Politechnika Warszawska

WYDZIAŁ ELEKTRYCZNY

Instytut Elektrotechniki Teoretycznej i Systemów Informacyjno-Pomiarowych

Praca dyplomowa inżynierska

na kierunku Informatyka Stosowana

w specjalności Inżynieria Oprogramowania

Projekt i implementacja biblioteki do poufnego przechowywania wiadomości po stronie serwera przy użyciu protokołu Labyrinth

Sebastian Piotr Pawliński

numer albumu 313869

promotor

dr inż. Bartosz Chaber

WARSZAWA 2025

Projekt i implementacja biblioteki do poufnego przechowywania wiadomości po stronie serwera przy użyciu protokołu Labyrinth

Streszczenie

Praca przedstawia projekt i implementację biblioteki klienckiej umożliwiającej poufne przechowywanie wiadomości po stronie serwera z wykorzystaniem protokołu Labyrinth opracowanego przez firmę Meta. Rozpoczyna się od opisu protokołu Labyrinth, omawiając kluczowe pojęcia oraz poszczególne fazy jego działania. W protokole tym wiadomości są szyfrowane przy użyciu sekretów znanych wyłącznie urządzeniom użytkownika, a następnie przechowywane na serwerze w formie zaszyfrowanej, co uniemożliwia ich odczytanie przez osoby nieuprawnione. Istotnym elementem protokołu jest mechanizm odzyskiwania dostępu do historii wiadomości na nowych urządzeniach użytkownika. Szczególną rolę odgrywa rotacja kluczy szyfrujących, która umożliwia unieważnianie nieaktywnych urządzeń i blokowanie ich dostępu do nowych wiadomości. Praca opisuje pierwszą otwartoźródłową implementację Labyrinth, której celem jest umożliwienie dyskusji oraz analizy protokołu. Dodatkowo omawiane są kluczowe kwestie, co czyni ją cennym źródłem do lepszego zrozumienia protokołu oraz jego mechanizmów.

Kolejnym elementem pracy jest omówienie stworzonej aplikacji internetowej w postaci komunikatora tekstowego, składającego się z części klienckiej wykorzystującej zaprezentowaną bibliotekę oraz serwera obsługującego operacje zgodnie z założeniami protokołu. Praca zawiera również szczegółowy opis procesu testowania aplikacji, mającego na celu weryfikację poprawności działania protokołu oraz zgodności implementacji z jego założeniami.

W ramach implementacji biblioteki wykorzystano język TypeScript, a część wizualną aplikacji zrealizowano przy użyciu React. Serwer został stworzony w języku Java korzystając ze szkieletu Spring Boot, a jako silnik bazy danych zastosowano PostgreSQL. Całość aplikacji przetestowano przy użyciu biblioteki Cypress.

Całość kończy się podsumowaniem, gdzie przedstawione zostały kierunki rozwoju projektu oraz wnioski odnośnie bezpieczeństwa protokołu Labyrinth.

Słowa kluczowe: komunikacja, cyberbezpieczeństwo, poufność, prywatność, komunikator, szyfrowanie, protokół Labyrinth

Design and Implementation of a Library for Confidential Server-Side Message Storage Using the Labyrinth Protocol

Abstract

The thesis presents the design and implementation of a client library enabling the secure storage of messages on the server side using the Labyrinth protocol developed by Meta. It begins with a description of the Labyrinth protocol, discussing key concepts and the various phases of its operation. In this protocol, messages are encrypted using secrets known only to the user's devices and then stored on the server in an encrypted form, preventing unauthorized access. A key element of the protocol is the mechanism for recovering access to message history on new devices. A crucial role is played by the rotation of encryption keys, which allows for the deactivation of inactive devices and blocks their access to new messages. The thesis describes the first open-source implementation of Labyrinth, aimed at enabling discussion and analysis of the protocol. Additionally, it addresses key issues, making it a valuable resource for a better understanding of the protocol and its mechanisms.

Another part of the thesis discusses the created web application in the form of a text instant messenger, consisting of the client-side utilizing the presented library and a server handling operations in accordance with the protocol's assumptions. The thesis also includes a detailed description of the application testing process, aimed at verifying the correctness of the protocol's operation and ensuring the implementation aligns with its assumptions.

The library implementation used TypeScript, and the visual part of the application was created using React. The server was built in Java using the Spring Boot framework, and PostgreSQL was used as the database engine. The entire application was tested using the Cypress library.

The thesis ends with a summary, presenting the directions for the project's development and conclusions regarding the security of the Labyrinth protocol.

Keywords: communication, cybersecurity, confidentiality, privacy, instant messenger, encryption, Labyrinth protocol

Spis treści

Rozdział 1

Wstęp

1.1 Opis problemu

Komunikacja odgrywa kluczową rolę w życiu ludzi, stopniowo wraz z postępem technologicznym forma komunikacji się zmienia, coraz częściej wybierane są rozwiązania komunikacji poprzez internet. Według raportu Urzędu Komunikacji Elektronicznej [12] o stanie rynku telekomunikacyjnego z roku 2023 można zauważyć coroczny spadek czasu trwania połączeń telefonicznych oraz liczby wysłanych wiadomości typu Short Message Service (SMS) przy jednoczesnym wzroście liczby użytkowników internetu. Rynek komunikacji na odległość przejmują komunikatory internetowe, których większość jest darmowa. W odróżnieniu od tradycyjnych form, użytkownik nie ponosi dodatkowych kosztów za wysłanie zdjęcia lub wiadomości za granicę. W większości takich rozwiązań nadawca wysyła wiadomość do serwera, którą serwer następnie przekazuje do odbiorcy. Powszechną formą zadbania o poufność użytkownika w typie komunikacji klient-serwer jest szyfrowanie Point-To-Point (P2P), polegające na szyfrowaniu danych pomiędzy punktami komunikacji.

W tym przypadku wiadomości są szyfrowane tylko na linii komunikacji, co oznacza że serwer ostatecznie posiada wiadomość w postaci jawnej. Wiele aplikacji zakłada przechowywanie takich danych ze względu na prostotę implementacji, a także jej szybsze działanie. Kluczową funkcjonalnością, jaką oferują tego typu aplikacje, jest przechowywanie historii wiadomości, co pozwala na jej przeglądanie z dowolnego urządzenia. Jest to szczególnie przydatne, gdy urządzenie użytkownika ulegnie awarii, a odzyskanie wiadomości w inny sposób nie będzie możliwe. Niestety prostota podczas tworzenia aplikacji i wygoda użytkowania często nie idzie w parze z bezpieczeństwem danych. Nierzadko serwery są ofiarami ataków hakerskich, których celem jest wyciągnięcie danych wrażliwych użytkowników. W przypadku komunikatorów szczególnie cenna jest historia korespondencji, w której mogą znajdować się dane takie jak: adresy zamieszkania, numery telefonów, hasła czy też nasze prywatne rozmowy oraz opinie. Dodatkowo historia pokazuje, że nie tylko o ataki hakerskie powinniśmy się bać. Śledztwo opisane w artykule The New York Times [1] ujawniło nieuczciwe praktyki jednej z największych firm na świecie, czyli Facebooka (obecnie Meta). W ramach umów z gigantami technologicznymi Facebook udostępniał dane użytkowników bez ich wiedzy i zgody. W szczególności udostępniane były prywatne

wiadomości firmom takim jak Netflix czy Spotify. Sytuacja pokazuje, że bezpieczeństwo danych nie powinno opierać się wyłącznie na zaufaniu do producentów aplikacji, ponieważ użytkownik nie może mieć pewności, co serwer zrobi z wiadomościami, którego do niego dotrą.

