УДК 004.056
DOI: 10.18523/2617-3808.2025.8.118-125

Михайленко О. І., Гороховський К. С., Гороховський С. С.

МЕТОД ШИФРОВАНОЇ КОМУНІКАЦІЇ У СТРАТЕГІЧНИХ ВЗАЄМОДІЯХ

У роботі досліджено можливості застосування наскрізного шифрування в середовищах з обмеженою довірою, зокрема в контексті стратегічних ігор та симуляцій. Незважаючи на широке впровадження таких технологій у сфері цифрової комунікації, їхній потенціал у специфічних сценаріях залишався недостатньо вивченим. Запропоновано новий підхід до захищеної взаємодії між учасниками шляхом адаптації криптографічного протоколу Double Ratchet. Розроблено application-level протокол, оптимізований для ігрових сценаріїв, і вперше реалізовано його компіляцію у WebAssembly, досліджено можливості використання протоколів наскрізного шифрування у браузерному середовищі, а також надання безпеки при зберіганні пар криптографічних ключів поза безпечними середовищами апаратного забезпечення, наведено приклад використання розробленого протоколу для забезпечення приватності у стратегічній комунікації. Проведено оцінку ефективності та безпеки рішення в умовах симульованого середовища з недовірою до сервера. Актуальність дослідження зумовлено зростанням потреби у захищеному зв'язку в умовах кібератак, зокрема в період воєнних дій.

Ключові слова: наскрізне шифрування, Double Ratchet, WebAssembly, стратегічні ігри, WebSocket, Rust, захищена комунікація.

Вступ

Існує багато задач, які потребують безпеки у комунікаційному каналі між двома сторонами. Традиційними методами для створення секретності у комунікації є методи симетричного та асиметричного шифрування, однак використання таких систем має певні недоліки, коли йдеться про системи з мінімальною довірою. До них зокрема можна віднести месенджери та шифровані дзвінки через Інтернет. За останнє десятиліття стрімко набрали популярність протоколи наскрізного шифрування. Через те, що Інтернет перетворювався і продовжує перетворюватися на усе більш централізовану систему, де контроль над даними консолідується у 7–8 компаніях (Meta, Apple, Amazon, Google та ін.), які за будь-якою вимогою держав змушені передавати приватні дані користувачів, виникли підходи, що застосовують комбінацію симетричного та асиметричного шифрування задля забезпечення комунікації двох або більше учасників без розкриття нешифрованих даних сервера, що реалізує передавання цих даних. Таким чином, з'явилися як приватні месенджери на кшталт Signal (раніше Whisper), так і функціональність приватних діалогів у інших популярних месенджерах, як-от WhatsApp та Telegram. Такі месенджери набули широкого використання на найвищих рівнях управління державою, що підтверджує нещодавній скандал із Signal та плануванням військових операцій США [1]. Певні особливості наскрізного шифрування роблять використання такої функціональності обмеженим. Це, в основному, означає втрату даних діалогу у випадку втрати самого девайсу, на якому відбувається комунікація. Іншою важливою проблемою наскрізного шифрування є сувора необхідність використання HSM [12] для безпечного зберігання криптографічних ключів на пристрої. Утім, у реаліях російського вторгнення в Україну потреба у наскрізному шифруванні значно зросла, адже не усі держави можуть забезпечити своїм збройним силам безпечний зв'язок на великих відстанях, який достатньо захищений від зовнішнього впливу та кібератак. Кібератаки становлять серйозний виклик комунікації, адже проникнення у систему, що зберігає ключі шифрування до багатьох пристроїв, автоматично означає ризик дешифрування цих даних і витоку секретної інформації. Також, у випадку роботи груп із захисту прав людей, деякі держави зацікавлені у небезпеці та фізичному знищенні цих груп. Наскрізне шифрування дає можливість створити безпечний канал комунікації у тому випадку, коли серверу неможливо повністю довіряти. Попри вже

широкий досвід використання наскрізного шифрування у системах для комунікації, певні ділянки залишаються недостатньо дослідженими. До них можна віднести стратегічні комунікації на кшталт симуляції військових дій із двома або більше учасниками, класичні ігри та інші. У цій статті розглядається можливість розширення застосування протоколів наскрізного шифрування у середовищах з відсутнім безпечним середовищем, створення application-level протоколу шифрованої комунікації на базі алгоритму Double Ratchet [8] та приклад його використання у класичній покроковій грі.

