних зовнішніх залежностей.

Підсумок структурної декомпозиції

Підсумовуючи розглянуту структурну декомпозицію, розроблювана система складається з 7 основних компонентів верхнього рівня

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		42

(AntiSqliMiddleware, PayloadExtractor, PayloadNormalizer, SignatureAnalyzer, MlDetector, RiskScorer, DetectionLogger), кожен з яких має чітко визначену відповідальність та внутрішню декомпозицію на 3–7 субкомпонентів. Загальна кількість елементарних класів у системі – близько 40, що відповідає типовій складності middleware-компонента промислового рівня.

Усі компоненти взаємодіють через інтерфейси, що уможливорює: модульне тестування; розширення системи без зміни ядра; адаптацію системи до різних сценаріїв розгортання (зокрема, через перевизначення компонентів у DI-контейнері). Така архітектура відповідає принципам SOLID, є типовою для сучасних .NET-застосунків та сприяє довгостроковій підтримованості розробленого ПЗ.

3.3 Розробка функціональної схеми

Функціональна схема системи відображає логіку її роботи, а саме – послідовність функцій (обробок), які виконуються над вхідними даними для отримання вихідних даних. На відміну від структурної схеми, що показує «з чого складається» система, функціональна схема показує «як працює» система – потоки даних та порядок їх обробки. Основні функціональні вузли системи зображено прямокутниками, а зв'язки між ними – лініями зі стрілками, що відповідає напрямку руху даних або керування. Схема відображає вхідні дані у вигляді HTTP-запиту, послідовність функцій обробки (витяг payload'у, перевірку винятків, нормалізацію, паралельний сигнатурний та ML-аналіз, композитний скоринг), асинхронну функцію журналювання та два можливих виходи системи. Функціональна схема розроблюваної системи AntiSQLi-WAF наведена на рисунку 3.3.

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		43

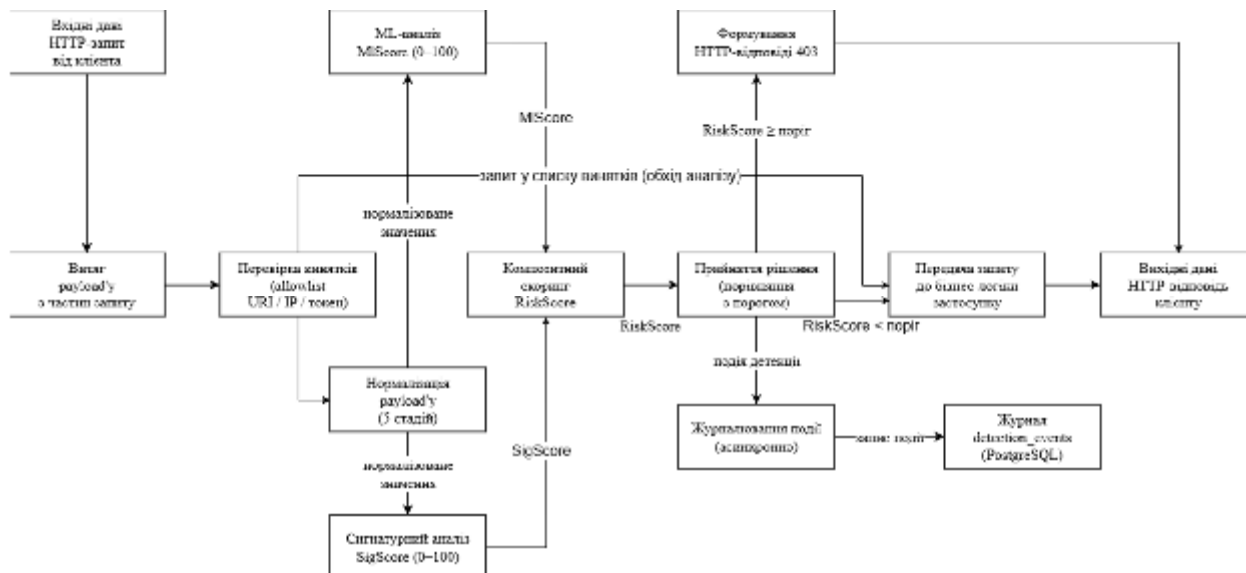


Рисунок 3.3 – Функціональна схема системи AntiSQLi-WAF

Розглянемо детальніше функціональні зв'язки, відображені на схемі.

Вхідні дані

Єдиним вхідним даним системи є HTTP-запит, що надходить з мережі (від HTTP-клієнта). Запит містить:

- метод HTTP (GET, POST, PUT, DELETE, PATCH тощо);
- URI запиту (схема, хост, шлях, query string);
- набір HTTP-заголовків;
- файли cookie;
- тіло запиту (опціонально, для методів з payload);
- метадані з'єднання (IP-адреса клієнта, час прийому, протокол TLS-сесії).

Запит надходить у вигляді об'єкта HttpContext, сформованого фреймворком ASP.NET Core. Об'єкт стандартизований та надає уніфікований API для доступу до усіх перелічених елементів незалежно від конкретного HTTP-сервера (Kestrel, IIS, HTTP.sys).

Функція «Витяг payload'у»

Функція виконує перетворення `HttpContext` → `List<PayloadEntry>`. Внутрішньо реалізована паралельним викликом усіх субекстракторів

(компонент PayloadExtractor, описаний у пункті 3.2), результати яких агрегуються у єдину колекцію. Кожен PayloadEntry містить три атрибути: SourceId, Value, Index.

Функція є чистою (не має побічних ефектів). Для оптимізації при значній кількості однотипних значень (наприклад, JSON з масивом 1000 елементів) функція реалізує адаптивний strategie: при перевищенні 100 значень у JSON-тілі переходить у режим вибіркового аналізу (аналізуються лише ключові поля, виходячи з мета-інформації про схему).

Функція «Перевірка allowlist»

Функція перевіряє, чи треба взагалі проводити аналіз для даного запиту. Перевірка виконується за наступними правилами (логічне АБО):

- запит до URI, який міститься у налаштованому списку винятків (за глобальним або префіксним співпадінням);
- запит надходить від клієнта з IP-адреси, що внесена у білий список (наприклад, внутрішній моніторинг);
- запит має спеціальний заголовок з налаштованим bypass-токеном (для legitimate автоматизованого тестування).

Якщо хоча б одна умова виконана, аналіз пропускається, запит передається наступному компоненту конвеєра, а у журнал не пишеться жодної події.

Функція «Нормалізація payload'у»

Функція виконує перетворення PayloadEntry → NormalizedPayload через п'ять послідовних стадій (детально описаних у пункті 3.1). NormalizedPayload містить: нормалізоване значення (рядок), оригінальне значення (рядок), діагностичний слід (список застосованих перетворень з результатами). Стадії нормалізації виконуються по черзі, кожна отримує результат попередньої.

Принципова особливість функції – детермінованість: для одного й того самого вхідного PayloadEntry функція завжди повертає однаковий

NormalizedPayload, що уможливлює кешування результатів для повторюваних запитів та спрощує тестування.

Функції «Сигнатурний аналіз» та «ML-аналіз» (паралельно)

Дві функції, що приймають один і той самий NormalizedPayload та повертають дві незалежні оцінки. Виконуються одночасно у двох task'ах через Task.WhenAll(), що дозволяє максимально використовувати багатоядерність системи.

Сигнатурний аналіз повертає об'єкт SignatureResult, що складається з SigScore (сумарна вага спрацьованих правил, 0–100) та списку спрацьованих правил з їх метаданими.

ML-аналіз повертає об'єкт MLResult, що складається з MLScore (нормований 0–100) та внутрішньої впевненості моделі (raw probability, 0–1). Може мати спеціальний стан «недоступний» (наприклад, через timeout).

Функція «Композитний скоринг»

Функція обчислює фінальний RiskScore та приймає рішення (Decision). Вхідні дані – SignatureResult, MLResult, ContextInfo. Вихід – пара (RiskScore, Decision). Decision є перерахуванням з трьох можливих значень:

- Allow – запит дозволено пройти далі без подальших дій.
- Block – запит має бути заблокований (повернути HTTP 403).
- ShadowBlock – запит має пройти далі, але має бути зафіксований у журналі як такий, що «був би заблокований».

Differences between Block та ShadowBlock визначаються поточним режимом роботи системи (з конфігурації). У режимі Block – рішення про блокування виконується реально; у режимі Shadow – фактичне блокування пропускається. Це дозволяє безпечно вводити систему в експлуатацію.

Функція «Журналювання»

Функція виконує асинхронний запис DetectionEvent у базу даних PostgreSQL та (за потреби) у резервний файл. Не блокує основний потік

обробки. Реалізована як черга з фоновим робочим, що періодично виконує batch-запис.

Вихідні дані функціональної схеми

У залежності від прийнятого рішення, система формує один з двох виходів:

- Allow / ShadowBlock – управління передається next() – наступному middleware у конвеєрі, який врешті досягає бізнес-логіки застосунку та повертає клієнту нормальну відповідь.

- Block – конвеєр припиняється, формується HTTP-відповідь зі статусом 403 Forbidden та налаштовуваним JSON-тілом (за замовчуванням – мінімальна інформація без витоку деталей детекції клієнту).

Окрім зовнішнього виходу до клієнта, в обох випадках відбувається другий, асинхронний вихід – запис у журнал детекції (для Block та ShadowBlock).

Інформаційні потоки

На функціональній схемі чітко виділяються три основні інформаційні потоки. Перший потік – основний потік обробки запиту: HTTP-запит → витяг payload'у → нормалізація → паралельний аналіз → скоринг → рішення → дія. Усі ці кроки виконуються синхронно у контексті обробки одного запиту і у сумі повинні займати не більше декількох мілісекунд.

Другий потік – асинхронне журналювання: подія детекції → черга → фоновий робочий → запис у БД. Цей потік повністю розгалужений з основним і не блокує клієнтський запит.

Третій потік – потік конфігурації: файл appsettings.json та змінні оточення → IOptions<AntiSqliOptions> → споживання усіма компонентами. У штатному режимі цей потік активується один раз при старті застосунку; у разі активації механізму гарячого перерахування – періодично при зміні файлу.

Чітка декомпозиція системи на функції з мінімальним перехреснуванням потоків даних спрощує верифікацію логіки роботи системи, її тестування та подальший розвиток.

3.4 Розробка діаграми процесів

Діаграма процесів є візуальним представленням графу процесів, що відбуваються у системі. Граф процесів є різновидом графу станів скінченного автомата, вершинами якого є визначені дії, а переходи між якими відбуваються по завершенню цих дій. На відміну від функціональної схеми, що показує потоки даних між функціями, діаграма процесів акцентує увагу на життєвому циклі обробки запиту як на послідовності станів-процесів, кожен з яких має чіткий початок, виконує визначену дію та має визначений вихід. Кожен вузол діаграми відповідає окремому процесу як атомарній одиниці обробки з визначеними передумовами активації та результатом, а кожен перехід – умові або події, що ініціює наступний процес. Така форма подання дозволяє явно відобразити паралельні підпроцеси, асинхронні гілки та точки прийняття рішень, які у функціональній схемі залишаються неявними. Графічне зображення процесів виконане відповідно до вимог ДСТУ ISO 5807:2016: процеси показано овалами, а переходи між ними – вигнутими лініями зі стрілками. Для розроблюваної системи AntiSQLi-WAF діаграма процесів дозволяє формалізувати поведінку компонента-перехоплювача HTTP-запиту, включно зі сценаріями паралельного запуску сигнатурного та ML-аналізу, асинхронного журналювання і умовного розгалуження залежно від результату композитного скорингу. Діаграма процесів розроблюваної системи AntiSQLi-WAF наведена на рисунку 3.4.