Problem ten można rozwiązać stosując szyfrowanie End-To-End (E2E), które zakłada szyfrowanie na urządzeniu nadawcy i odszyfrowanie wyłącznie u odbiorcy. Dzięki temu serwer pośredniczący w komunikacji nie ma dostępu do treści jawnej wiadomości. Dzięki temu modelowi zabezpieczeń nawet w przypadku ataku na serwer bądź nieuczciwego działania, wiadomości są bezużyteczne bez odszyfrowania. Biorąc pod uwagę aktualnie zalecany szyfr AES, to złamanie go nawet przy użyciu najwydajniejszego superkomputera jest praktycznie niemożliwe [9]. Czas złamania tego szyfru przekracza przyjęty czas istnienia wszechświata. Można zatem założyć, że wiadomości przechowywane w ten sposób są bezpieczne.

Szczególnie ważnym dla rozwoju szyfrowania E2E jest protokół Signal, który szczegółowo opisuje kolejne kroki niezbędne do zapewnienia poufnej i bezpiecznej komunikacji. Protokół ten jest uznawany za najbardziej bezpieczną i zaawansowaną specyfikację w tej dziedzinie i jest podstawą szyfrowania w aplikacji Signal Messenger. Aplikacja, podobnie jak biblioteki implementujące ten protokół, są otwartoźródłowe, co pozwala na analizę i krytykę ekspertów cyberbezpieczeństwa z całego świata. Mimu wielu zalet szyfrowania E2E istnieją również pewne ograniczenia, protokół Signal opisuje tylko jak bezpiecznie przekazać wiadomość, a nie jak ją przechować. Aplikacje te nie oferują natywnej możliwości przechowywania historii korespondencji w chmurze, co jest mniej wygodne dla użytkowników przyzwyczajonych do tej funkcji.

Wartą uwagi aplikacją jest Messenger firmy Meta, która od początku swojego istnienia zakładał przechowywanie wiadomości po stronie serwera bez zastosowania szyfrowania E2E. W grudniu 2023 roku, po latach kontrowersji związanych z naruszeniami prywatności, Meta opublikowała artykuł [2], w którym zapowiedziała wprowadzenie ulepszeń do aplikacji. Firma poinformowała o planach dodania szyfrowania E2E, jako domyślnego sposobu zabezpieczenia wiadomości w oparciu o protokół Signal. Szczególnym wyzwaniem dla inżynierów Meta było pogodzenie tej zmiany z dotychczasowym działaniem Messengera, ponieważ aktualna baza użytkowników była przyzwyczajona do korzystania z aplikacji w przeglądarce i natywnego przechowywania wiadomości po stronie serwera. Do rozwiązania tego problemu Meta opracowała protokół „Labyrinth" [10], który pozwala na poufne przechowywanie wiadomości po stronie serwera przy jednoczesnym zachowaniu możliwości ich odzyskania na nowo dodanych urządzeniach użytkownika. Jest to pierwsze publicznie znane podejście do problemu przechowywania wiadomości po stronie serwera przy stosowaniu komunikatorów szyfrowanych E2E.

1.2 Cel projektu

Firma Meta udostępniła jedynie dokument opisujący protokół, który, niestety, cechuje się zwięzłością, nieprecyzyjnymi opisami oraz licznymi niedopracowanymi elementami. Dokument zawiera m.in. nieużywane definicje, pomija kluczowe szczegóły techniczne i pozostawia wiele istotnych kwestii

bez wyjaśnienia. Dodatkowo brak dostępu do zasobów takich jak kod źródłowy czy fragmenty implementacji utrudnia pełne zrozumienie i odtworzenie założeń protokołu. W związku z tym praca przedstawia protokół Labyrinth w sposób, w jaki został on zinterpretowany na podstawie dostępnych informacji.

Głównym celem pracy jest analiza i opis działania protokołu Labyrinth oraz stworzenie biblioteki klienckiej umożliwiającej jego obsługę. Biblioteka ta została zaprojektowana z myślą o intuicyjnym i wygodnym wykorzystaniu przez innych programistów. Aby zweryfikować poprawność implementacji, opracowana została aplikacja czatu tekstowego, umożliwiająca użytkownikom bezpieczną komunikację oraz przechowywanie wiadomości na serwerze w sposób poufny.

Istotnym elementem projektu jest również zdobycie wiedzy z zakresu współczesnej kryptografii oraz przeprowadzenie analizy protokołu pod kątem bezpieczeństwa i funkcjonalności. Dodatkowo praca może stanowić wartościowe źródło wiedzy dla osób, które w przyszłości podejmą się analizy protokołu Labyrinth. Wiele niejasnych aspektów oryginalnej dokumentacji zostało tutaj uszczegółowionych, co może znacznie ułatwić ich zrozumienie i interpretację.

Rozdział 2

Opis Labyrinth

2.1 Podstawy kryptograficzne

Niezbędne operacje kryptograficzne konieczne do zrozumienia protokołu Labyrinth zostały opisane poniżej.

- **Szyfrowanie** – proces służący do zabezpieczenia danych przed niechcianym dostępem, polega na zamianie czytelnej formy danych (`plaintext`) na niezrozumiałą (`ciphertext`). Proces odwrotny nazywany jest odszyfrowaniem.
- **Szyfrowanie symetryczne** – typ szyfrowania, w którym ten sam klucz jest używany zarówno do szyfrowania jak i odszyfrowania danych. Szyfrowanie symetryczne jest dużo wydajniejsze od szyfrowania asymetrycznego opisanego poniżej, jednak aby posłużyć się nim w komunikacji, należy najpierw bezpiecznie przekazać klucz.
- **Szyfrowanie asymetryczne** – metoda wykorzystująca parę kluczy: publiczny i prywatny. Klucz publiczny służy do szyfrowania danych i jest dostępny dla wszystkich, natomiast klucz prywatny służy do ich odszyfrowania i pozostaje poufny.
- **Funkcja wyprowadzania kluczy** – z ang. Key Derivation Function (KDF), algorytm umożliwiający generowanie nowych kluczy na podstawie istniejącego (`ikm` – Input Key Material). Unikalność kluczy zapewnia dodanie dodatkowych informacji (`info`) czy soli (`salt`). Proces wyprowadzania klucza jest deterministyczny i nieodwracalny, co oznacza, że z dla tego samego wejścia zawsze otrzymuje się ten sam klucz wyjściowy.