1. Протокол шифрування

Пропонований протокол шифрування є імплементацією протоколу Double Ratchet без використання шифрування хедерів. Для імплементації були вибрані такі криптографічні алгоритми:

1. Для реалізації функцій ***ENCRYPT*** і ***DECRYPT*** — алгоритм шифрування AES-256-GCM-SIV [5]. Вибір цього алгоритму обґрунтовується додатковим захистом системи від зламів у випадку перевикористання послідовності байтів nonce.
2. Для реалізації функцій ***GENERATE_DH*** і ***DH*** — алгоритм Diffie-Hellman, що використовує криву Curve25519 [2], або ж скорочено X25519. Це відповідає рекомендаціям авторів специфікації протоколу Double Ratchet.
3. Для реалізації ***KDF_RK*** — алгоритм HKDF [7] з хеш-функцією SHA-256. У рекомендації авторів вказано, що автор імплементації має самостійно вибрати послідовність байтів для інформаційного параметру. Ми вибрали набір байтів, який кодує слово “Checkers”.
4. Для реалізації ***KDF_CK*** — алгоритм HMAC [6] з хеш-функцією SHA-256. Це відповідає рекомендаціям авторів специфікації протоколу.

Імплементація протоколу виконана мовою Rust [13]. Мова Rust вибрана для імплементації тому, що вона забезпечує: безпеку пам’яті мови, широкий вибір криптографічних бібліотек, можливість компіляції у бінарний формат WebAssembly [14].

Протокол шифрування, окрім реалізації функцій, визначених у Double Ratchet, також реалізує API:

1. Функція ***w_init_ratchet_sender*** дозволяє генерувати інтерфейс мовою Javascript та ініціалізує криптографічні параметри відправника першого повідомлення у протоколі.
2. Функція ***w_init_ratchet_receiver*** дозволяє генерувати інтерфейс мовою Javascript та ініціалізує криптографічні параметри отримувача першого повідомлення у протоколі.
3. Структура ***Header*** відповідає структурі ***Header*** у Double Ratchet та містить публічний ключ відправника, створеного алгоритмом X25519; кількість надісланих повідомлень у минулому ланцюгу надсилання, та кількість надісланих повідомлень у поточному ланцюгу надсилання.
4. Структура ***UserClientData*** зберігає у собі стан усіх криптографічних параметрів, що потрібні для шифрування та дешифрування повідомлень, а також ініціалізації сторін. Протокол Double Ratchet визначає дві операції зміни ключів: symmetric key ratchet та asymmetric key ratchet. Symmetric key ratchet відбувається при шифруванні та дешифруванні кожного нового повідомлення в одному ланцюгу надсилання / отримання. Таким ланцюгом називають упорядковану послідовність повідомлень, які використовували стале значення результату алгоритму Diffie-Hellman операції двох користувачів. Для шифрування або дешифрування кожного повідомлення у ланцюгу надсилання або ланцюгу отримання відповідно протокол мусить добути новий ключ із батьківського ключа ланцюга. Батьківський ключ ланцюга оновлюється з кожним новим шифрованим або дешифрованим повідомленням. Щоби забезпечити функціональність, структура зберігає початковий батьківський ключ для видобування першого ключа ланцюга надсилання / отримання, поточні ключі ланцюгів надсилання / отримання, кількість надісланих повідомлень, кількість отриманих повідомлень, довжину попереднього ланцюга надсилання, а також пропущені повідомлення. Особливістю імплементації протоколу є використання X25519 публічного ключа отримувача у ролі ***associated data*** [3]. Оскільки публічний ключ отримувача може змінюватися при операції asymmetric key ratchet, протокол зберігає дані про пропущені повідомлення у хеш-таблиці та в такому форматі: [*публічний_ключ_відправника, номер_повідомлення*] -> [*ключ_дешифрування, публічний_ключ_отримувача*]. Ця сукупність даних дозволяє проводити операцію symmetric key ratchet. Втім, протокол Double Ratchet також визначає операцію asymmetric key ratchet. Для забезпечення цієї операції, у структурі ***UserClientData*** зберігаються параметри ***dh_s*** (приватний X25519 ключ відправника) і ***dh_r*** (публічний X25519 ключ отримувача).

5. Структура ***HeaderCiphertext*** представляє трійку значень Header, байтове представлення зашифрованого тексту і новий стан ***UserClientData*** після шифрування. Повертається методом ***w_ratchet_encrypt***.
6. Структура ***Plaintext*** представляє пару значень, що містить байтове представлення незашифрованого тексту і новий стан ***UserClientData*** після дешифрування. Повертається методом ***w_ratchet_decrypt***.
7. Функції ***w_ratchet_encrypt*** та ***w_ratchet_decrypt***. Ці функції є головними при шифруванні та дешифруванні повідомлень. Утім, окрім отримання звичних вхідних параметрів (незашифрований текст для ***encrypt***, хедер та зашифрований текст для ***decrypt***), функції також отримують копію структури ***UserClientData***, закодованої як значення з Javascript. Визначивши API, що доступне Javascript, потрібно зробити крок від програми мовою Rust до модуля Javascript.