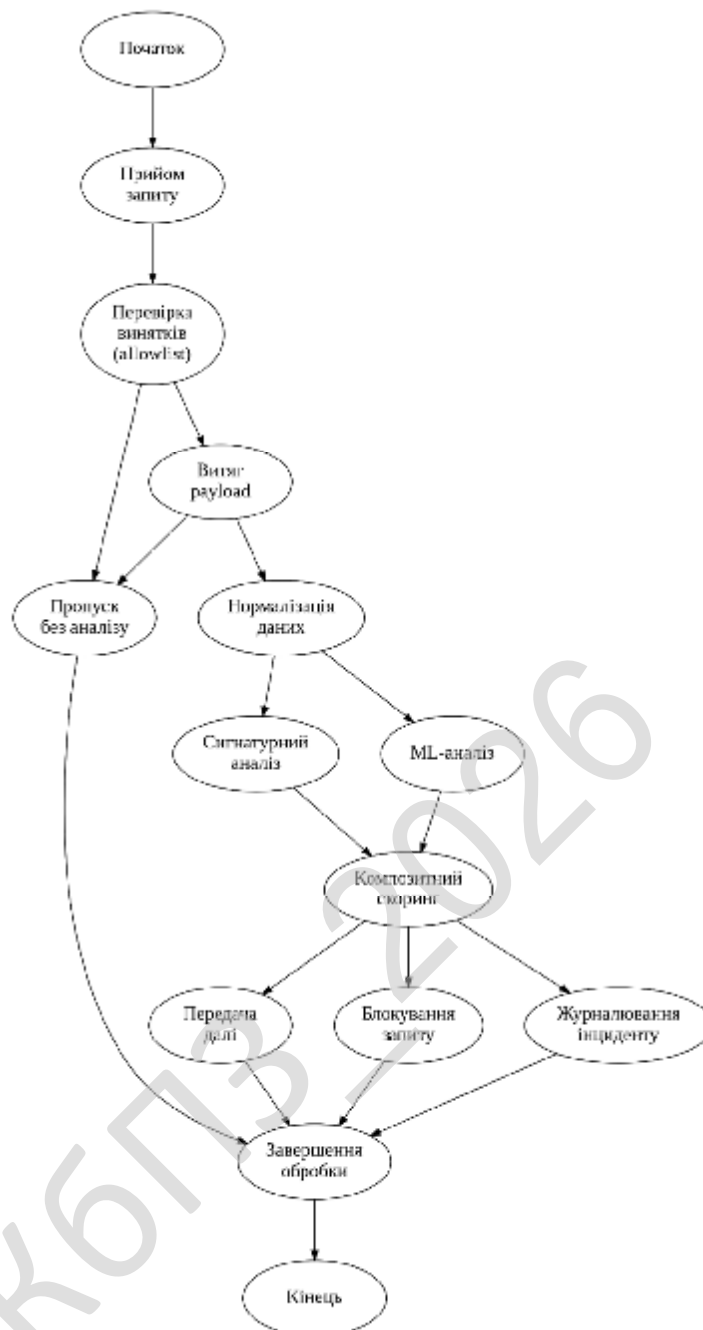


Рисунок 3.4 – Діаграма процесів системи AntiSQLi-WAF

Розглянемо детальніше кожен з процесів, відображених на діаграмі.

Процес «Прийом запити»

Початковий процес життєвого циклу обробки. Активується при отриманні HTTP-запиту мережевим стеком (Kestrel або іншим). Виконує: парсинг HTTP-протоколу (заголовків, статусного рядка); побудову HttpContext; встановлення TLS-сесії (за потреби); виклик початку конвеєра

middleware. Час виконання процесу – від мікросекунд до 1-2 мс. Завершується активацією першого middleware у конвеєрі.

Процес «Перевірка винятків»

Активується при отриманні HttpContext компонентом AntiSqliMiddleware. Виконує перевірки allowlist (URI, IP, заголовки), описані у пункті 3.3. Має два можливих переходи: при позитивній перевірці – процес «Пропуск без аналізу»; при негативній – процес «Витяг payload'у». Час виконання – мікросекунди (перевірка хеш-таблиць та регулярних виразів).

Процес «Пропуск без аналізу»

Активується при позитивній перевірці allowlist. Виконує єдину дію – виклик next() для передачі управління наступному middleware у конвеєрі ASP.NET Core. Завершується одразу без побічних ефектів.

Процес «Витяг payload'у»

Активується при негативній перевірці allowlist. Виконує паралельний витяг усіх потенційно-небезпечних значень з HttpContext (компонент PayloadExtractor). Має два можливих переходи: при отриманні непорожнього списку значень – процес «Нормалізація»; при отриманні порожнього списку – процес «Пропуск без аналізу» (з фіксацією у журналі діагностичного запису). Час виконання – від мікросекунд до 5 мс, залежно від обсягу запиту.

Процес «Нормалізація» (паралельно для кожного значення)

Активується для кожного отриманого PayloadEntry. Виконує послідовне застосування 5 стадій нормалізації. Внутрішньо може розпаралелитися на нормалізацію декількох значень одночасно. Завершується активацією процесу «Паралельний аналіз» для кожного нормалізованого payload'у. Час виконання однієї нормалізації – порядку 0.05–0.2 мс.

Процес «Паралельний аналіз»

Активується одночасним викликом двох підпроцесів: «Сигнатурний аналіз» та «ML-аналіз». Виконує Task.WhenAll() та чекає завершення обох. Після завершення активує процес «Композитний скоринг». Має додатковий

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		50

механізм timeout-захисту: якщо ML-аналіз не завершився за 50 мс, він примусово переривається, та скоринг виконується тільки на основі сигнатурного результату з відповідним записом у журнал.

Підпроцес «Сигнатурний аналіз»

Активується процесом «Паралельний аналіз». Виконує оцінку нормалізованого payload'у проти бази сигнатурних правил (компонент SignatureAnalyzer). Внутрішньо складається з циклу по правилах, кожне з яких застосовується з власним timeout 100 мс. Завершується поверненням SignatureResult. Час виконання – порядку 0.5–2 мс (залежно від кількості правил у активній базі).

Підпроцес «ML-аналіз»

Активується процесом «Паралельний аналіз» одночасно з підпроцесом «Сигнатурний аналіз». Виконує токенизацію вхідного значення, побудову вхідного тензора, виклик inference у ONNX Runtime, декодування виходу. Завершується поверненням MlResult. Час виконання – порядку 1–5 мс (для CNN-моделі). При перевищенні внутрішнього timeout 50 мс – повертає MlResult у стані «недоступний».

Процес «Композитний скоринг»

Активується завершенням процесу «Паралельний аналіз». Виконує обчислення композитного RiskScore як зваженої суми SigScore, MlScore та ContextWeight. Приймає рішення про дію (Allow/Block/ShadowBlock) шляхом порівняння RiskScore з порогом. Завершується активацією одного з трьох наступних процесів залежно від прийнятого рішення.

Процес «Журналювання» (асинхронно)

Активується процесом «Композитний скоринг» у випадку рішення Block або ShadowBlock. Виконується у фоновому task без блокування основного потоку. Формує DetectionEvent з повним контекстом, ставить у внутрішню чергу для batch-запису у PostgreSQL. У разі помилки запису –

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		51

повторює спроби з експоненціальним backoff, при тривалій недоступності – використовує резервний JSON-файл.

Процес «Передача далі» (Allow / ShadowBlock)

Активується процесом «Композитний скоринг» у випадку рішення Allow або ShadowBlock. Викликає next() для передачі управління наступному middleware у конвеєрі ASP.NET Core. Бізнес-логіка застосунку обробляє запит звичайним чином, повертає відповідь клієнту. Завершується після завершення усього конвеєра.

Процес «Блокування запиту»

Активується процесом «Композитний скоринг» у випадку рішення Block (і поточного режиму Block). Не викликає next(), а замість цього формує HTTP-відповідь зі статусом 403 Forbidden та JSON-тілом, що містить мінімальну інформацію (за замовчуванням – { «error»: «Request blocked by security policy», «incidentId»: «<UUID>» }). Завершується відправленням відповіді клієнту.

Кінцевий процес

Усі гілки виконання сходяться у кінцевому процесі, який завершує цикл обробки запиту. Внутрішньо включає завершення HTTP-сесії (керування Keep-Alive), повернення статистичних метрик у Prometheus (через стандартний механізм метрик ASP.NET Core), очистку контекстних ресурсів.

Хронологія виконання

У штатному сценарії обробки запиту (без затримок мережі та БД), повний цикл від отримання запиту до завершення кінцевого процесу займає типово 3–8 мс при сумарному обсязі payload'у до 4 КБ. Це включає 1–2 мс на парсинг HTTP, 0.5–2 мс на сигнатурний аналіз (паралельно з 1–5 мс на ML), 0.1–0.3 мс на скоринг та рішення. Журналювання, як зазначено, виконується асинхронно і не входить у час відповіді клієнту.

У сценарії блокування запиту цикл коротший, оскільки етапи «Передача далі» та виконання бізнес-логіки пропускаються; типовий час від отримання

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		52

запиту до повернення 403 – 4–10 мс. Цей час є прийнятним для будь-якого практичного веб-сервісу і не створює відчутних затримок для легітимних користувачів.

Підсумок діаграми процесів

Діаграма процесів дає системний погляд на повний життєвий цикл обробки HTTP-запиту розроблюваною системою, чітко показує усі можливі гілки виконання, точки прийняття рішень, асинхронні підпроцеси (журналювання) та хронологію подій. Така деталізація необхідна для подальшої реалізації системи – кожен процес транслюється у відповідний метод або клас, кожен перехід між процесами – у певний виклик або умовну гілку. Окрім того, діаграма процесів є цінним джерелом для документування поведінки системи для майбутніх розробників, адміністраторів та аудиторів безпеки.

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		53

4 РЕАЛІЗАЦІЯ РОБОТИ. РОЗРАХУНКИ ТА ЕКСПЕРИМЕНТАЛЬНІ ДАНІ, ЩО ПІДТВЕРДЖУЮТЬ ВІРНІСТЬ ПРОЕКТНИХ І ПРОГРАМНИХ РІШЕНЬ

У даному розділі описуються алгоритми, що реалізують функціональність системи на рівні окремих процедур, наводяться блок-схеми ключових алгоритмів, описується реалізація класів, функцій та таблиць баз даних, а також наводяться результати експериментального оцінювання ефективності системи на тестових датасетах.

4.1 Розробка блок-схем та опис алгоритмів функціонування системи

У цьому підрозділі представлено детальні блок-схеми трьох ключових алгоритмів системи: основного алгоритму роботи AntiSqliMiddleware (рисунок 4.1), алгоритму багатоетапної нормалізації payload'у (рисунок 4.4) та алгоритму композитного скорингу та прийняття рішення (рисунок 4.5). Блок-схеми побудовані з використанням умовних позначень за ДСТУ ISO 5807:2016. Для кожного алгоритму наведено детальний опис його логіки, особливостей реалізації та опрацювання граничних випадків.

Алгоритм роботи основної програми (AntiSqliMiddleware)

Алгоритм роботи AntiSqliMiddleware є основним алгоритмом системи; він описує повний цикл обробки одного HTTP-запиту. Алгоритм оформлений як асинхронний метод InvokeAsync, що відповідає інтерфейсу IMiddleware платформи ASP.NET Core. На вхід надходить об'єкт HttpContext, а результатом виконання є або передача запиту наступному компоненту конвеєра через await next(), або переривання обробки з поверненням відповіді HTTP 403. Блок-схема алгоритму наведена на рисунку 4.1.