Do szyfrowania symetrycznego wykorzystano algorytm Advanced Encryption Standard (AES) w trybie blokowym Galois/Counter Mode (GCM), który zapewnia jednoczesną poufność i integralność danych. Dzięki temu chronione są zarówno treść komunikacji, jak i jej autentyczność. Tryb GCM wykorzystuje tag uwierzytelniania generowany podczas szyfrowania, który pozwala zweryfikować, czy dane nie zostały zmodyfikowane. Dodatkowym argumentem tej implementacji jest Additional Authenticated Data (AAD) wykorzystywany jako dodatkowa informacja uwzględniana przy weryfikacji znacznika uwierzytelniania.

Szyfrowanie asymetryczne odbywa się przy pomocy funkcji zdefiniowanej w protokole Labyrinth. Jest ona określana `labyrinth_hpke_encrypt` a jej opis został zdefiniowany pseudokodem przedstawionym na listingu 1. Funkcja zakłada korzystanie z dwóch par kluczy służących oddzielnie do szyfrowania i uwierzytelniania. Kolejnym parametrem jest Pre-Shared Key (PSK), którego nazwę można tłumaczyć jako „wstępnie współdzielony klucz". Jest on używany jako sekret znany przez wszystkich uczestników komunikacji przed szyfrowaniem. Składnik AAD jest dodatkowym argumentem wykorzystywanym przez `aes_gcm_256_encrypt`. Wykorzystany w ramach listingu operator || oznacza konkatenację bajtów.

Wartą szczególnej uwagi jest funkcja x25519. Służy ona do ustalenia wspólnego sekretu na podstawie klucza prywatnego i publicznego. Nazwa funkcji odnosi się do specyficznej implementacji opisanej w dokumencie RFC 7748 [6], która zapewnia bezpieczeństwo na poziomie 128 bitów. W implementacji tej zastosowano krzywą eliptyczną **Curve25519**, która jest specyficzną krzywą Montgomery'ego o określonych parametrach.

```
function labyrinth_hpke_encrypt(
    recipient_enc_pub,
    sender_auth_pub,
    sender_auth_priv,
    psk,
    aad,
    plaintext
) {
    assert_valid_curve_point(recipient_enc_pub)
    assert(length(psk) <= 32 bytes)
    (pub_ephem, priv_ephem) := generate_x25519_keypair()
    id_id := x25519(sender_auth_priv, recipient_enc_pub)
    id_ephem := x25519(priv_ephem, recipient_enc_pub)
    fresh_secret := id_id || id_ephem
    inner_aad :=
        0x01 || sender_auth_pub || recipient_enc_pub || pub_ephem || aad
    subkey := hkdf(fresh_secret, psk, inner_aad)
    nonce := 0x000000000000000000000000
    ciphertext := aes_gcm_256_encrypt(subkey, nonce, aad, plaintext)
    return 0x01 || pub_ephem || ciphertext
}
```

Listing 1. Pseudokod funkcji labyrinth_hpke_encrypt, źródło: protokół Labyrinth

Z kolei jako funkcję wyprowadzania kluczy wybrano implementację HMAC-based key derivation function (HKDF) opisaną w specyfikacji RFC 5869 [4]. Dokumentacja definiuje metodę, nazywaną **Extract-and-Expand**, która pozwala na uzyskanie do 255 kluczy na podstawie jednego. Mimo tak dużej liczby w protokole wyprowadzane są w taki sposób maksymalnie dwa klucze.

2.2 Skrócony opis rozwiązania

Protokół Labyrinth zakłada, że wiadomości są przechowywane na serwerze w postaci zaszyfrowanej. Zanim jednak trafią do serwera, muszą zostać dostarczone do użytkownika. Po ich odebraniu aplikacja użytkownika szyfruje wiadomości lokalnie, a następnie przesyła je na serwer, gdzie są bezpiecznie przechowywane. Dzięki temu użytkownik może pobrać historię wiadomości przy ponownym uruchomieniu aplikacji.

Kluczowym elementem tego procesu są sekrety przechowywane lokalnie przez użytkownika, które służą między innymi do szyfrowania wiadomości. W przypadku ich utraty istnieje możliwość odzyskania tych sekretów za pomocą kodu odzyskiwania, co pozwala na dostęp do historii wiadomości na nowych urządzeniach.

Protokół przewiduje również sytuację, w której urządzenie użytkownika pozostaje nieaktywne przez dłuższy czas. Pozostawienie tych samych sekretów na zarówno aktywnych, jak i nieaktywnych urządzeniach mogłoby stanowić zagrożenie bezpieczeństwa danych. Aby temu zapobiec, protokół wprowadza pojęcie epoki.

Epoka to okres, w którym żadne urządzenie użytkownika nie zostało unieważnione. Wszystkie urządzenia należące do danej epoki mają dostęp do tych samych sekretów. Jeśli któreś z urządzeń przekroczy maksymalny czas nieaktywności, tworzona jest nowa epoka, do której należą jedynie aktywne urządzenia. Dzięki temu mechanizmowi minimalizowane jest ryzyko utraty bezpieczeństwa wynikające z braku rotacji sekretów.

2.3 Słownik pojęć

Urządzenie Urządzenie to kryptograficzna reprezentacja powiązana z fizycznym sprzętem użytkownika, takim jak smartfon czy komputer. Składa się z zestawu kluczy wykorzystywanych do podpisu oraz szyfrowania danych:

- **deviceKeyPriv**, **deviceKeyPub** – para kluczy służąca do podpisywania innych kluczy w protokole,
- **epochStorageKeyPriv**, **epochStorageKeyPub** – para kluczy używana do asymetrycznego szyfrowania sekretów,
- **epochStorageAuthKeyPriv**, **epochStorageAuthKeyPub** – para kluczy używana do uwierzytelniania podczas szyfrowania asymetrycznego.

Epoka Epoka to okres w protokole Labyrinth, w którym obowiązują określone klucze kryptograficzne. Epoka składa się z następujących elementów:

- **epochRootKey** – główny sekret używany pośrednio do szyfrowania wiadomości i umożliwia dołączanie do następnych epok,
- **epochId** – identyfikator epoki nadawany przez serwer,

- **epochSequenceId** – wartość liczbowa określająca, którą z kolei jest dana epoka (np. pierwsza, druga, trzecia).