WebAssembly нативно підтримується у більшості великих браузерів. Це означає, що браузери можуть слугувати хостами для контейнеризованого виконання бінарного коду WebAssembly, виділяти пам'ять WebAssembly модулям і викликати функції цих модулів. Оскільки WebAssembly модуль у Javascript є обгорткою над виконуваним файлом, з точки зору користувача бібліотеки, потрібно знати вказівники до функцій, типи аргументів цих функцій тощо. Щоби полегшити практичне використання бібліотеки, частою практикою є створення зовнішніх Javascript функцій та класів, що звертаються до відомих на момент компіляції секцій у бінарному файлі. Для мови програмування Rust зазначену інфраструктуру надають такі компоненти:

1) нативна компіляція коду у формат *wasm32;*
2) бібліотека ***wasm-bindgen*** [10], що надає інформацію про те, для яких компонентів генерувати Javascript відповідники;
3) застосунок командного рядка ***wasm-pack*** [11], котрий генерує Javascript модуль для WebAssembly коду.

Завдяки цим компонентам можливо виконати перехід від наявності Rust бібліотеки, яку можуть використовувати програми, написані мовою Rust, C, або C++, до WebAssembly модуля, який може виконуватися в браузерному середовищі. Альтернативними підходами, які використала *libsignal*, є створення клієнтських бібліотек мовами програмування нативних платформ — Swift для iOS, Java для Android.

У нашому випадку, ***wasm-pack*** виконує критичну роль, генеруючи Javascript модуль, що містить чотири файли: ***double_ratchet.d.ts***, ***double_ratchet.js***, ***double_ratchet_bg.wasm.d.ts***, ***double_ratchet_bg.wasm***. Погляньмо на них детальніше. ***double_ratchet.d.ts*** містить типізовані визначення функцій як розробленої бібліотеки, так і внутрішніх функцій ***wasm-bindgen***.

```
1 /* tslint:disable */
2 /* eslint-disable */
3 export function w_new_heade:(sender_public_key: Uint8Array, previous_sending_chain_length: bigint, num_sent: bigint): any;
4 export function w_init_ratchet_sender(dh_r: Uint8Array, shared_secret: Uint8Array): any;
5 export function w_init_ratchet_receiver(dh_s: Uint8Array, shared_secret: Uint8Array): any;
6 export function w_ratchet_encrypt(data: any, plaintext: Uint8Array): HeaderCiphertext;
7 export function w_ratchet_decrypt(data: any, header: Header, ciphertext: Uint8Array): Plaintext;
```

Рис. 1. Сигнатури функцій, що генеруються для мови Typescript

Файл ***double_ratchet.js*** містить імплементацію наведених у рис. 1 функцій, а також внутрішніх функцій ***wasm-bindgen***.

Файл ***double_ratchet_bg.wasm.d.ts*** містить додаткові генеровані Typescript сигнатури для коректної роботи ***wasm-pack***.

Файл ***double_ratchet_bg.wasm*** є виконуваним файлом, що генерується після компіляції Rust коду у цільову архітектурну трійку wasm32-unknown-unknown.

Протокол шифрування в загальному випадку складається з трьох кроків: встановлення значення спільного секрету, ініціалізації сторін, шифрування та дешифрування повідомлень.

Однією з небагатьох «логічних» вразливостей цього алгоритму є саме перший крок — створення спільного секрету. Автори Double Ratchet пропонують [8] встановлювати спільний секрет за допомогою протоколів узгодження ключів. Обравши алгоритм X25519, клієнти створюють пари ключів, де перший клієнт надсилає свій публічний ключ другому клієнту, після чого другий клієнт виконує операцію скалярного множення публічного ключа (точка на еліптичній кривій Curve25519) на секретний ключ (скаляр із поля, визначеного Curve25519) та надсилає свій публічний ключ першому

клієнту для виконання такої самої операції множення. Таким чином, користувачі прийдуть до одних і тих самих значень спільного секрету. Важливою проблемою такого підходу є атака Man-In-The-Middle з боку активного нападника, який може надати клієнтам власні значення публічного ключа і створити шифрований канал комунікації з будь-яким із клієнтів. Традиційно, для подолання таких проблем використовується протокол TLS, втім це все одно вимагатиме довіри до сервера, що є посередником у цій комунікації. Для розв'язання цієї проблеми ведуться дослідження в напрямі застосування zero-knowledge proofs у протоколі TLS [4]. У наступному розділі ми наведемо приклад генерації спільного секрету.