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		54

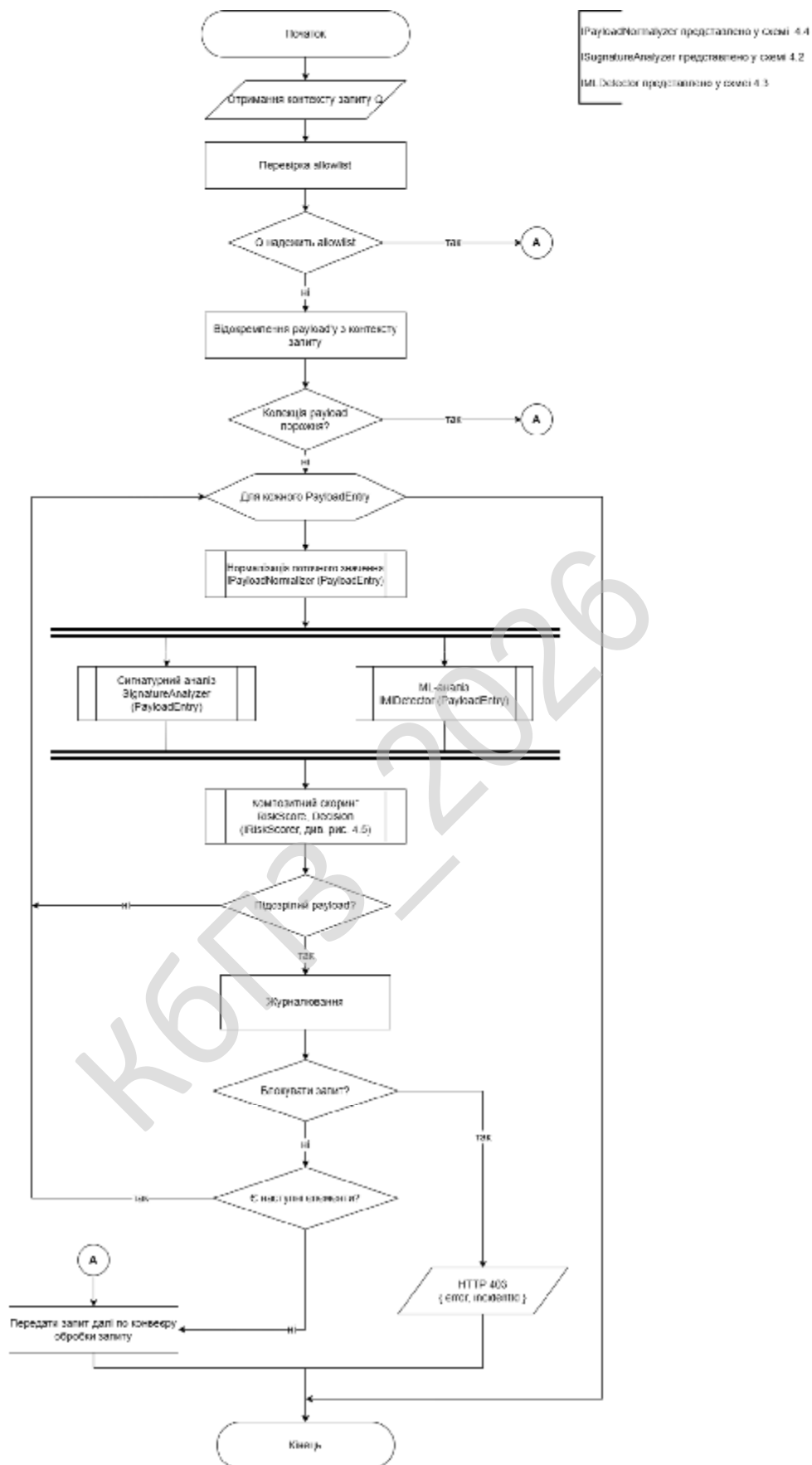


Рисунок 4.1 – Блок-схема алгоритму роботи основної програми
(AntiSqliMiddleware)

Опишемо детальніше логіку основних кроків алгоритму.

Крок 1. Перевірка allowlist. Алгоритм починається з перевірки списку винятків. Для цього використовується ефективна структура даних HashSet<string> для URI та ImmutableList<IPNetwork> для IP-адрес. Перевірка займає $O(1)$ для URI та $O(n)$ для IP, де n – кількість мережевих діапазонів у allowlist (типово менше 10). Якщо хоча б одне з правил виключення спрацює, виконання переходить одразу до кроку 9 (виклик next() без подальшої обробки).

Крок 2. Витяг payload'у з усіх частин запиту. Виконується через компонент IPayloadExtractor. Внутрішньо реалізовано як паралельний виклик усіх субекстракторів через Task.WhenAll(). Результати агрегуються у одну колекцію List<PayloadEntry>. У штатному режимі цей крок займає 0.5–3 мс, у залежності від обсягу запиту. Якщо отримана колекція порожня (наприклад, для запиту без параметрів та порожнього тіла), алгоритм переходить до кроку 9.

Крок 3. Цикл по payload-значеннях. Алгоритм входить у цикл, де для кожного отриманого PayloadEntry виконуються кроки 4–6 (нормалізація, паралельний аналіз, скоринг). Цикл оптимізований: якщо для будь-якого зі значень отриманий RiskScore перевищує поріг, обробка решти значень припиняється (short-circuit), щоб не витратити ресурси на надлишковий аналіз.

Крок 4. Нормалізація поточного значення. Виклик компонента IPayloadNormalizer, що повертає об'єкт NormalizedPayload. Деталі – у підпункті 4.1.

Крок 5. Паралельний запуск двох каналів аналізу. Використовується конструкція await Task.WhenAll(sigTask, mlTask) для одночасного виконання сигнатурного та ML-аналізу.

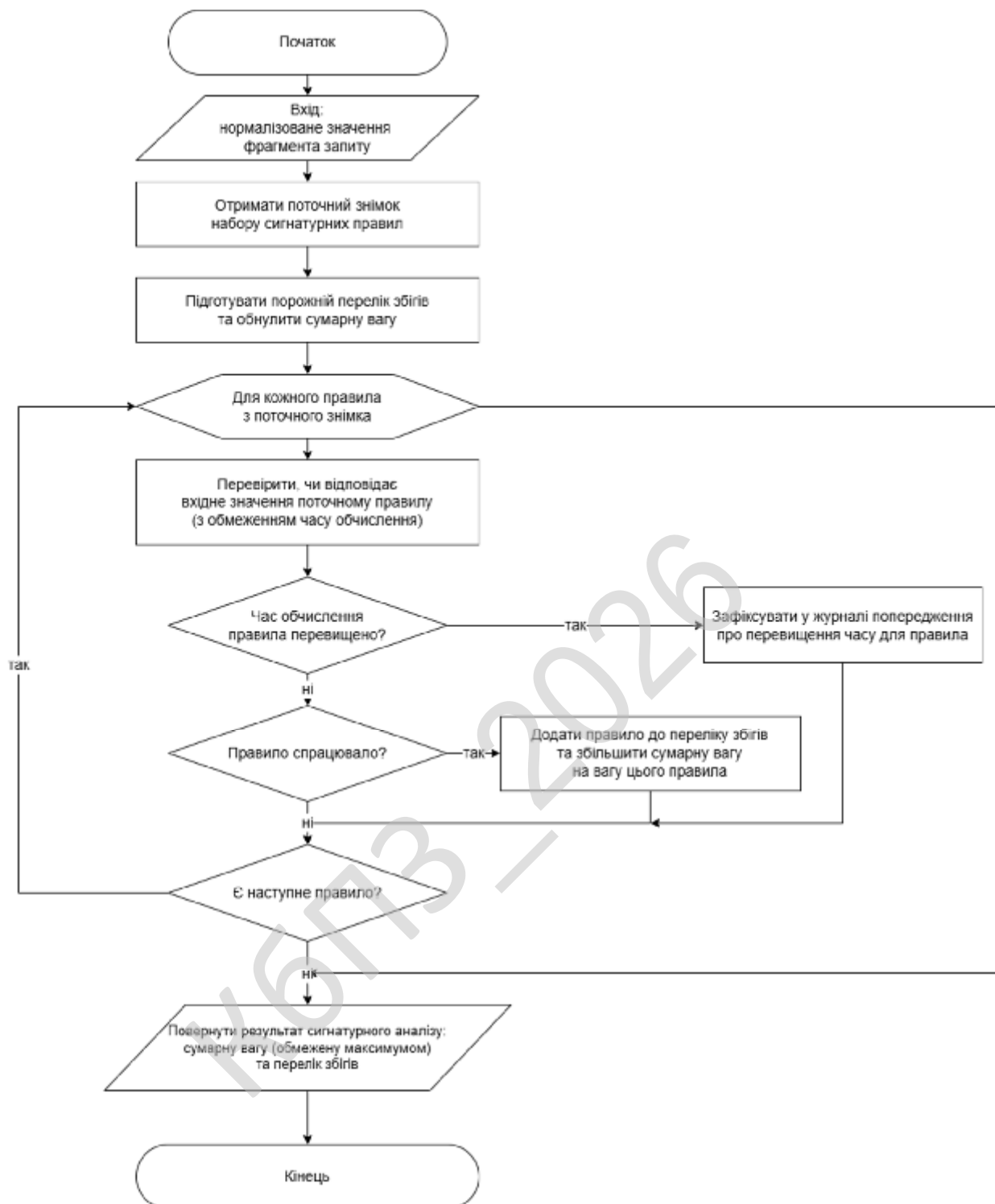


Рисунок 4.2 – Блок-схема сигнатурного аналізу (ISignatureAnalyzer)

Алгоритм ML-аналізу передбачає контрольовану деградацію за відсутності завантаженої моделі (`_session is null` → `IsUnavailable = true`) та жорстке обмеження часу виведення `InferenceTimeout = 50` мс через `WaitAsync`. На відміну від імовірнісного ML-каналу, сигнатурний аналізатор діє

детерміновано – зіставленням навантаження з регулярними виразами словника правил із захистом від ReDoS через RuleTimeout = 100 мс (рисунок 4.2).

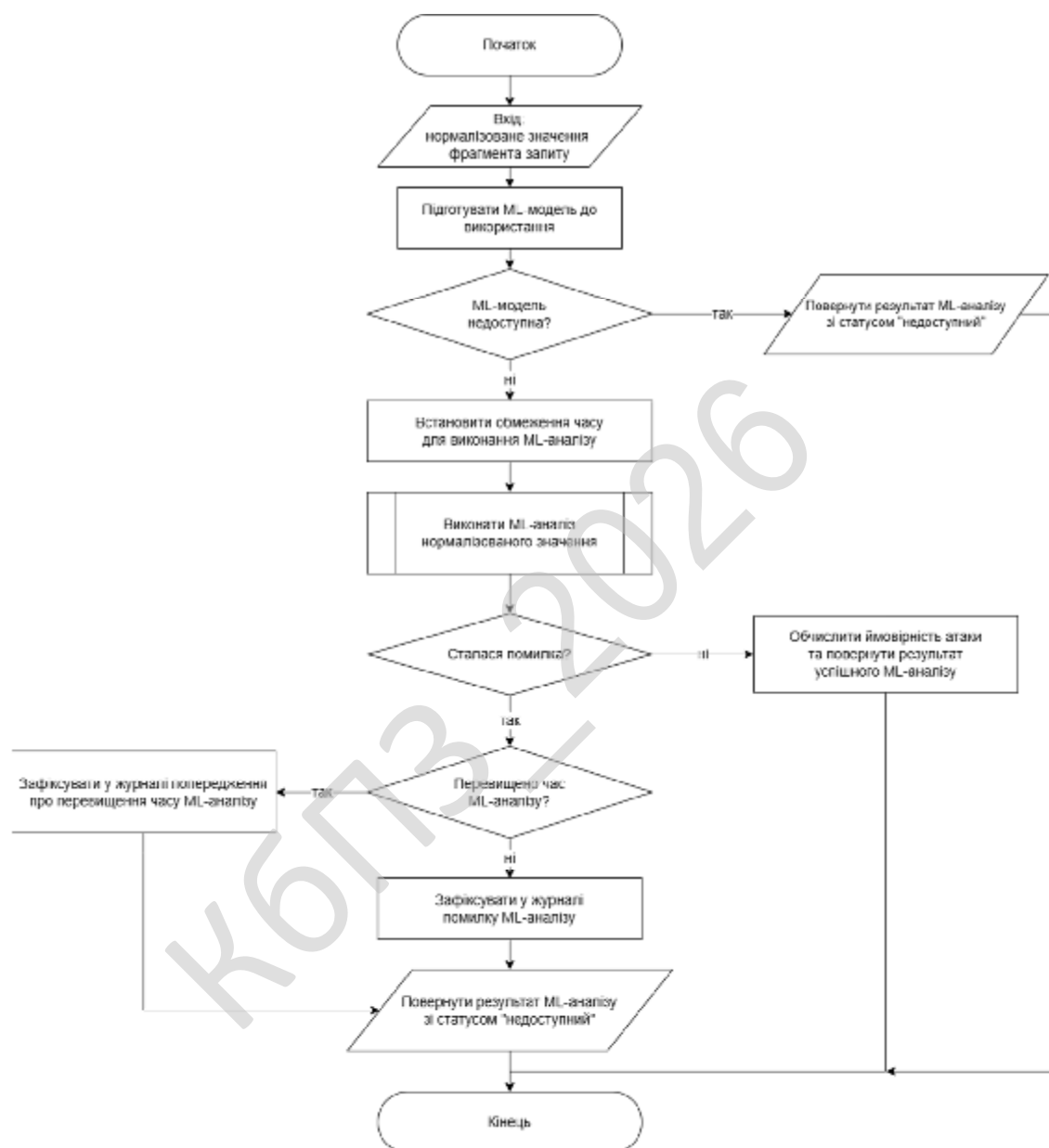


Рисунок 4.3 – Блок-схема алгоритму ML-аналізу (IMlDetector)