Wirtualne urządzenie Wirtualne urządzenie to abstrakcyjny byt w protokole, który pełni funkcję urządzenia służącego wyłącznie do odczytu sekretów epok. Tworzone jest podczas pierwszej inicjalizacji i posiada zestaw kluczy kryptograficznych podobny do zwykłego urządzenia:

- **deviceKeyPriv**, **deviceKeyPub** – para kluczy służąca do podpisywania innych kluczy w protokole,
- **epochStorageKeyPriv**, **epochStorageKeyPub** – para kluczy używana do asymetrycznego szyfrowania sekretów.

Wirtualne urządzenie różni się od zwykłego urządzenia brakiem kluczy używanych do uwierzytelniania nowych epok. Dzięki temu może jedynie odczytywać istniejące epoki, ale nie może inicjować nowych.

2.4 Fazy protokołu

Ze względu na złożoność protokołu, najłatwiejszą formą jego wyjaśnienia jest faz protokołu. W trakcie ich omawiania zostaną objaśnione kluczowe pojęcia oraz ich role.

2.4.1 Inicjalizacja

Faza inicjalizacji zachodzi, gdy po raz pierwszy kiedykolwiek korzystamy z aplikacji. Wobec tego aplikacja nie posiada żadnych danych ani lokalnie, ani na serwerze. W związku z tym konieczne jest wygenerowanie podstawowych sekretów.

Aplikacja wykonuje następujące kroki:

1. Tworzy reprezentację urządzenia – wygenerowanie kluczy na krzywej eliptycznej Curve25519.
2. Inicjuje pierwszą epokę.
3. Wygenerowanie kodu odzyskiwania i reprezentacji wirtualnego urządzenia – wygenerowanie kluczy wirtualnego urządzenia na krzywej eliptycznej Curve25519.
4. Wysyła do serwera:
   - klucze publiczne urządzenia,
   - zaszyfrowaną postać wirtualnego urządzenia,
   - zaszyfrowaną postać pierwszej epoki,
   - kody uwierzytelniające przynależność urządzenia i wirtualnego urządzenia do epoki (opisane w podsekcji 2.4.2).

Inicjacja pierwszej epoki polega na wygenerowaniu pierwszego klucza głównego **epochRootKey** jako losowego ciągu 32 bajtów, natomiast **epochSequenceId** jest ustawiany na **0**. Wartość **epochId** zostanie nadana później przez serwer, po wysłaniu danych z punktu 4.

Kolejnym istotnym krokiem jest wygenerowanie kodu odzyskiwania, który składa się z 40 znaków wybranych z alfabetu:

`ACDEFHJKLMNPQRSTUVWXYZ0123456789`

Alfabet ten został dobrany tak, aby wykluczyć występowanie homoglifów, czyli znaków o podobnym wyglądzie, które mogłyby wprowadzać użytkownika w błąd. Przykładowo, pominięto znak `I`, który może być podobny do `1`, Jeżeli użytkownik wpisze w kodzie odzyskiwania znak `I`, to zostanie on podmieniony na `1`.

Kod odzyskiwania składa się z:

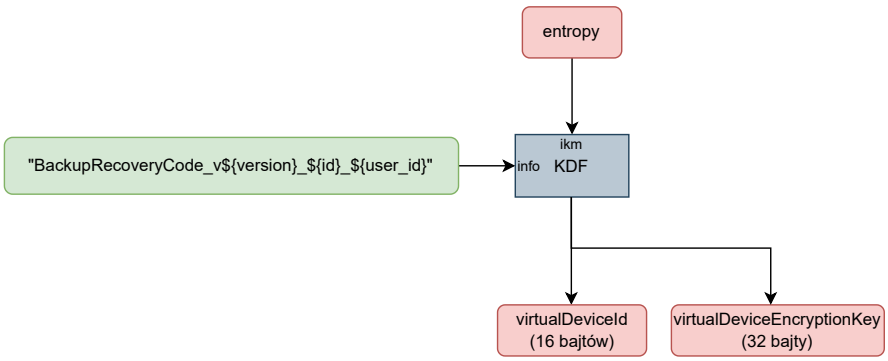
- 1 znak – wersja protokołu, pozwalająca na identyfikację odpowiedniej implementacji,
- 1 znak – identyfikator kodu (obecnie nieużywany, ustawiany na **0**),
- 34 znaki – losowy sekret określany w protokole jako **entropy**,
- 4 znaki – kod korekcji błędów, który umożliwia odzyskanie do 3 błędnie wprowadzonych znaków w części **entropy**.

Warto jednak zaznaczyć, że część kodu korekcji błędów została pominięta w mojej pracy, ponieważ uznano ją za zbędną w kontekście realizowanego rozwiązania.

Dzięki entropii zawartej w kodzie odzyskiwania można wyprowadzić klucz niezbędny do zaszyfrowania sekretów pierwszej epoki oraz reprezentacji wirtualnego urządzenia. Dane te są następnie przesyłane na serwer, co umożliwia ich późniejsze pobranie w celu odzyskania historii wiadomości.

Na rysunku 1 przedstawiono sposób wyprowadzenia:

- **virtualDeviceId** – identyfikator wirtualnego urządzenia, wymagany do pobrania jego reprezentacji,
- **virtualDeviceEncryptionKey** – klucz służący do zaszyfrowania symetrycznego pierwszej epoki i reprezentacji wirtualnego urządzenia.



Rysunek 1. Wyprowadzenie identyfikatora wirtualnego urządzenia i klucza szyfrującego jego reprezentację oraz pierwszą epokę, opracowanie własne

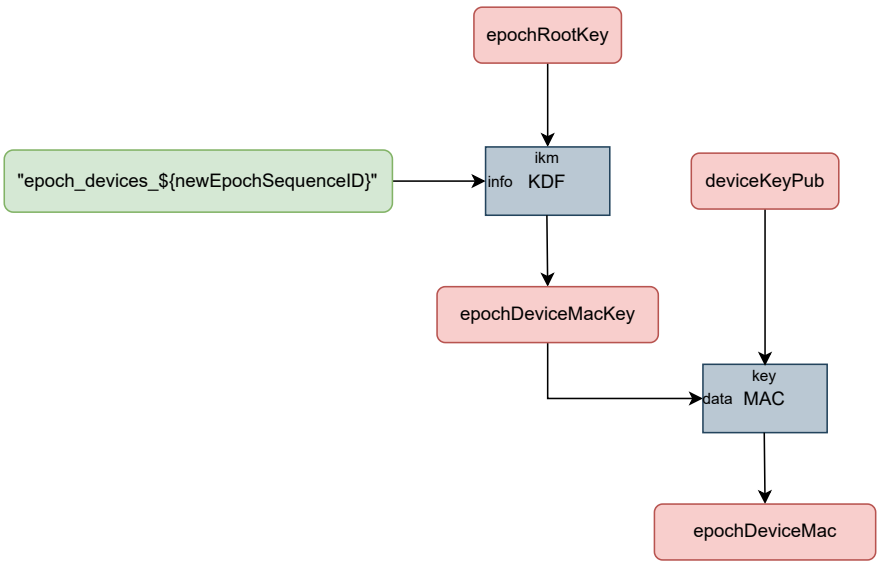
Warto zauważyć, że do wyprowadzenia tych informacji potrzebny jest również identyfikator użytkownika `user_id`, nadawany przez serwer podczas rejestracji użytkownika. Dzięki temu nawet jeśli dwóch różnych użytkowników wygeneruje ten sam kod odzyskiwania, ich klucze kryptograficzne będą unikalne.