Другий крок — ініціалізація сторін — відбувається на боці відправника першого повідомлення та отримувача першого повідомлення незалежно.

Третій крок — шифрування та дешифрування повідомлень. Отримавши доступ до структури ***UserClientData***, можливо шифрувати будь-який plaintext для отримувача.

Практична значущість розробленого протоколу особливо актуальна в контексті сучасних безпекових викликів, зокрема в ситуаціях, де не можна довіряти мережевій інфраструктурі або існує ризик кібератак, що компрометують важливі операції. Створення альтернативи ***libsignal*** із можливістю інтеграції у середовища WebAssembly значно розширює сферу можливих застосувань наскрізного шифрування, зокрема у веборієнтованих застосунках, ігрових платформах та системах командування і контролю військами, що виконуються у браузері або інших середовищах [15].

2. Сервер комунікації

У цьому розділі розглядається створення сервера для комунікації учасників певного стратегічного завдання. Створений протокол може використовуватися у задачах, що абстрагують текстову комунікацію від клієнта і представляють певні дані у відомому і реплікованому форматі. Прикладом є симуляції поля бою або покрокові ігри з відомими публічними параметрами. До останніх можна віднести шахи і шашки: учасники починають комунікацію з наперед відомого та відкритого стану, тобто шахової дошки з розставленими на ній фігурами. Специфічний вибір таких задач дає змогу вилучити сервер із ланцюга прийняття ігрових рішень, адже немає недетермінованих процесів, що відповідають за зміну стану гри. У нашому випадку функції сервера, зокрема створення каналу комунікації між двома наперед відомими користувачами, були розширені можливістю створювати відкриті сесії, до яких можуть доєднатися будь-які користувачі.

Щоб підтримувати безперервну комунікацію між користувачами, потрібно обрати протокол, що дозволяє як отримувати повідомлення користувачів, так і передавати повідомлення користувачам на сторону клієнта. Для такої задачі був обраний протокол Websocket. Наступним поняттям сервера є сесія, яка складається з двох гравців, кожен з яких підключений до сервера окремим Websocket підключенням і має внутрішній ідентифікатор підключення, що зберігається у пам'яті сервера. Для кожного користувача виділяється канал необмеженого розміру, що дозволяє надсилати користувачу відповіді та отримувати від користувача запити. Важливо зазначити, що користувачів та сесії сервер розглядає як *ефемерні*. Ефемерність сесії означає, що той самий фізичний користувач може закрити підключення до сервера, у випадку чого сервер видалить цього користувача з хеш-таблиці користувачів і з сесії. За повторного підключення користувача до сервера користувач розглядатиметься як абсолютно новий.

Сервер імплементує певну логічну структуру повідомлень. Для коректної роботи, користувач має мати змогу створити відкриту сесію (анонсувати свою присутність іншим користувачам), отримавши ідентифікатор сесії; створити запит на долучення до наявної сесії; підтвердити запит на долучення до наявної сесії; надіслати повідомлення.

Загальною структурою повідомлення, що надсилають користувачі, є структура ***ClientMessage***, що містить поля ***sid***, ***cmd*** та ***sig***. Поле ***sid*** може опційно містити ідентифікатор сесії, представлений додатним двобайтним цілим числом. Поле ***cmd*** містить серіалізовану команду, визначену у переліку варіантів ***Cmd***. Поле ***sig*** обов'язково містить цифровий підпис певного набору байтів, що використовується для верифікації автентичності відправленого повідомлення. Верифікація цифрового підпису є головним механізмом автентифікації ефемерних користувачів у імплементації протоколу. Алгоритм EdDSA (Ed25519) був обраний для створення та верифікації цифрових підписів. Вибір алгоритму EdDSA обумовлений його високою продуктивністю, малим розміром ключів та підписів, що особливо важливо у вебсередовищах, де ресурси обмежені. До переваг EdDSA над альтернативними

алгоритмами (ECDSA на кривій secp256k1, Schnorr-Ristretto на кривій Ed25519) також можна додати широку доступність бібліотек для клієнтських середовищ.