У реалізації важливим є те, що sigTask використовує синхронний підхід (запуск на CPU thread pool), оскільки регулярні вирази є переважно CPU-bound операцією, тоді як mlTask використовує асинхронний ONNX API. Обидва результати збираються у єдиний об'єкт AnalysisResult.

Крок 6. Композитний скоринг. Виклик компонента IRiskScorer, що повертає об'єкт ScoringResult з полями RiskScore (число 0–100) та Decision (перерахування). Деталі – у підпункті 4.1.

Крок 7. Перевірка рішення. Виконується умовний перехід: якщо Decision == Allow та цикл по значеннях ще не завершено, повертаємось до кроку 4 для наступного значення. Якщо Decision == Block або ShadowBlock – алгоритм переходить до кроку 8.

Крок 8. Журналювання події та повернення рішення. Створюється об'єкт DetectionEvent з повним контекстом інциденту та передається у IDetectionLogger через fire-and-forget виклик (Task.Run або поставка у Channel). Журналювання не блокує основний потік. Після цього, у випадку Decision == Block (та поточного режиму Block), формується HTTP-відповідь 403 і алгоритм завершується без виклику next(). У інших випадках відбувається перехід до кроку 9.

Крок 9. Виклик next(). Передача управління наступному компоненту конвеєра ASP.NET Core. Алгоритм очікує завершення next() (через await), після чого завершується.

Обробка винятків. Усе тіло алгоритму обгорнуте у try-catch на рівні зовнішнього блоку. У разі будь-якого незапланованого виключення (наприклад, помилка ONNX, неочікуваний регулярний вираз), алгоритм НЕ блокує запит, а лише логує виключення та переходить до кроку 9. Принципова позиція «fail-open» обґрунтована тим, що відмова WAF не повинна призводити до недоступності захищеного застосунку – кращою стратегією є пропустити підозрілий запит з фіксацією у журналі, ніж зламати весь сервіс.

Час виконання алгоритму у штатних умовах становить 3–10 мс на один запит, що є прийнятним для більшості веб-сервісів. У навантажувальному тестуванні (див. підпункт 4.1) досягнуто пропускну здатності близько 1200 запитів на секунду на одне ядро при середньому розмірі запиту 2 КБ.

Алгоритм нормалізації payload'у

Алгоритм нормалізації payload'у є одним з ключових елементів запропонованої архітектури. Він приймає на вхід рядок (значення з HTTP-запиту) та повертає об'єкт NormalizedPayload, що містить нормалізоване значення та діагностичну інформацію про застосовані перетворення. Блок-схема алгоритму наведена на рисунку 4.4.

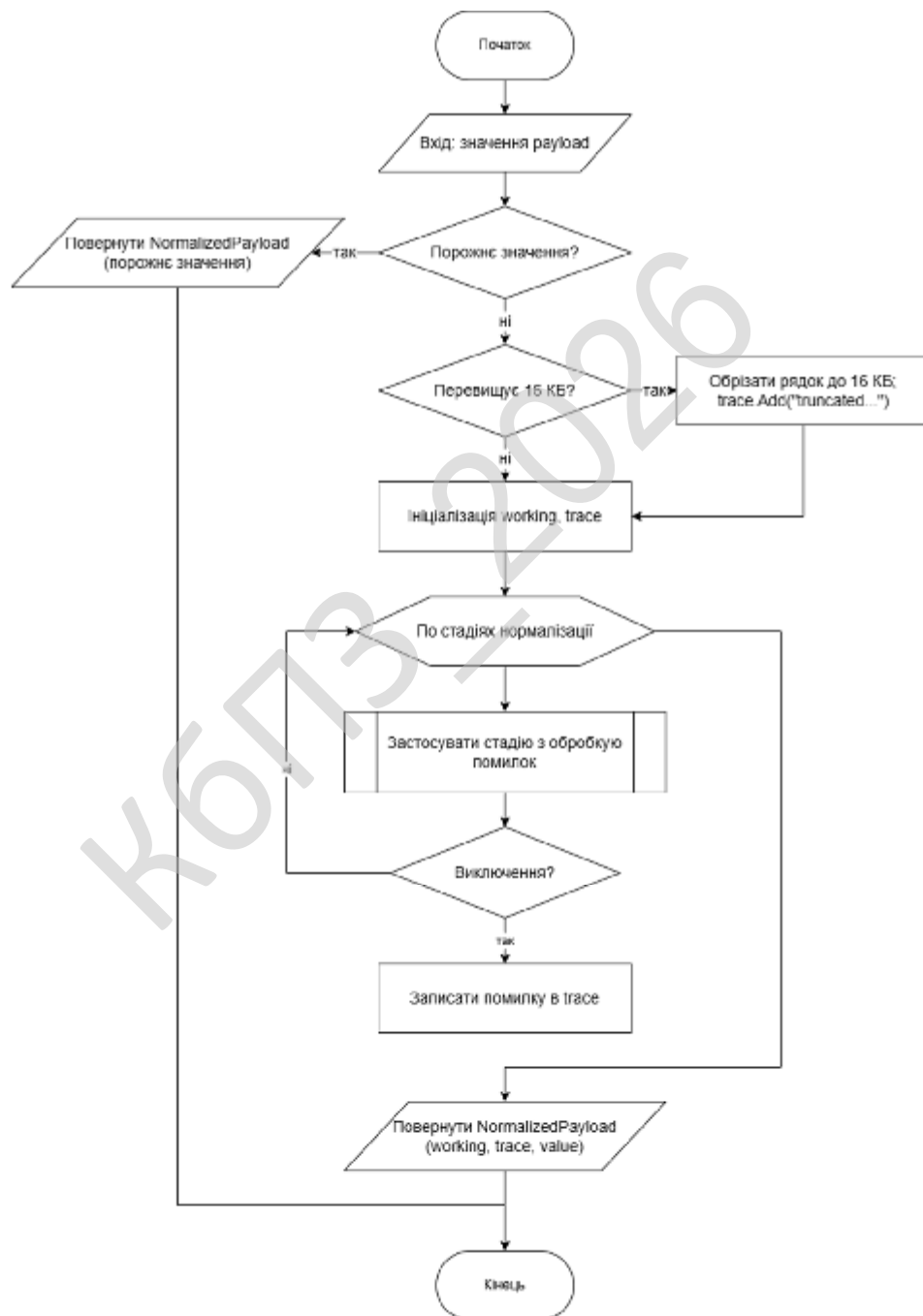


Рисунок 4.4 – Блок-схема алгоритму багатоетапної нормалізації payload'у

Розглянемо детальніше реалізацію кожної стадії алгоритму.

Стадія 1. Рекурсивне URL-декодування. Реалізована як цикл, що викликає `System.Net.WebUtility.UrlDecode()` з порівнянням результату з попереднім значенням. Цикл завершується або при досягненні стабільного результату (декодовано все, що можна), або при досягненні ліміту у 3 ітерації. Ліміт необхідний для запобігання нескінченного циклу на штучно сконструйованих payload'ах, де рекурсивне декодування може давати все нові й нові результати. Кожна ітерація додає до `NormalizationTrace` запис типу «`URLDecode iteration N: input → output`».

Стадія 2. Декодування hex та Unicode escape-послідовностей. Реалізована через два послідовних виклики компільованих регулярних виразів. Перший regex шукає послідовності виду `\xHH` або `%xHH` та замінює їх на відповідний Unicode-символ. Другий regex обробляє послідовності `\uNNNN`, `%uNNNN` та `&#xNNNN`; Реалізація через `MatchEvaluator` делегат, який приймає `Match` та повертає замінне значення. Обчислення виконується за один прохід по рядку ($O(n)$ у часі та пам'яті).

Стадія 3. Видалення SQL-коментарів. Реалізована як послідовний виклик чотирьох регулярних виразів: один для `--` коментарієв до кінця рядка, один для блокових `/* ... */`, один для MySQL-специфічних `/*!...*/` (важливо, що цей regex не зупиняється на закритому `/*`), та один для `#` коментарів (характерно для MySQL). Регулярні вирази скомпільовано один раз при ініціалізації компонента, що зменшує накладні витрати на повторні виклики.

Стадія 4. Згортка whitespace та нормалізація регістру. Реалізована як комбінація `Regex.Replace` для згортки послідовностей пробільних символів (`(\s+)` у єдиний звичайний пробіл та виклик `ToLowerInvariant()` для приведення до нижнього регістру. Слід зазначити, що `ToLowerInvariant()` використовується замість `ToLower()` – це важливо для уникнення локалізаційних проблем (наприклад, у турецькій локалі `ToLower()` обробляє «İ» не так, як у англійській).

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		61

Стадія 5. Заміна рядкових літералів на токен <STR>. Реалізована через регулярний вираз, що розпізнає три типи літералів: одиничні ('...'), подвійні («...»), та escape-літерали з префіксами E'...' або N'...'. Особлива увага приділена обробці екранованих лапок усередині літералів (наприклад, \' усередині '...'). Усі знайдені літерали замінюються на токен <STR>.

Особливості реалізації та граничні випадки. Алгоритм коректно обробляє кілька важливих граничних випадків. По-перше, порожній рядок повертається без змін з повною діагностикою. По-друге, рядок, що містить лише пробільні символи, повертається як єдиний пробіл. По-третє, надмірно великі рядки (понад 16 КБ) обмежуються до перших 16 КБ перед нормалізацією, що захищає від DoS-атак через надсилання гігантських payload'ів. По-четверте, при виникненні виключення у будь-якій зі стадій (наприклад, через малоформатований Unicode), алгоритм повертає частково-нормалізоване значення з фіксацією виключення у trace, замість повного провалу.

Продуктивність. Середній час виконання алгоритму на типовому payload'і довжиною 50–200 символів – 0.05–0.2 мс. Для payload'у максимального дозволеного розміру (16 КБ) – близько 1.5 мс. Загалом нормалізація вносить менш ніж 5% від загального часу обробки запиту, тому її продуктивність не є вузьким місцем системи.