Pierwsza epoka oraz reprezentacja urządzenia są szyfrowane symetrycznie uzyskanym kluczem i przesyłane na serwer, co pozwala na późniejsze wykorzystanie tych danych. Na zakończenie fazy inicjalizacji wymagane jest zapisanie pierwszej epoki oraz reprezentacji urządzenia lokalnie.

W fazie otwierania nowej epoki (opisanej w sekcji 2.4.5) do wirtualnego urządzenia przesyłany jest sekret pozwalający na dołączenie do nowo otwartej epoki. Jest to kluczowe, aby odzyskiwanie historii wiadomości należącej do różnych epok było możliwe. Należy jednak pamiętać, iż nie można przechowywać reprezentacji wirtualnego urządzenia. Pozostawienie jego reprezentacji skutkowałoby ominięciem mechanizmu tworzenia nowych epok do unieważnienia urządzeń użytkownika.

2.4.2 Uwierzytelnianie przynależności do epoki

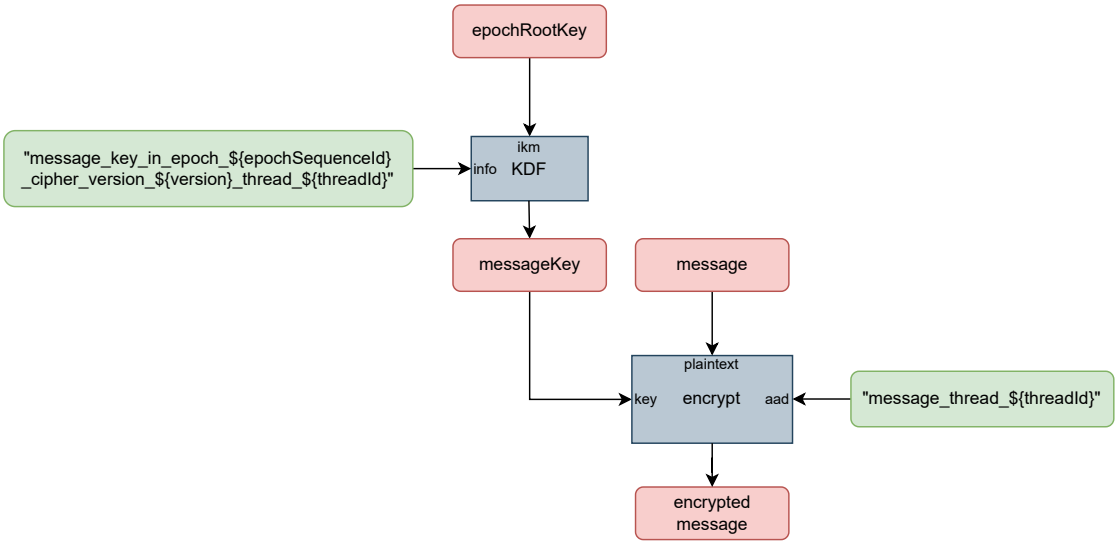
Każde z urządzeń musi uwierzytelnić swoją przynależność do epoki, tak aby aplikacja kliencka wykryła nieprawidłowość w przypadku podstawienia urządzenia przechowywanego na serwerze przez osobę niepożądaną. W tym celu tworzony jest kod uwierzytelniający, który pozwala innym urządzeniom na sprawdzenie, czy rzeczywiście urządzenie istniejące na serwerze należy do epoki. W protokole kod ten jest nazwany jako **epochDeviceMac**, a jego generacja została przedstawiona na rysunku 2.



Rysunek 2. Generacja kodu uwierzytelniającego przynależność do epoki, opracowanie własne

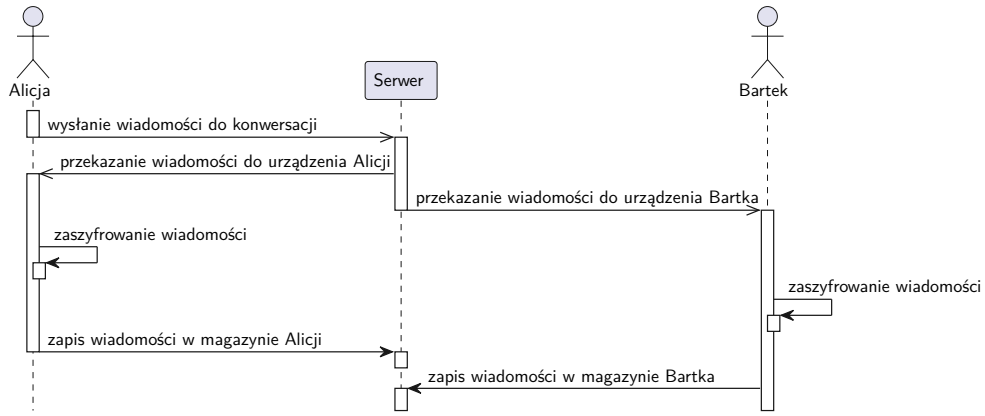
2.4.3 Zapisanie wiadomości po stronie serwera

Użytkownik po otrzymaniu wiadomości od innego użytkownika musi ją zaszyfrować, korzystając z klucza na podstawie danych aktualnej epoki. Szyfrowanie wiadomości zostało przedstawione na rysunku 3.



Rysunek 3. Szyfrowanie wiadomości, opracowanie własne

Przykładowy scenariusz przedstawia rysunek 4 ilustrujący przesłanie wiadomości przez Alicję do konwersacji, w której znajduje się ona i Bartek. W przypadku zapisu wiadomości na serwerze ważne jest, aby podać numer epoki **epochSequenceId** sygnalizujący, którymi sekretami epoki użytkownik szyfrował wiadomości.