Структура ***AnnounceArgs*** містить аргументи для створення сесії користувачем — ім'я цього користувача та його публічний ключ алгоритму EdDSA для верифікації цифрових підписів. Структура ***JoinSessionArgs*** містить параметри для створення запиту на доєднання користувача до вже наявної сесії — ім'я користувача, що доєднується, публічний ключ алгоритму X25519, та публічний ключ алгоритму EdDSA для верифікації цифрових підписів. Структура ***ConfirmJoinSessionArgs*** визначає параметри для прийняття вказаного у параметрі ***guest_username*** користувача та публічний ключ алгоритму X25519 користувача, що створив сесію. На цьому етапі користувачі мають можливість створити спільний секрет та ініціалізуватися як відправник і отримувач. Після доєднання другого користувача до сесії сервера відомі всі необхідні публічні ключі алгоритму EdDSA, проте у структурі ***SendSesssionMessage*** обов'язковою є наявність параметра ***public***, що містить публічний ключ EdDSA відправника повідомлення для визначення ідентичності надсилача. Крім цього, вона містить шифрований текст повідомлення у кодуванні Base64 і дані структури ***Header*** з протоколу шифрування. Для опрацювання усіх повідомлень використовується формат серіалізації JSON. Як бачимо з набору повідомлень, що може отримувати сервер, сервер не підтримує реєстрації користувачів зі збереженням їхніх даних. Перевагою цього підходу є архітектурна простота рішення, недоліком — можливість підробки повідомлень завдяки підслуховуванню повідомлень користувачів та перевикористання цифрових підписів. Ця проблема розв'язується дизайном сервера: сесії є ефемерними, тобто для кожної нової сесії користувача вимагається створення нової пари EdDSA ключів і використання підписів для автентифікації. На практиці це означає, що підписи однією парою ключів публічних даних відбуваються лише один раз, бо для однієї сесії виділяється можливість лише один раз підписати та надіслати повідомлення ***Announce***, ***JoinSession***, ***ConfirmJoinSession***. У випадку ***SendSessionMessage*** використання підпису статичних даних зробило б перевикористання підпису статичних даних вразливим, адже підпис не залежав би від даних, що надсилаються. Щоби цьому запобігти, для сесії користувача сервер зберігає число, що відоме клієнту та серверу одночасно, і це число додається до даних, на яких виконується цифровий підпис.

У аргумент ***session_id*** може передаватися будь-яке позитивне додатне число, яке може бути постійним, що менш безпечно, і змінним, що унеможливлює підроблення або перевикористання підпису за дотримання умови ефемерності.

Для внутрішнього управління комунікаційними сесіями сервер зберігає дві структури даних — хеш-таблицю з сесіями, де ключем є ідентифікатор сесії, значенням — стан сесії, і хеш-таблицю, що дозволяє визначити сесію за ім'ям користувача. Стан сесії — це набір даних, наданих обома членами цієї сесії. У цій імплементації стан сесії містить стан хоста (відправника першого повідомлення) і стан гостя (отримувача першого повідомлення).

Щоб користувачі могли не тільки відправляти повідомлення у сесію, а й отримувати «відповіді», тобто факт успішно опрацьованого повідомлення від учасника сесії, сервер має надсилати трансформовані відповіді одному з учасників. Загалом, відповідь має таку саму структуру, як і запит, окрім наявності публічного ключа EdDSA в полях структури. Залежно від отриманого повідомлення, сервер визначає отримувача відповіді. Наприклад, анонсувавши сесію, сервер не надішле відповідь іншим користувачам, окрім автора повідомлення, позаяк до сесії ніхто більше не належить. У випадку запиту ***GetSessions*** відповідь також надається автору повідомлення. Відповідь для ***JoinSession*** працює дещо інакше: автор повідомлення чекає на підтвердження від хоста, тому відповідь потрібно передати хосту. Схожим чином працює відповідь до ***ConfirmJoinSession***, де автор сповіщає гостя про підтвердження сесії. ***SendSessionMessage*** за своїм визначенням завжди надсилає повідомлення протилежній стороні.

При обробленні повідомлень можуть виникнути помилки. Наприклад, надсилати повідомлення у сесії, де немає другого користувача, є помилковою дією. Так само усі цифрові підписи мають успішно верифікуватися. Врахувавши різні випадки, де можуть виникнути помилки, можна використати конструкцію enum мови Rust та бібліотеку, що дозволяє перетворювати окремі помилки в описовий текст.