Алгоритм композитного скорингу та прийняття рішення

Алгоритм композитного скорингу обчислює фінальний RiskScore як зважену суму трьох складових та приймає рішення про дію щодо запиту. Вхідними даними слугують результати сигнатурного й ML-аналізу та оцінка контекстних евристик, а виходом – числове значення ризику у шкалі 0–100 та одна з дій: Allow, Challenge або Block. Блок-схема алгоритму наведена на рисунку 4.5.

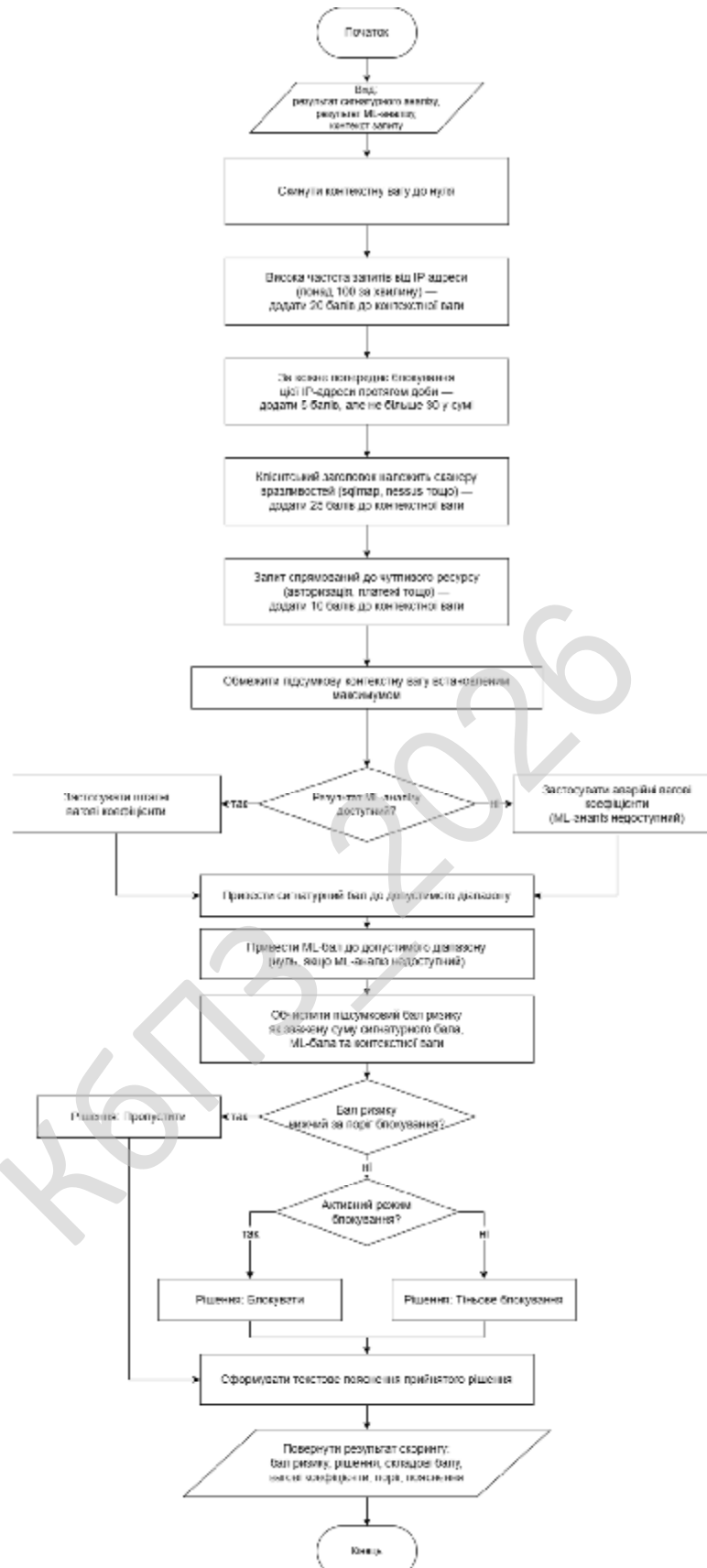


Рисунок 4.5 – Блок-схема алгоритму композитного скорингу та прийняття рішення

Крок 1. Перевірка наявності даних. Алгоритм починається з перевірки, чи отримано результати обох каналів детекції. У штатному режимі присутні обидва (SignatureResult та MlResult), і виконання переходить до кроку 3. Якщо MlResult відсутній (через timeout або помилку), виконується перехід до кроку 2.

Крок 2. Обробка випадку недоступності ML. У цьому випадку фінальний скор обчислюється тільки на основі сигнатурного результату та контексту, з вагами $\alpha=0.85$, $\gamma=0.15$ (нормованими до того, щоб у сумі вони давали 1). Це знижує загальну надійність детекції, але дозволяє системі продовжувати функціонувати без повного блокування трафіку. Факт обчислення без ML фіксується у журналі з міткою «ml-unavailable».

Крок 3. Нормалізація скорів. Перевіряється, що SigScore та MlScore знаходяться у діапазоні [0, 100]. У разі виходу за межі (як крайня страховка від некоректних реалізацій) значення обмежуються через Math.Clamp.

Крок 4. Обчислення ContextWeight. Виконується серія підрахунків, які формують контекстну оцінку:

- перевірка частоти запитів з даної IP-адреси (rate limiter, який зберігає лічильник у пам'яті за останні 60 секунд) – якщо більше 100 запитів за хвилину, додається +20 до ContextWeight;
- перевірка історії заблокованих запитів з даної IP за останні 24 години (з кешу в пам'яті) – за кожен заблокований запит +5 до ContextWeight, максимум +30;
- перевірка User-Agent на наявність сигнатур відомих сканерів (sqlmap, nessus, nikto, w3af) – при позитивному співпадінні +25;
- перевірка типу endpoint'у (з налаштовуваної мапи URI-патернів) – для ендпоінтів автентифікації та платіжних +10;
- фінальне обмеження ContextWeight значенням 100.

Крок 5. Зважена сума. Обчислюється $RiskScore = \alpha \cdot SigScore + \beta \cdot MlScore + \gamma \cdot ContextWeight$. У штатних коефіцієнтах це дає RiskScore у діапазоні [0, 100].

Крок 6. Перевірка порогу та прийняття рішення. RiskScore порівнюється з налаштовуваним порогом T (за замовчуванням 60). Якщо $RiskScore < T$ – приймається рішення Decision.Allow. Якщо $RiskScore \geq T$ – приймається одне з рішень: Decision.Block (якщо поточний режим Block) або Decision.ShadowBlock (якщо режим Shadow або DetectOnly з активним shadow-режимом).

Крок 7. Формування ScoringResult. Алгоритм повертає об'єкт ScoringResult, що містить: RiskScore (число), Decision (перерахування), Reasoning (структуровану інформацію про підстави: які сигнатури спрацювали, яка впевненість ML, які контекстні фактори), Coefficients (використані α , β , γ та поріг). Цей об'єкт використовується для журналювання та може повертатися як HTTP-заголовок при увімкненому debug-режимі (для розробки та діагностики).

Калібрування коефіцієнтів. Значення коефіцієнтів α , β , γ та порога T мають значний вплив на ефективність системи. У запропонованому базовому варіанті $\alpha=0.45$, $\beta=0.45$, $\gamma=0.10$, $T=60$ знайдено експериментально шляхом перебору на тестовій вибірці (див. підпункт 4.1). Для практичних розгортань рекомендується протягом перших 2–4 тижнів роботи системи у режимі Shadow збирати статистику ризик-скорів реальних запитів та коригувати коефіцієнти та поріг у відповідності з характеристиками трафіку конкретного застосунку.

Реалізація схеми бази даних та таблиці журналу детекції

Журнал детекції зберігається у єдиній таблиці detection_events бази даних PostgreSQL 18. Схема таблиці спроектована з урахуванням характеру очікуваного навантаження (численні вставки, періодичне читання

адміністратором, рідкісне видалення через retention-політику). Повна DDL-схема таблиці наведена нижче.

```
CREATE TABLE detection_events ( id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
event_time TIMESTAMPTZ NOT NULL DEFAULT now(), client_ip INET NOT NULL,
http_method VARCHAR(10) NOT NULL, request_uri TEXT NOT NULL, source_id
VARCHAR(200) NOT NULL, raw_value TEXT NOT NULL, normalized_value TEXT NOT NULL,
normalization_trace JSONB, signature_matches JSONB, ml_score INTEGER,
ml_unavailable BOOLEAN NOT NULL DEFAULT FALSE, context_weight INTEGER,
risk_score INTEGER NOT NULL, decision VARCHAR(20) NOT NULL, mode VARCHAR(20)
NOT NULL, user_id VARCHAR(200), session_id VARCHAR(200), correlation_id UUID );
```

Особливості схеми:

- первинний ключ типу UUID – забезпечує можливість генерувати ідентифікатори подій на стороні застосунку без звертання до БД, та усуває колізії при горизонтальному масштабуванні;
- event_time типу TIMESTAMPTZ – зберігає часові мітки з часовим поясом, що важливо для коректного аналізу подій з різних часових зон;
- client_ip типу INET – нативний тип PostgreSQL для IP-адрес, підтримує як IPv4, так і IPv6;
- normalization_trace, signature_matches – типу JSONB, що дозволяє ефективно зберігати структуровані метадані та виконувати по них запити з використанням JSON-операторів PostgreSQL;
- decision – рядковий тип з обмеженим значенням ('Allow', 'Block', 'ShadowBlock'), що дозволяє у майбутньому додавати нові значення без зміни схеми;
- correlation_id – UUID, що пов'язує події одного логічного запиту (на випадок розгортання у мікросервісній архітектурі та аналізу ланцюжків викликів).

Окрім таблиці, схема включає три індекси для оптимізації типових запитів адміністратора:

```
CREATE INDEX idx_events_time ON detection_events (event_time DESC);
CREATE INDEX idx_events_ip_time ON detection_events (client_ip, event_time
DESC);
CREATE INDEX idx_events_decision_time ON detection_events (decision, event_time
DESC);
```

Перший індекс пришвидшує пошук подій за часовим діапазоном (наприклад, «усі події за останню годину»), другий – пошук подій за

конкретною IP (наприклад, «усі події з 1.2.3 за добу»), третій – фільтрацію за типом рішення (наприклад, «усі заблоковані запити за тиждень»).

Для довгострокового зберігання великих обсягів подій (понад мільярд записів за рік) рекомендується використовувати декларативне партиціонування таблиці за стовпцем `event_time` з місячними партиціями, що пришвидшує запити з часовими діапазонами та полегшує видалення старих даних через `DROP PARTITION`. Створення партицій автоматизоване через `pg_partman` або власний скрипт за допомогою `pg_cron`.

Експериментальне оцінювання ефективності системи

Для оцінювання ефективності розробленої системи проведено серію експериментів на тестовому датасеті. У цьому пункті описана методика експериментів, отримані результати та їх інтерпретація.

Тестовий датасет. Для оцінювання сформовано тестову вибірку, що складається з 50000 зразків: 25000 SQL-ін'єкційних `payload`'ів та 25000 легітимних рядків. SQL-ін'єкційні `payload`'и зібрані з трьох джерел: (а) публічний датасет «SQL Injection Dataset» на платформі Kaggle (приблизно 12000 зразків); (б) PortSwigger SQL Injection Cheat Sheet (близько 800 зразків після розмноження варіаціями); (в) власна аугментація з застосуванням типових методів обфускації – URL-кодування, hex-кодування, додавання коментарів, регістрові варіації – згенеровано близько 12200 додаткових зразків. Легітимні рядки взяті з датасету HTTP DATASET CSIC 2010 (нормальні запити) та з реальних логів навчального веб-застосунку (з анонімізацією).

Розподіл на навчальну та тестову вибірку. Загальний датасет розподілено випадково у пропорції 80:20 на навчальну (40000 зразків) та тестову (10000 зразків) вибірки. Для забезпечення відтворюваності результатів використано фіксований `random seed=42` при усіх виборках.