Rysunek 4. Przesłanie wiadomości przez Alicję do Bartka, opracowanie własne

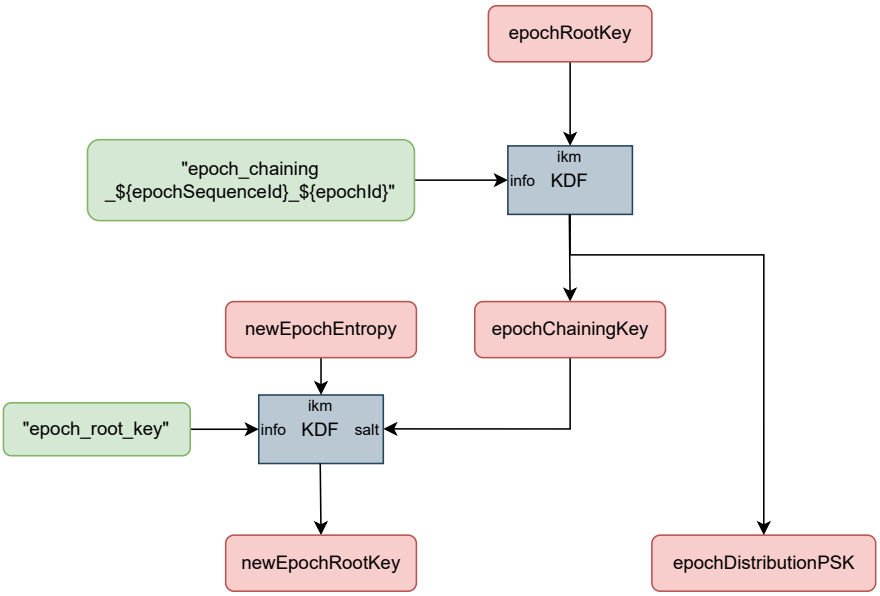
2.4.4 Pobranie historii korespondencji

Zapisane wiadomości użytkowników są przechowywane w ich dedykowanym magazynie. Po otworzeniu pożądanej konwersacji serwer zwraca wszystkie wiadomości wraz z załączonym **epochSequenceId** informującym o numerze epoki, w której poszczególne wiadomości zostały zaszyfrowane. Użytkownik przed odczytaniem tych wiadomości musi je odszyfrować uzyskanym kluczem tak samo jak w przypadku szyfrowania (patrz rysunek 3).

2.4.5 Utworzenie nowej epoki

W przypadku, gdy urządzenie pozostaje nieaktywne przez dłuższy czas, konieczna jest rotacja sekretów epoki. Jeśli jedno z urządzeń wykryje, że inne urządzenie przekroczyło dopuszczalny czas nieaktywności, musi ono zainicjować proces tworzenia nowej epoki.

Nowa epoka jest generowana na podstawie danych z poprzedniej, a kluczowym elementem tego procesu jest użycie entropii, oznaczonej jako **newEpochEntropy**. Wartość ta jest losowo generowana jako 32-bajtowy sekret. Na podstawie **newEpochEntropy** i danych poprzedniej epoki wyznaczany jest nowy klucz główny **newEpochRootKey**, proces ten przedstawia rysunek 5. Poza nowym kluczem głównym generowany jest również drugi klucz **epochDistributionPSK**, którego rola zostanie omówiona później.



Rysunek 5. Wyprowadzenie **newEpochRootKey** oraz **epochDistributionPSK**, opracowanie własne

Numer nowej epoki, oznaczony jako **newEpochSequenceId**, jest wyznaczany poprzez inkrementację numeru poprzedniej epoki zgodnie z równaniem:

```
newEpochSequenceId = epochSequenceId + 1
```

Z kolei identyfikator epoki **epochId** jest nadawany przez serwer podczas rejestracji nowej epoki.

Aby umożliwić innym urządzeniom na dołączenie do nowej epoki, konieczne jest udostępnienie sekretu **newEpochEntropy**. W celu zachowania bezpieczeństwa sekret nie może być przesłany w postaci jawnej. Z tego względu zastosowane zostało szyfrowanie asymetryczne, które pozwala na indywidualne szyfrowanie sekretu dla poszczególnych urządzeń z wykorzystaniem kluczy publicznych umieszczonych na serwerze aplikacji. Protokół wskazuje, aby do szyfrowania zastosować funkcję `labyrinth-hpke-encrypt`, której użycie przedstawione jest na listingu 2.

```
encryptedNewEpochEntropy = labyrinth_hpke_encrypt(
    recipient_enc_pub = recipientEpochStorageKeyPub,
    sender_auth_pub = senderEpochStorageAuthKeyPub,
    sender_auth_priv = senderEpochStorageAuthKeyPriv,
    psk = epochDistributionPSK,
    aad = "epoch_${newEpochSequenceId}",
    plaintext = newEpochEntropy
)
```

Listing 2. Szyfrowanie asymetryczne **newEpochEntropy**

Jak widać, jednym z parametrów użytych podczas szyfrowania jest **epochDistributionPSK** zdefiniowany wcześniej na rysunku 5. Pełni on funkcję dodatkowego zabezpieczenia, które wymusza konieczność posiadania danych z poprzedniej epoki do odszyfrowania **newEpochEntropy**.

W procesie muszą być uwzględnione jedynie urządzenia, które nie przekroczyły czasu maksymalnej nieaktywności. W ten sposób eliminuje się ryzyko, że osoba, która pozyska dane z nieaktywnego urządzenia należącego do poprzednich epok, uzyska dostęp do nowszych epok. Dodatkową warstwą bezpieczeństwa jest weryfikacja:

- **epochDeviceMac** – pozwala na sprawdzenie czy urządzenie uwierzytelniło swoją przynależność do aktualnej epoki, wobec tego nie narażamy protokołu na podstawienie publicznej reprezentacji urządzenia przez osobę niepożądaną (opisane w sekcji 2.4.2),
- **epochStorageKeyPub** – weryfikowany jest podpis klucza w celu wykrycia potencjalnego podstawienia przez osobę trzecią.

W szczególności **newEpochEntropy** musi zostać zawsze wysłane do wirtualnego urządzenia, w przeciwnym przypadku niemożliwe będzie odtworzenie sekretów nowej epoki na nowo dodanym urządzeniu.

Ważnym aspektem jest konieczność podania pary kluczy uwierzytelniających do szyfrowania, dzięki tej własności `labyrinth_hpke_encrypt` niemożliwe jest rozsyłanie **newEpochEntropy** przy użyciu wirtualnego urządzenia. W związku z tym wirtualne urządzenie może służyć tylko do odczytu sekretów epok.