Таким чином, сервер комунікації містить перелік повідомлень, відповідей та помилок для безпечного використання у рамках шифрованої комунікації. Реалізований серверний компонент має важливу практичну значущість, зокрема у задачах, що потребують швидкої, безпечної та приватної кому-

нікації у вебдодатках, стратегічних симуляціях, покрокових іграх та інших сценаріях з високими вимогами до конфіденційності та низької затримки передавання інформації.

3. Використання методу у покроковій грі

Як було зазначено у вступі, наскрізне шифрування може використовуватися у різних сферах, де існує передавання інформації між двома сторонами. Однією з таких сфер є покрокові ігри для двох людей. Вибір цієї сфери обґрунтовується схожістю впровадження наскрізного шифрування у попередньо незашифровану комунікацію, адже стратегічні взаємодії між двома сторонами, такі як симуляції військових дій, є вичерпним набором станів та повідомлень. Для ілюстрації такого впровадження були обрані шашки. Шашки — це добре досліджена у сфері комп'ютерних наук класична гра, котра ідеально підпадає під використання шифрованої комунікації між гравцями.

Проблема імплементації полягає у тому, що дані про будь-яку мультиплеєрну гру зазвичай зберігаються на сервері, що не відповідає критеріям наскрізного шифрування. Щоби метод відтворення стану працював коректно, потрібно перенести задачу зберігання даних та оброблення ходів на сторону клієнта. Для подолання цієї проблеми використовується WebAssembly: уся гра відбувається на сторонах двох клієнтів, де кожен надсилає стан дошки після закінчення власного ходу. Клієнти можуть проводити валідацію зміни стану, тобто перевіряти на правильність послідовність ходів, зроблених у межах однієї зміни стану. Для простоти імплементації цей крок пропускається.

Як базова імплементація був обраний проєкт [9] авторства Toms Zvirbulis під назвою chemkers. Проєкт містить імплементацію шашок мовою Rust, що компілюється у WebAssembly механізмами, описаними у розділі 1. Також у проєкті наявна імплементація клієнтської частини застосунку — вебдодаток мовою Typescript з використанням бібліотеки ***preact***.

У базовій імплементації був механізм генерування ходів за допомогою підходів minimax та alpha-beta pruning, проте у цьому випадку такі підходи були зайвими. Замість цього гравці очікують закінчення ходу від противника і застосовують наданий противником стан до дошки на клієнті. Для підтримки такої роботи застосунку був доданий модуль ***network-context*** за патерном Redux. У модулі, відповідно до патерна, є actions, reducer, dispatch типи.

Підключення до сервера гри відбувається завдяки встановленню WebSocket з'єднання.

Шифрування та дешифрування повідомлень можливе завдяки використанню модуля Javascript, створеного в розділі 1.

Остання велика деталь, яка уможливлює зашифрований мультиплеєр для шашок, — сесії. Для того, щоби не дозволити початок гри до формування сесії, було розроблено окремий TSX компонент, котрий реалізує створення та підключення до сесій.

Висновки

У цій роботі було досліджено можливість розширення застосування протоколів наскрізного шифрування на основі алгоритму Double Ratchet у застосунках із низькою довірою до сервера, зокрема у покрокових іграх та стратегічних взаємодіях. Головним досягненням цієї роботи є власна імплементація протоколу Double Ratchet мовою Rust із можливістю компіляції у формат WebAssembly, реалізований сервер на базі WebSocket, а також вебклієнт гри у шашки для демонстрації роботи розробленого протоколу. Область наскрізного шифрування потребує дослідження додаткових застосувань поза вже звичними, як-от шифрована комунікація у текстових месенджерах. Унікальність роботи полягає у першості імплементації Double Ratchet, що компілюється у WebAssembly. Розроблена імплементація протоколу може безпечно використовуватися у будь-яких застосунках, що мають на меті впровадження наскрізного шифрування та задовольняються критерієм ефемерності сесій (у тих випадках, де секрети зберігаються поза безпечним середовищем). Вибір Rust гарантував безпеку пам'яті та ефективність роботи криптографічних алгоритмів. WebAssembly дозволив швидко та безпечно інтегрувати ці алгоритми у вебклієнти. WebSocket забезпечив низьку затримку передавання даних між клієнтами, критично важливу для інтерактивних застосунків. Реалізований сервер підтримує ефемерні сесії, які гарантують мінімальні ризики компрометації інформації, і використовує цифрові підписи (EdDSA) для автентифікації користувачів. Логічна маршрутизація запитів забезпечує ефективне передавання повідомлень у захищених сценаріях. Вибір класичної гри у шашки як прикладу дозволив ефективно продемонструвати переваги наскрізного шифрування та