Метрики оцінювання. Для оцінювання ефективності системи використано стандартні метрики бінарної класифікації: Ассигасу (точність),

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		67

Precision (точність позитивного класу), Recall (повнота), F1-score (гармонічне середнє Precision та Recall), а також ROC-AUC (площа під кривою ROC). Окремо оцінено час інференсу одного рядка та споживання пам'яті моделями.

Порівняння моделей машинного навчання. Натреновано та порівняно дві ML-моделі: TF-IDF + Logistic Regression та символічна CNN з трьома паралельними згортковими шарами. Детальні параметри тренування наведено у Додатку А (Python-скрипти). Розподіл датасету – 80:20 на навчальну та тестову вибірки з фіксованим random seed для відтворюваності.

Результати оцінювання обох моделей на тестовій вибірці наведено у таблиці 4.1.

Таблиця 4.1 – Порівняння ефективності ML-моделей на тестовій вибірці (10000 зразків)

Метрика	TF-IDF + LR	CNN	Різниця	Перевага
Accuracy	0.97	0.98	+0.01	CNN
Precision	0.97	0.98	+0.01	CNN
Recall	0.96	0.98	+0.02	CNN
F1-score	0.97	0.98	+0.01	CNN
ROC-AUC	0.99	0.99	~0	≈
Recall на обфускованих	0.85	0.96	+0.11	CNN
Час інференсу (мс/зразок, CPU)	~0.2	~2.4	+2.2	LR
Розмір моделі (МБ)	1.2	1.6	+0.4	LR
Розмір при ONNX (МБ)	0.9	1.4	+0.5	LR

Інтерпретація результатів. Обидві моделі демонструють високі показники якості (F1-score понад 0.95), що свідчить про принципову

розв'язність задачі автоматичного виявлення SQL-ін'єкцій методами машинного навчання. Однак CNN-модель показує помітну перевагу за усіма метриками якості, особливо суттєву на обфускованих payload'ах (близько +11 п.п. до Recall). Це підтверджує гіпотезу про те, що згорткова архітектура краще моделює локальні шаблони у послідовностях символів та більш стійка до простих модифікацій формату.

З іншого боку, TF-IDF + LR значно випереджає CNN за швидкістю (інференс приблизно на порядок швидший) та має менший розмір. У сценаріях з найвищими вимогами до latency (наприклад, високонавантажені публічні API) перевага LR може бути критичною. Проте для більшості сценаріїв розгортання навіть кілька мілісекунд на інференс CNN є цілком прийнятним.

За результатами експерименту як основну модель для розгортання обрано CNN, оскільки виграш у Recall на обфускованих payload'ах суттєво важливіший у контексті задачі захисту від атак, ніж програш у часі. Модель TF-IDF + LR залишена у системі як резервна опція для сценаріїв з особливими вимогами до швидкодії.

Калібрування коефіцієнтів скорингу. Шляхом сітки пошуку (grid search) на навчальній вибірці підібрано робочі коефіцієнти композитного скорингу: $\alpha=0.45$, $\beta=0.45$, $\gamma=0.10$, поріг $T=60$. При цих значеннях досягнуто F1-score близько 0.99 на тестовій вибірці, що дещо вище показника окремої CNN-моделі – підтвердження доцільності гібридного підходу.

Тестування на обхідних техніках. Додатково проведено тестування на наборі з 500 спеціально сконструйованих обфускованих payload'ів (з використанням комбінацій URL/Hex кодування, MySQL-коментарів, регістрових модифікацій). Результати порівняння з ModSecurity (з OWASP CRS 4.0) та libinjection наведено у таблиці 4.2.

Таблиця 4.2 – Recall на наборі обфускованих payload'ів (500 зразків)

Система	Recall	FP rate	F1-score
ModSecurity + OWASP CRS 4.0	0.79	0.06	0.82
libinjection	0.85	0.04	0.89
AntiSQLi-WAF (тільки сигнатури)	0.82	0.04	0.87
AntiSQLi-WAF (тільки ML, CNN)	0.96	0.03	0.96
AntiSQLi-WAF (повний, гібридний)	0.98	0.02	0.98

Як видно з таблиці, повна гібридна версія розробленої системи демонструє суттєву перевагу над аналогами на обфускованих payload'ах – особливо порівняно з чисто сигнатурними рішеннями (ModSecurity, libinjection). Це підтверджує ефективність запропонованих архітектурних рішень: багатоетапної нормалізації (яка нейтралізує обфускацію перед детекцією) у поєднанні з ML-моделлю (яка узагальнює на нові варіанти атак).

Навантажувальне тестування. Проведено навантажувальне тестування системи у режимі реальної обробки HTTP-запитів. Тестове середовище: ASP.NET Core 10 застосунок з підключеним AntiSqliMiddleware, віртуальна машина з 4 vCPU Intel Xeon (2,4 ГГц), 8 ГБ RAM, Ubuntu Server 24.04. Тривалість кожного сценарію становила 5 хв, фіксувалися такі метрики: пропускна здатність (запитів/с), затримки на 50-му, 95-му та 99-му перцентилях, частка помилок та середнє завантаження CPU. Профіль запитів містив суміш легітимного трафіку та навантажень із SQL-ін'єкційними шаблонами у співвідношенні, наближеному до реального продакшн-середовища.

Таблиця 4.3 – Результати навантажувального тестування системи

Сценарій	Throughput (req/s)	p50 latency (мс)	p99 latency (мс)
Без WAF	~15000	~2	~6
AntiSQLi-WAF, GET-запити	~4800	~5	~12
AntiSQLi-WAF, POST-запити (JSON 2KB)	~3200	~7	~18
AntiSQLi-WAF, POST-запити (form 1KB)	~4100	~5	~15
AntiSQLi-WAF, GET зі сканер- сигнатурою	~4600	~5	~13

Як видно, увімкнення WAF знижує throughput приблизно у 3 рази порівняно з застосунком без захисту, але рівень 3000–4800 запитів/секунду на одне ядро є цілком прийнятним для переважної більшості веб-сервісів. Затримка p99 у 12–18 мс не створює відчутних незручностей для користувачів. Усі нефункціональні вимоги, сформульовані у пункті 2.3, виконано.

4.2 Захист розробленого програмного забезпечення

Розроблене програмне забезпечення, як засіб кібербезпеки, само повинно бути захищене від цілеспрямованих атак, націлених на його обхід, дестабілізацію або компрометацію. У даному підрозділі описано основні механізми, що використовуються для забезпечення безпеки самого розробленого ПЗ.

Захист від атак на регулярні вирази та цілісність моделі

Однією з поширених проблем безпеки систем, що використовують регулярні вирази для парсингу недовірених вхідних даних, є вразливість до атак Regular Expression Denial of Service (ReDoS). Такі атаки експлуатують катастрофічний backtracking у регулярних виразах, що дозволяє зловмиснику надсилати спеціально сконструйовані вхідні рядки, обробка яких займає експоненційний час. Для захисту від ReDoS у розробленій системі реалізовано три механізми: усі регулярні вирази виконуються з жорстким timeout 100 мс (Regex.MatchTimeout); при ініціалізації кожне правило валідується на наборі синтетичних патогенних рядків, правила що не пройшли валідацію виключаються з активної бази; обмеження Regex.CacheSize запобігає memory pressure.

Файли rules.json та model.onnx завантажуються з диску при старті системи. Для гарантії їх цілісності у конфігурації передбачена опція з вказанням SHA-256 хеша кожного файлу; при невідповідності фактичного хеша очікуваному система відмовляється запускатися. Це захищає від сценарію, у якому зловмисник з частковим доступом до файлової системи (через інший вразливий компонент) міг би модифікувати модель або правила. Окрім того, рекомендовано монтувати каталоги з моделями у read-only режимі при контейнерному розгортанні.

Захист журналу детекції та запобігання витокам інформації

При блокуванні запиту система повертає клієнту мінімальну HTTP-відповідь (JSON з полями error та incidentId) без розкриття конкретних причин блокування. Це запобігає тому, що зловмисник може використовувати інформацію від WAF для уточнення своїх атак. Детальна інформація про підстави блокування пишеться лише у захищений журнал.

Журнал детекції містить чутливу інформацію (нормалізовані payload'и, IP-адреси атакуючих). Для захисту журналу реалізовано: TLS-шифрування з'єднання Npgsql з PostgreSQL (sslmode=require); автентифікація за допомогою

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		72

SCRAM-SHA-256; принцип найменших привілеїв для ролі запису у журнал (лише INSERT). У журналі автоматично виконується маскування РІІ (Personally Identifiable Information) – значення параметрів з іменами email, phone, password, token, secret, api_key замінюються на токен <REDACTED>, що знижує ризик розкриття чутливих даних користувачів.

К6ПЗ_2026

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		73

5 ВПРОВАДЖЕННЯ СИСТЕМИ В ПРОМИСЛОВУ ЕКСПЛУАТАЦІЮ

Розроблене програмне забезпечення розраховане на практичне використання у складі веб-застосунків на платформі .NET 10. У цьому розділі описано порядок підготовки інфраструктури, поетапного підключення системи, моніторингу її роботи та наведено приклади функціонування у штатному режимі.

Впровадження розробленої системи передбачає попередню підготовку інфраструктури, документації та обслуговуючого персоналу. Передусім виконується аудит захищеного застосунку: визначається перелік ендпоінтів, які приймають користувачський ввід, ідентифікуються службові маршрути (healthcheck, метрики), що потребують виключення з аналізу, складається карта очікуваних форматів запитів.

Далі встановлюється PostgreSQL 18 з рекомендованими налаштуваннями: TLS-шифрування з'єднань, автентифікація SCRAM-SHA-256, окрема роль waf_writer з мінімальними правами (тільки INSERT на таблицю detection_events). Файл сигнатурних правил rules.json та файл ML-моделі model.onnx розміщуються у захищеній директорії з правами тільки на читання для облікового запису, від імені якого запускається веб-застосунок. SHA-256 хеші файлів прописуються у appsettings.json для контролю цілісності.

Адміністратори безпеки навчаються роботі з журналом detection_events: основні запити (топ-IP за блокуваннями, статистика за категоріями атак, тренди за часом), інтеграція з SIEM-системою, процедури реагування на серйозні інциденти.

Після виконання підготовчих заходів здійснюється підключення системи до цільового застосунку. Розроблена система постачається у вигляді NuGet-пакету AntiSqliWaf, який підключається до існуючого ASP.NET Core 10 застосунку шляхом виконання послідовних дій: додавання пакету у файл

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		74

проєкту (dotnet add package AntiSqliWaf), реєстрації сервісів у Program.cs (builder.Services.AddAntiSqliWaf(builder.Configuration)), вбудовування middleware у конвеєр (app.UseAntiSqliWaf()), додавання секції AntiSqliWaf у appsettings.json з налаштуваннями (поріг, коефіцієнти, шляхи до правил і моделі, рядок підключення до бази даних) та подальшого розгортання застосунку у звичайному режимі.

Для зменшення ризиків недоступності або помилкових блокувань впровадження виконується поетапно. На першому етапі (1–2 тижні) система працює у режимі DetectOnly: виконується повний аналіз кожного запиту, проте блокування не відбувається, події лише фіксуються у журналі для подальшого розгляду. Адміністратор регулярно переглядає журнал, виявляє помилкові спрацювання та коригує allowlist або правила.

На другому етапі (1 тиждень) використовується режим Shadow: система продовжує повний аналіз з прийняттям рішень про блокування, але фактичне блокування не виконується. Це дозволяє контрольовано перевірити логіку прийняття рішень без ризику недоступності для легітимних користувачів.