2.4.6 Dołączenie do nowej epoki

Gdy aplikacja zostanie powiadomiona o utworzeniu nowej epoki przez inne urządzenie, musi ona pobrać zaszyfrowaną entropię nowej epoki **newEpochEntropyEncrypted**. Następnie odszyfrowuje entropię przy użyciu własnych kluczy (publicznych i prywatnych), a także publicznych urządzenia, które utworzyło nową epokę. W tym celu wykorzystano funkcję `labyrinth_hpke_decrypt` przedstawionej na listingu 3. Sekret **epochDistributionPSK** jest wyprowadzany w ten sam sposób co

```
decryptedNewEpochEntropy = labyrinth_hpke_decrypt(
    recipient_enc_pub = recipientEpochStorageKeyPub,
    recipient_enc_priv = recipientEpochStorageKeyPriv,
    sender_auth_pub = senderEpochStorageAuthKeyPub,
    psk = epochDistributionPSK,
    aad = "epoch_${newEpochSequenceId}",
    ciphertext = newEpochEntropyEncrypted,
)
```

Listing 3. Szyfrowanie asymetryczne **newEpochEntropy**

w przypadku szyfrowania, a jego wyprowadzenie zostało omówione wcześniej na rysunku 5 w podsekcji 2.4.5.

2.4.7 Odzyskanie wszystkich epok na nowym urządzeniu

W sytuacji, gdy użytkownik przeszedł już proces inicjalizacji i chce uzyskać dostęp do historii wiadomości na nowym urządzeniu, musi przeprowadzić proces odzyskiwania wszystkich epok aż do najnowszej. Proces ten rozpoczyna się od wprowadzenia kodu odzyskiwania, który umożliwia uzyskanie **virtualDeviceId** oraz **virtualDeviceEncryptionKey**. Operacja ta zachodzi identycznie jak na wcześniej przedstawionym rysunku 1.

Identyfikator wirtualnego urządzenia **virtualDeviceId** pozwala na pobranie jego zaszyfrowanej reprezentacji oraz sekretów pierwszej epoki, natomiast **virtualDeviceEncryptionKey** umożliwia ich odszyfrowanie. Po pomyślnym odszyfrowaniu danych pierwszej epoki następuje proces odzyskiwania kolejnych epok w sposób łańcuchowy – każda kolejna epoka jest rekonstruowana przy użyciu sekretów z epoki poprzedniej w sposób opisany w sekcji 2.4.6. Dzięki temu użytkownik może stopniowo uzyskać dostęp do wszystkich nowszych epok, kontynuując proces od pierwszej aż do najaktualniejszej.

Rozdział 3

Projekt implementacji biblioteki

Przed implementacją rozwiązania przeanalizowałem protokół Labyrinth oraz opracowałem streszczenie protokołu przedstawione w rozdziale 2. Element planowania był szczególnie kluczowy, ponieważ niezrozumienie którejś z faz protokołu mogło skutkować dużymi zmianami w kodzie.

3.1 Wybrane technologie

Biblioteka została napisana w języku **TypeScript** z uwagi na statyczne typowanie, które jest szczególnie przydatne w większych projektach. Język ten jest transpilowany do **JavaScript**, który jest natywnie obsługiwany w przeglądarkach internetowych. Oprócz tego wykorzystywane są rozwiązania kryptograficzne, w szczególności:

- Web Crypto API[1] – interfejs natywnie wspierany przez wszystkie współczesne przeglądarki,
- noble-curves[2] – popularna biblioteka napisana w `JavaScript` pozwalająca na obsługę operacji na krzywych eliptycznych.

3.2 Struktury w aplikacji klienckiej

Aby efektywnie zarządzać kodem i uniknąć błędów w trakcie implementacji, stworzyłem klasy odpowiadające najważniejszym elementom protokołu. Wśród nich można wyróżnić:

- klucz publiczny i klucz prywatny,
- urządzenie,
- urządzenie wirtualne.

[1] <https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API>
[2] <https://github.com/paulmillr/noble-curves>

3.2.1 Klucz publiczny i klucz prywatny

W ramach protokołu wielokrotnie korzystamy z pary kluczy zarówno urządzenia, jak i wirtualnego urządzenia. Klucze reprezentowane są przez klasy **PublicKey** i **PrivateKey**, a ich najważniejsze metody przedstawia diagram 6.

Rysunek 6. Diagram klas reprezentujący parę kluczy

Jako wewnętrzną implementację pary kluczy wykorzystuję implementację zgodną ze specyfikacją RFC8032 [3] wykorzystującą krzywą **edwards25519**. Generowanie pary kluczy przedstawia listing 4.

```
class PrivateKey {
    private readonly ed25519PrivateKey: Uint8Array;

    private constructor(ed25519PrivateKey: Uint8Array) {
        cryptoAssert(ed25519PrivateKey.length === KEY_LENGTH_BYTES);
        this.ed25519PrivateKey = ed25519PrivateKey;
    }

    public static generate(): PrivateKey {
        return new PrivateKey(ed25519.utils.randomPrivateKey());
    }

    public getPublicKey(): PublicKey {
        return new PublicKey(ed25519.getPublicKey(this.ed25519PrivateKey));
    }
}

/* generacja klucza prywatnego */
const kluczPrywatny = PrivateKey.generate();
/* generacja klucza publicznego */
const kluczPubliczny = kluczPrywatny.getPublicKey();
```

Listing 4. Generacja pary kluczy

Implementacja krzywej **edwards25519** pozwala na podpis kluczem prywatnym metodą `sign` i weryfikację podpisu kluczem publicznym metodą `verify`, ale nie wspiera ustalania wspólnego sekretu. Zamiast generować dodatkową parę kluczy zgodnych z krzywą **Curve25519**, klucz **edwards25519** może zostać przekonwertowany na **Curve25519**, co upraszcza kod. Klasa **PrivateKey** przechowuje klucz w postaci **edwards25519** i dokonuje konwersji na **Curve25519** dzięki matematycznemu powiązaniu [11] między wykorzystywanymi krzywymi eliptycznymi.