можливості реалізованого протоколу. Всі криптографічні операції, включно із генерацією ключів, шифруванням та дешифруванням повідомлень, успішно виконуються на клієнтських пристроях. Важливим є покращення механізмів обробки помилок та оптимізація роботи WebAssembly. Цікавою ділянкою подальших досліджень є створення механізмів zero-knowledge proofs для запобігання атакам типу Man-In-The-Middle під час створення спільного секрету, оптимізація інтеграції з криптографічними модулями апаратної безпеки (HSM) та дослідження можливостей масштабування рішення. Описане дослідження важливе в умовах частих кібератак на інфраструктуру різного рівня, коли вимоги до конфіденційності та безпеки інформації значно зросли. Запропонований підхід може використовуватися для розв'язання реальних задач захисту інформації, де критично важливою є довіра до каналу передавання даних. Таким чином, у процесі роботи створено комплексне рішення, що охоплює криптографічний протокол, серверну частину і вебклієнт, яке демонструє життєздатність застосування наскрізного шифрування у браузерних середовищах та мультиплеєрних іграх. Роботу може бути використано як основу для подальших досліджень та розробок у сфері безпеки комунікаційних систем та приватності у мультиплеєрних іграх.

Список літератури

1. BBC [Electronic resource]. — Signalgate. — 2025. — Mode of access: https://www.bbc.com/ukrainian/articles/c4g9n525lp7o (date of access: 26.03.2025).
2. Bernstein D. Curve25519: New Diffie-Hellman Speed Records / D. Bernstein // M. Yung, Y. Dodis, A. Kiayias, T. Malkin, eds. Public Key Cryptography - PKC 2006. PKC 2006. Lecture Notes in Computer Science. — Springer, Berlin, Heidelberg, 2026. — Vol. 958. — Pp. 207–228. — doi.org/10.1007/11745853_14.
3. Black J. Authenticated encryption / J. Black // H. C. A. van Tilborg, ed. Encyclopedia of Cryptography and Security. — Springer, Boston, MA, 2005. — https://doi.org/10.1007/0-387-23483-7_15.
4. Grubbs P. Zero-Knowledge Middleboxes [Electronic resource] / P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, M. Walfish // 31st USENIX Security Symposium (USENIX Security 22). — 2022. — Mode of access: https://www.usenix.org/conference/usenixsecurity22/presentation/grubbs (date of access: 15.06.2025).
5. Gueron S. AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption [Electronic resource] / S. Gueron, A. Langley, Y. Lindell. — 2019. — Mode of access: https://www.rfc-editor.org/info/rfc8452.
6. Krawczyk H. HMAC: Keyed-Hashing for Message Authentication [Electronic resource] / H. Krawczyk, M. Bellare, R. Canetti. — 1997. — Mode of access: https://www.rfc-editor.org/info/rfc2104 (date of access: 15.06.2025).
7. Krawczyk H. HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [Electronic resource] / H. Krawczyk, P. Eronen. — 2010. — Mode of access: https://www.rfc-editor.org/info/rfc5869 (date of access: 15.05.2025).
8. Perrin T. The Double Ratchet Algorithm [Electronic resource] / T. Perrin, M. Marlinspike. — 2016. — Mode of access: https://signal.org/docs/specifications/doubleratchet, https://doi.org/10.1007/978-3-031-33386-6_16 (date of access: 15.06.2025).
9. Redux. A JS library for predictable and maintainable global state management [Electronic resource] // Redux. — 2025. — Mode of access: https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers (date of access: 15.06.2025).
10. Rustwasm. wasm-bindgen [Electronic resource] // Github. — Mode of access: https://github.com/rustwasm/wasm-bindgen (date of access: 15.06.2025).
11. Rustwasm. wasm-pack [Electronic resource] // Github. — Mode of access: https://github.com/rustwasm/wasm-pack (date of access: 15.06.2025)
12. Sommerhalder M. Hardware Security Module [Electronic resource] / M. Sommerhalder // V. Mulder, A. Mermoud, V. Lenders, B. Tellenbach, eds. Trends in Data Protection and Encryption Technologies. Springer, Cham, 2023. — Mode of access: URL: https://www.rfc-editor.org/info/rfc8452 (date of access: 24.06.2025).
13. The Rust Programming Language [Electronic resource]. — Mode of access: https://www.rust-lang.org/ (date of access: 15.06.2025).
14. WebAssembly Community Group [Electronic resource] // WebAssembly Specification. — 2025. — Mode of access: https://webassembly.github.io/spec/core/ (date of access: 15.06.2025).
15. Zvirbulis T. Chemkers [Electronic resource] / T. Zvirbulis // Github. — 2022. — Mode of access: https://github.com/tomszir/chemkers (date of access: 15.06.2025).