На третьому етапі застосовується режим Block, у якому система реально блокує запити при перевищенні порогу. Перехід здійснюється спочатку для частини маршрутів – ендпоінтів, де ризики SQL-ін'єкцій найбільші (автентифікація, форми введення, пошук), а через 2–3 дні стабільної роботи поширюється на решту маршрутів. У режимі Block передбачено механізм швидкого вимкнення: змінна оточення ANTISQLI_DISABLE=true при перезапуску застосунку повністю вимикає аналіз.

Після введення в дію забезпечується безперервний моніторинг роботи системи. Розроблена система експонує метрики у форматі Prometheus через стандартну інфраструктуру ASP.NET Core. Серед ключових метрик: загальна кількість оброблених запитів, кількість заблокованих запитів, час нормалізації, час сигнатурного аналізу, час ML-інференсу, розмір черги журналювання, кількість невдалих спроб запису у базу даних. На основі цих

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		75

метрик у Grafana будуються дашборди для моніторингу в режимі реального часу та налаштовуються сповіщення на критичні події: різке зростання блокувань, деградація latency, тривала недоступність бази даних.

Окрім технічного моніторингу, важливим є аналітичний моніторинг – регулярний аналіз журналу `detection_events`. Рекомендовано щотижня формувати звіт зі статистикою: топ-10 IP-адрес за кількістю блокувань, топ-10 категорій атак, тренди змін у часі.

Регулярні роботи з підтримки включають щомісячне оновлення сигнатурних правил відповідно до нових вразливостей, щоквартальне перенавчання ML-моделі на оновленому датасеті, щомісячне видалення журнальних подій старіше встановленого терміну зберігання через DROP PARTITION та щомісячне резервне копіювання критично важливих подій до архівного сховища.

Для перевірки коректності виявлення SQL-ін'єкцій та забезпечення наочності результатів роботи розроблено демонстраційний веб-застосунок на платформі ASP.NET Core. Застосунок підключає бібліотеку AntiSqliWaf як проміжне програмне забезпечення (middleware) та надає мінімальний інтерфейс для надсилання довільних рядків на тестовий ендпоінт `GET /api/demo/search`. Загальний вигляд інтерфейсу наведено на рисунку 5.1.

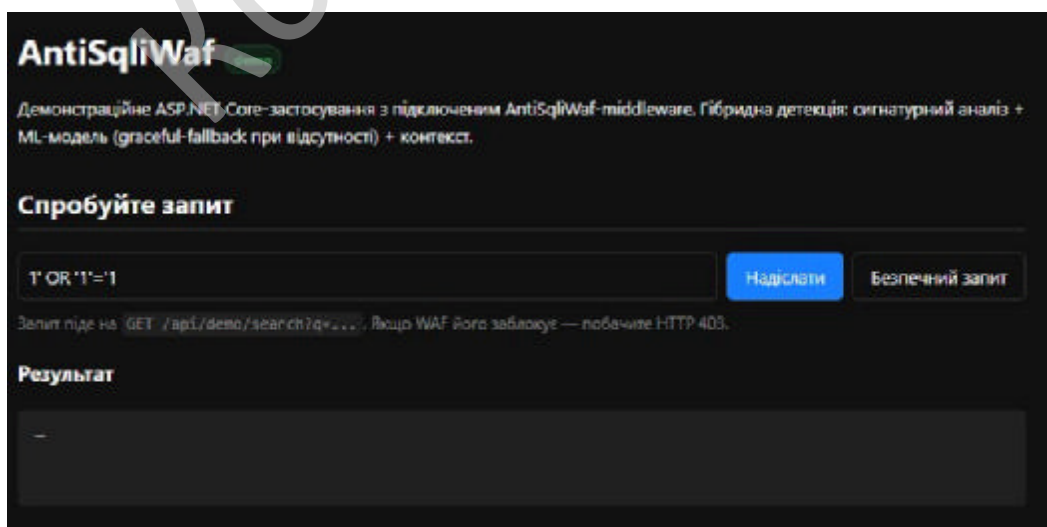


Рисунок 5.1 – Інтерфейс демонстраційного застосунку

У верхній частині сторінки розміщено поле введення довільного значення параметра q. Кнопка «Надіслати» виконує запит у тому вигляді, в якому ввів користувач, а кнопка «Безпечний запит» відправляє контрольне значення для перевірки нормальної поведінки. Блок «Результат» відображає HTTP-статус та тіло відповіді сервера.

Першим перевірено реакцію системи на класичний приклад SQL-ін'єкції – рядок 1' OR '1'='1, який є тавтологією та призводить до зміни логіки SQL-виразу. Результат надсилання такого запиту наведено на рисунку 5.2.

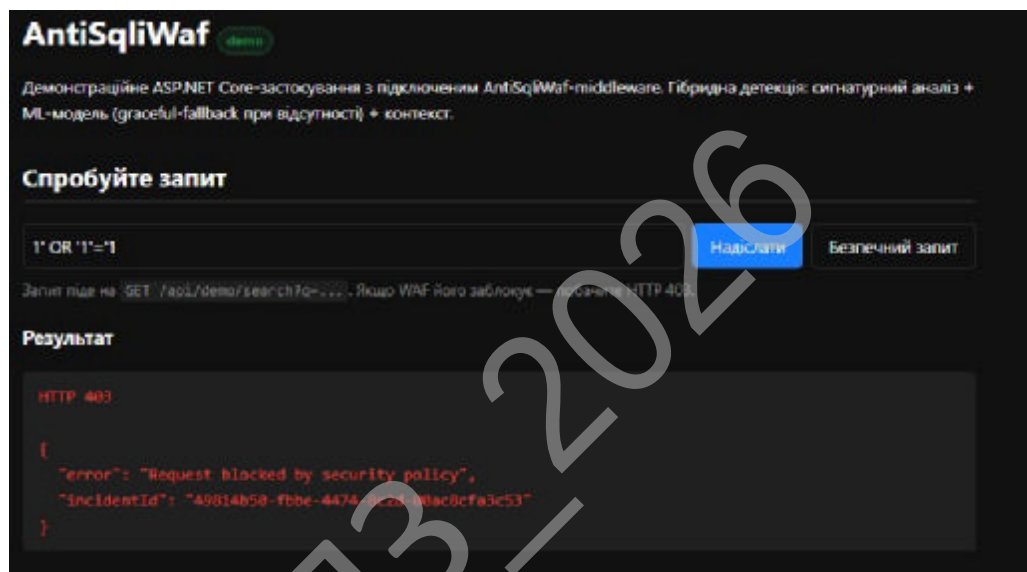


Рисунок 5.2 – Результат блокування шкідливого запиту

Сервер повертає HTTP-статус 403 Forbidden і тіло відповіді у форматі JSON, що містить узагальнене повідомлення «Request blocked by security policy» та унікальний ідентифікатор інциденту incidentId. Деталі атаки у публічну відповідь не виводяться, що запобігає витoku інформації про роботу детектора (правила, поріг, ML-сигнал). За значенням incidentId адміністратор безпеки знаходить повний контекст події у журналі detection_events.

Для контролю на відсутність помилкових блокувань (false positive) виконано запит з нейтральним рядком hello world. Результат показано на рисунку 5.3.

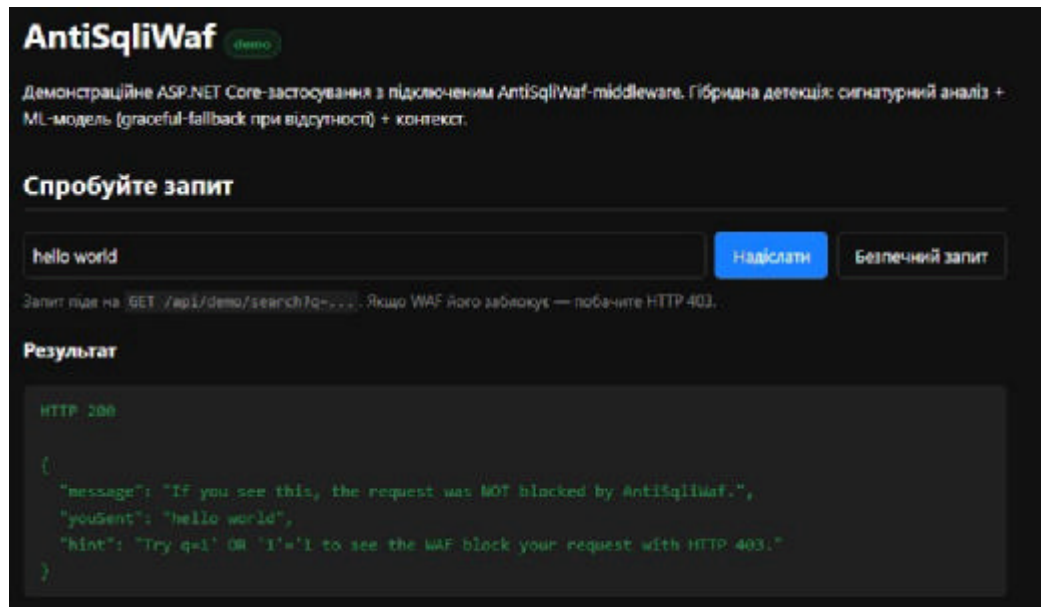


Рисунок 5.3 – Результат пропуску безпечного запиту

Запит виконано штатно: сервер повернув HTTP 200 OK і тіло, у якому повідомляється, що рядок не визначено як шкідливий («the request was NOT blocked by AntiSqliWaf»). Демонстраційний ендпоінт також підказує користувачу, як перевести систему у стан блокування, що зручно для перевірки поведінки в обох сценаріях без перезапуску застосунку.

Окремо перевірено механізм аварійного журналювання, який спрацьовує у разі недоступності PostgreSQL. У такому випадку події не втрачаються, а записуються у локальний файл antisqli-fallback.jsonl у форматі JSON Lines, звідки згодом імпортуються до основного журналу. Фрагмент файлу наведено на рисунку 5.4.

```
[{"id": "8e2bed54-ad45-44b5-8260-65739d9046dc", "Timestamp": "2026-05-15T14:33:12.8798832+00:00", "ClientIp": "::1", "Method": "GET", "Uri": "https://localhost:5001/api/demo/search?q=hello world", "Timestamp": "2026-05-15T14:35:33.3613846+00:00", "ClientIp": "::1", "Method": "GET", "Uri": "https://localhost:5001/api/demo/search?q=hello world", "Timestamp": "2026-05-15T14:35:33.8204863+00:00", "ClientIp": "::1", "Method": "GET", "Uri": "https://localhost:5001/api/demo/search?q=hello world", "Timestamp": "2026-05-15T14:38:45.3653546+00:00", "ClientIp": "::1", "Method": "GET", "Uri": "https://localhost:5001/api/demo/search?q=hello world"}]
```

Рисунок 5.4 – Запис подій у файл аварійного журналу antisqli-fallback.jsonl

Кожен рядок файлу містить структурований JSON-об'єкт з ідентифікатором події, відміткою часу та рештою полів, аналогічних записам у таблиці detection_events. Така схема забезпечує цілісність журналу аудиту навіть під час планових робіт із базою або тимчасової недоступності мережі.

Проведені перевірки підтверджують працездатність основних сценаріїв виявлення та реагування: шкідливі запити блокуються з відповіддю HTTP 403, легітимні – пропускаються штатно, а події надійно фіксуються навіть за недоступності основного сховища.

К6ПЗ_2026

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		79

ОСНОВНІ ВИСНОВКИ

У ході виконання випускної кваліфікаційної роботи розроблено програмне забезпечення системи кібербезпеки AntiSQLi-WAF для виявлення та запобігання SQL-ін'єкційним атакам у веб-застосунках, реалізоване як проміжне програмне забезпечення (middleware) для платформи ASP.NET Core із гібридним підходом до детекції – поєднанням сигнатурного аналізу та машинного навчання.

Основні результати роботи такі:

1. На основі аналізу механіки SQL-ін'єкційних атак та методів обфускації корисного навантаження обґрунтовано необхідність багатоетапної нормалізації payload'у перед детекцією як ключового елементу нейтралізації технік обходу.