References

BBC. (2025). Signalgate. https://www.bbc.com/ukrainian/articles/c4g9n525lp7o.

Bernstein, D. J. (2006). Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, T. Malkin (Eds.), *Public Key Cryptography - PKC 2006*. PKC 2006. Lecture Notes in Computer Science, Vol. 3958. Springer, Berlin, Heidelberg.

Black, J. (2005). Authenticated encryption. In H. C. A. van Tilborg (Ed.), *Encyclopedia of cryptography and Security*. Springer, Boston, MA. https://doi.org/10.1007/0-387-23483-7_15.

Grubbs, P., Arun, A., Zhang, Y., Bonneau, J., & Walfish, M. (2022). Zero-knowledge middleboxes. In *31st USENIX Security Symposium (USENIX Security 22)*. https://www.usenix.org/conference/usenixsecurity22/presentation/grubbs.

Gueron, S., Langley, A., & Lindell, Y. (2019). AES-GCM-SIV: Nonce misuse-resistant authenticated encryption.

Krawczyk, H., & Eronen, P. (2010). HMAC-based extract-and-expand key derivation function (HKDF). https://www.rfc-editor.org/info/rfc5869.

Krawczyk, H., Bellare, M., & Canetti, R. (1997). HMAC: Keyed-hashing for message authentication. https://www.rfc-editor.org/info/rfc2104.

Perrin, T., & Marlinspike, M. (2016). The Double Ratchet Algorithm. https://signal.org/docs/specifications/.

Redux. (2025). A JS library for predictable and maintainable global state management. https://redux.js.org.

Rust Project Developers. (n. d.). The Rust programming language. https://www.rust-lang.org/.
Rustwasm. (n. d.). wasm-bindgen. GitHub. https://github.com/rustwasm/wasm-bindgen.
Rustwasm. (n. d.). wasm-pack. GitHub. https://github.com/rustwasm/wasm-pack.
Sommerhalder, M. (2023). Hardware security module. In V. Mulder, A. Mermoud, V. Lenders, & B. Tellenbach (Eds.), *Trends in Data Protection and Encryption Technologies*. Springer, Cham. https://www.rfc-editor.org/info/rfc8452.
WebAssembly Community Group. (2025). WebAssembly specification. https://webassembly.github.io/spec/.
Zvirbulis, T. (2022). Chemkers. GitHub. https://github.com/tomszir/chemkers.

O. Mykhailenko, K. Gorokhovsky, S. Gorokhovsky

METHOD OF ENCRYPTED COMMUNICATION IN STRATEGIC INTERACTIONS

The paper explores the possibility of expanding the use of end-to-end encryption protocols based on the Double Ratchet algorithm in applications with low trust in the server, particularly in turn-based games and strategic interactions. The relevance of the research is due to the growing need for secure communication in cyberattacks, especially during military operations. The field of end-to-end encryption requires the study of additional applications beyond the usual ones, such as encrypted communication in text messengers. The developed implementation of the protocol can be safely used in any applications that aim to implement end-to-end encryption and satisfy the criterion of session ephemerality (in cases where secrets are stored outside a secure environment). The implemented server supports ephemeral sessions, which guarantee minimal risks of information compromise, and uses digital signatures (EdDSA) for user authentication. Logical routing of requests ensures efficient message transmission in secure scenarios. The choice of the classic game of checkers as an example allowed the authors to effectively demonstrate the advantages of end-to-end encryption and the capabilities of the implemented protocol. All cryptographic operations, including key generation, encryption and decryption of messages, are successfully performed on client devices. It is important to improve error handling mechanisms and optimize the operation of WebAssembly. An interesting area of further research is the creation of zero-knowledge proof mechanisms to prevent Man-In-The-Middle attacks during the creation of a shared secret, optimizing integration with cryptographic hardware security modules (HSM), and exploring the scalability of the solution. The proposed approach can be used to solve real-world information security problems where trust in the data transmission channel is critically important. Thus, the work has created a comprehensive solution that includes a cryptographic protocol, a backend, and a web client, which demonstrates the viability of end-to-end encryption in browser environments and multiplayer games. The work can be used as a basis for further research and development in the field of security of communication systems and privacy in multiplayer games.

Keywords: end-to-end encryption, Double Ratchet, WebAssembly, strategy games, WebSocket, Rust, secure communication.

Матеріал надійшов 25.06.2025