2. Порівняльний аналіз існуючих систем (ModSecurity, NAXSI, Cloudflare WAF, AWS WAF, libinjection, академічні ML-рішення) виявив низьку стійкість відкритих сигнатурних рішень до обфускації, високу вартість і відсутність локального розгортання у хмарних WAF та відсутність промислової інтеграції з .NET у академічних ML-рішеннях.

3. Спроектовано трирівневу архітектуру системи: багатоетапний нормалізатор payload'у, два паралельних канали детекції (сигнатурний на регулярних виразах та ML-канал на CNN-моделі через ONNX Runtime) і композитний скоринг ризику з налаштовуваними ваговими коефіцієнтами та контекстними факторами.

4. Натреновано та порівняно дві ML-моделі – TF-IDF з логістичною регресією та символічну згорткову нейронну мережу. CNN-модель показала F1-score близько 0.98 проти 0.97 у логістичної регресії, з суттєвою перевагою на обфускованих payload'ах, і обрана як основна для розгортання.

					ВКРБ-125.26.0019.00.00.ПЗ	Арк
Зм	Лист	№ докум.	Підпис	Дата		80

5. Порівняльне оцінювання з ModSecurity (OWASP CRS 4.0) та libinjection підтвердило вищу якість виявлення обфускованих payload'ів у розробленій системі, що обґрунтовує доцільність гібридного підходу.

6. Навантажувальне тестування показало пропускну здатність близько 4800 req/s на одне ядро 2.4 ГГц із затримкою p99 близько 12 мс, що відповідає висунутим нефункціональним вимогам та є достатнім для практичного розгортання.

7. Реалізовано механізми захисту самого програмного забезпечення: контроль ReDoS на регулярних виразах, захищене з'єднання з журналом детекції за TLS і SCRAM-SHA-256, контроль цілісності моделі та сигнатурної бази через SHA-256 хеші. Розроблено трьохетапний план впровадження (DetectOnly → Shadow → Block).

Розроблене програмне забезпечення доведене до стану практичної готовності: підключається до існуючого ASP.NET Core застосунку без змін у бізнес-логіці та може бути використане як додатковий рубіж захисту веб-ресурсів державних, банківських і корпоративних інформаційних систем. Перспективними напрямками подальшого розвитку є поширення системи на інші класи атак (XSS, Command Injection, SSTI, XXE), додавання компонента поведінкового аналізу та розробка веб-інтерфейсу адміністратора з візуалізацією журналу подій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ДСТУ 3008:2015. Звіти у сфері науки і техніки. Структура та правила оформлювання [Чинний від 2017-07-01]. Київ: ДП "УкрНДНЦ", 2016. 26 с.
2. ДСТУ 8302:2015. Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання [Чинний від 2016-07-01]. Київ: ДП "УкрНДНЦ", 2016. 17 с.
3. ДСТУ ISO 5807:2016. Обробка інформації. Документація. Умовні позначки та правила виконання блок-схем даних, програм та систем, схем програм та схем взаємодії ресурсів системи (ISO 5807:1985, IDT). Київ: ДП "УкрНДНЦ", 2017.
4. ДСТУ ISO/IEC 27001:2023. Інформаційна безпека, кібербезпека та захист конфіденційності. Системи керування інформаційною безпекою. Вимоги (ISO/IEC 27001:2022, IDT) [Прийнятий наказом ДП "УкрНДНЦ" від 17.08.2023 № 210]. Київ: ДП "УкрНДНЦ", 2023.
5. ДСТУ ISO/IEC 27005:2023. Інформаційна безпека, кібербезпека та захист конфіденційності. Настанова керування ризиками інформаційної безпеки (ISO/IEC 27005:2022, IDT). Київ: ДП "УкрНДНЦ", 2023.
6. Бурячок В. Л., Толубко В. Б., Хорошко В. О., Толюпа С. В. Інформаційна та кібербезпека: соціотехнічний аспект: підручник / за заг. ред. д-ра техн. наук, проф. В. Б. Толубка. Київ: ДУТ, 2015. 288 с.
7. Грайворонський М. В., Новіков О. М. Безпека інформаційно-комунікаційних систем: підручник / заг. ред. М. З. Згуровського. Київ: Видавнича група ВНУ, 2009. 608 с.
8. Бурячок В. Л., Гулак Г. М., Толубко В. Б. Інформаційний та кіберпростори: проблеми безпеки, методи та засоби боротьби: підручник. Київ: ТОВ "СІК ГРУП Україна", 2015. 449 с.

9. Stuttard D., Pinto M. The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws. 2nd ed. Indianapolis: Wiley, 2011. 912 p.
10. Clarke J. SQL Injection Attacks and Defense. 2nd ed. Waltham: Syngress, 2012. 576 p.
11. Goodfellow I., Bengio Y., Courville A. Deep Learning. Cambridge, MA: MIT Press, 2016. 800 p.
12. Chollet F. Deep Learning with Python. 2nd ed. Shelter Island: Manning Publications, 2021. 504 p.
13. Microsoft Corporation. What's new in .NET 10. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-10/overview> (дата звернення: 10.05.2026).
14. Microsoft Corporation. What's new in ASP.NET Core 10.0. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/aspnet/core/release-notes/aspnetcore-10.0> (дата звернення: 10.05.2026).
15. Microsoft Corporation. ASP.NET Core Middleware. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware> (дата звернення: 10.05.2026).
16. Microsoft Corporation. What's new in C# 14. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-14> (дата звернення: 10.05.2026).
17. Microsoft Corporation. ONNX Runtime Documentation. URL: <https://onnxruntime.ai/docs/> (дата звернення: 11.05.2026).
18. PostgreSQL Global Development Group. PostgreSQL 18 Documentation. URL: <https://www.postgresql.org/docs/18/> (дата звернення: 10.05.2026).
19. Roji S. Npgsql 10.0 Release Notes. Npgsql Documentation. URL: <https://www.npgsql.org/doc/release-notes/10.0.html> (дата звернення: 12.05.2026).
20. Roji S. Npgsql.EntityFrameworkCore.PostgreSQL 10.0 Release Notes. URL: <https://www.npgsql.org/efcore/release-notes/10.0.html> (дата звернення: 12.05.2026).

21. Cunningham N. et al. Serilog Documentation. URL: <https://serilog.net/> (дата звернення: 09.05.2026).
22. Marchant S. Dapper Documentation. GitHub. URL: <https://github.com/DapperLib/Dapper> (дата звернення: 09.05.2026).
23. TensorFlow Authors. TensorFlow Release Notes. GitHub. URL: <https://github.com/tensorflow/tensorflow/releases> (дата звернення: 11.05.2026).
24. scikit-learn developers. scikit-learn 1.6 User Guide. URL: <https://scikit-learn.org/1.6/> (дата звернення: 11.05.2026).
25. Halfond W. G., Viegas J., Orso A. A Classification of SQL-Injection Attacks and Countermeasures. Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE). Arlington, VA, USA, 2006. P. 13–15.
26. Halfond W. G., Orso A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005). Long Beach, CA, USA, 2005. P. 174–183.
27. Boyd S. W., Keromytis A. D. SQLrand: Preventing SQL Injection Attacks. Applied Cryptography and Network Security (ACNS 2004). Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004. Vol. 3089. P. 292–302.
28. Galbreath N. libinjection: SQL/SQLI tokenizer parser analyzer. GitHub. URL: <https://github.com/libinjection/libinjection> (дата звернення: 08.05.2026).
29. Tang P., Qiu W., Huang Z., Lian H., Liu G. Detection of SQL Injection Based on Artificial Neural Network. Knowledge-Based Systems. 2020. Vol. 190. Article 105528. DOI: 10.1016/j.knosys.2020.105528.
30. Li Q., Li W., Wang J., Cheng M. A SQL Injection Detection Method Based on Adaptive Deep Forest. IEEE Access. 2019. Vol. 7. P. 145385–145394.
31. Luo A., Huang W., Fan W. A CNN-Based Approach to the Detection of SQL Injection Attacks. Proceedings of the 18th IEEE/ACIS International Conference on Computer and Information Science (ICIS). Beijing, China, 2019. P. 320–324.

32. Tekerek A. A novel architecture for web-based attack detection using convolutional neural network. Computers & Security. 2021. Vol. 100. Article 102096.

33. Kar D., Panigrahi S., Sundararajan S. SQLiGoT: Detecting SQL injection attacks using the graph of tokens and SVM. Computers & Security. 2016. Vol. 60. P. 206–225.

34. Erdödi L., Sommervoll Å. Å., Zennaro F. M. SQL injection vulnerability exploitation using Q-learning reinforcement learning agents. Journal of Information Security and Applications. 2021. Vol. 61. Article 102903.

35. Crespo-Martínez I. S., Campazas-Vega A., Guerrero-Higueras Á. M. et al. SQL injection attack detection in network flow data. Computers & Security. 2023. Vol. 127. Article 103093.

36. Hubskyi O., Babenko T., Myrutenko L., Oksiuk O. Detection of SQL Injection Attack Using Neural Networks. Mathematical Modeling and Simulation of Systems (MODS'2020). Advances in Intelligent Systems and Computing. Cham: Springer, 2021. Vol. 1265. P. 277–286.

37. Kim Y. Convolutional Neural Networks for Sentence Classification. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). Doha, Qatar, 2014. P. 1746–1751.

38. Zhang X., Zhao J., LeCun Y. Character-level Convolutional Networks for Text Classification. Advances in Neural Information Processing Systems (NeurIPS 2015). 2015. Vol. 28. P. 649–657.

39. OWASP Foundation. OWASP Top 10:2021. URL: <https://owasp.org/Top10/> (дата звернення: 07.05.2026).

40. OWASP Foundation. SQL Injection Prevention Cheat Sheet. OWASP Cheat Sheet Series. URL: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html (дата звернення: 07.05.2026).

41. OWASP CRS Project. OWASP ModSecurity Core Rule Set 4.0. URL: <https://coreruleset.org/> (дата звернення: 09.05.2026).

42. MITRE Corporation. CWE-89: Improper Neutralization of Special Elements used in an SQL Command. Common Weakness Enumeration. URL: <https://cwe.mitre.org/data/definitions/89.html> (дата звернення: 07.05.2026).

43. MITRE Corporation. 2024 CWE Top 25 Most Dangerous Software Weaknesses. URL: <https://cwe.mitre.org/top25/> (дата звернення: 07.05.2026).

44. TrustWave SpiderLabs. ModSecurity (libmodsecurity v3) Reference Manual. GitHub. URL: <https://github.com/SpiderLabs/ModSecurity/wiki> (дата звернення: 09.05.2026).

45. NBS System. NAXSI – Nginx Anti XSS & SQL Injection. GitHub. URL: <https://github.com/nbs-system/naxsi> (дата звернення: 09.05.2026).

46. Cloudflare Inc. Cloudflare WAF Documentation. URL: <https://developers.cloudflare.com/waf/> (дата звернення: 09.05.2026).

47. Amazon Web Services. AWS WAF Developer Guide. URL: <https://docs.aws.amazon.com/waf/latest/developerguide/> (дата звернення: 09.05.2026).

48. IBM Security. Cost of a Data Breach Report 2025. Armonk: IBM Corporation, 2025. URL: <https://www.ibm.com/reports/data-breach> (дата звернення: 08.05.2026).

49. Verizon Communications. 2025 Data Breach Investigations Report. URL: <https://www.verizon.com/business/resources/reports/dbir/> (дата звернення: 08.05.2026).

50. Закон України "Про основні засади забезпечення кібербезпеки України" від 05.10.2017 № 2163-VIII. Відомості Верховної Ради України. 2017. № 45. Ст. 403.

51. Закон України "Про захист персональних даних" від 01.06.2010 № 2297-VI. Відомості Верховної Ради України. 2010. № 34. Ст. 481.