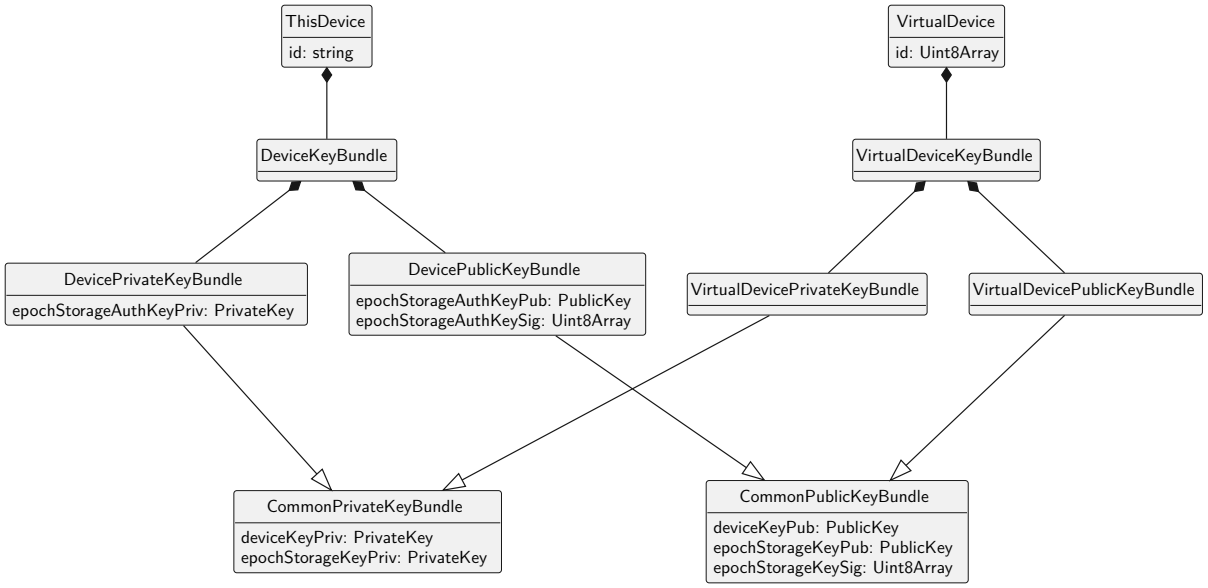
Przykład konwersji z wykorzystaniem funkcji `edwardsToMontgomeryPriv`, zapewnianej przez bibliotekę **noble-curves**, został przedstawiony na listingu 5.

```
class PrivateKey {
    /* implementacja pozostałych metod... */

    public agree(otherKey: PublicKey): Uint8Array {
        // konwersja klucza prywatnego ed25519 na x25519
        const x25519PrivateKey = edwardsToMontgomeryPriv(
            this.ed25519PrivateKey,
        );
        const x25519OtherPublicKey = otherKey.getX25519PublicKeyBytes();
        return x25519.getSharedSecret(x25519PrivateKey, x25519OtherPublicKey);
    }

    /* implementacja pozostałych metod... */
}
```

Listing 5. Ustalanie wspólnego klucza

3.2.2 Reprezentacja urządzenia i urządzenia wirtualnego

Zarówno urządzenie (w postaci klasy **ThisDevice**), jak i urządzenie wirtualne (**VirtualDevice**) składa się z zestawów kluczy i identyfikatora. Reprezentacje urządzeń prezentuje rysunek 7.



Rysunek 7. Diagram klas przedstawiający reprezentacje urządzenia i urządzenia wirtualnego

Różnicą pomiędzy tymi typami urządzeń jest brak kluczy uwierzytelniających **epochStorageAuthKeyPriv** i **epochStorageAuthKeyPub**. Wydzielono również klasę współdzielącą te same typy

kluczy, aby pokazać podobieństwo pomiędzy dwoma typami urządzeń i zredukować powtarzający się w klasach kod.

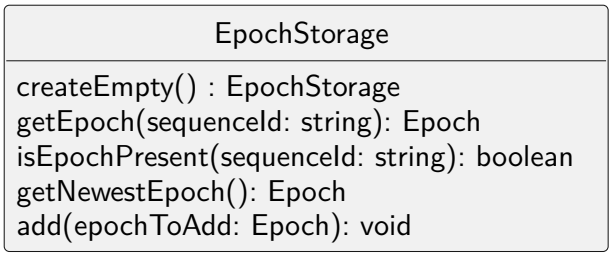
3.2.3 Reprezentacja epoki

Na listingu 6 epoka została przedstawiona jako prosty typ do przechowywania danych.

```
export type Epoch = {
    id: string;
    sequenceId: string;
    rootKey: Uint8Array;
};
```

Listing 6. Reprezentacja epoki

Dodatkowo stworzona została klasa **EpochStorage** służąca jako struktura danych do przechowywania epok. Dodatkową jej funkcją jest walidacja, czy epoki są dodawane w odpowiedniej kolejności oraz, czy w momencie dodawania nie istnieje już jej reprezentacja. Rysunek 8 ilustruje wybrane najważniejsze metody udostępnione przez klasę.



Rysunek 8. Diagram klas przedstawiający metody klasy **EpochStorage**

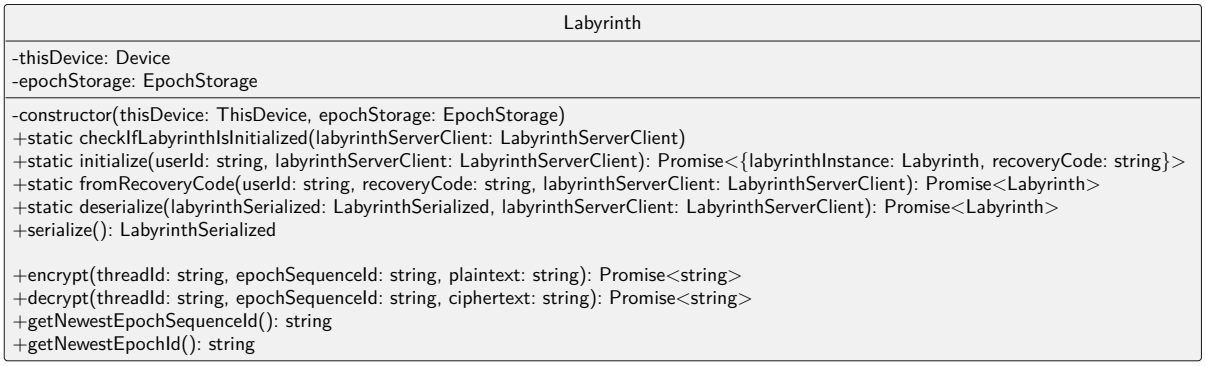
3.3 Interfejs biblioteki

W projektowaniu biblioteki bardzo ważne jest zapewnienie intuicyjnego interfejsu, z którego mogą korzystać programiści. W tym celu posłużyłem się wzorcem projektowym fasady [5], który zakłada istnienie punktu dostępowego udostępniającego jedynie operacje, które będą używane przez inny obiekt (w tym przypadku aplikację kliencką). Do zaprojektowania interfejsu kluczowe było zidentyfikowanie przypadków użycia aplikacji i faz protokołu, które powinny zostać wykonane:

- pierwsze logowanie urządzenia niezarejestrowanego – konieczna faza inicjalizacji,
- ponowne logowanie – potrzebne odczytanie danych przechowywanych lokalnie i sprawdzenie, czy w czasie nieaktywności powstała nowa epoka,
- logowanie na nowym urządzeniu – odzyskanie dostępu do epok przy użyciu kodu odzyskiwania,
- zaszyfrowanie wiadomości,

- odszyfrowanie wiadomości.

Na rysunku 9 przedstawiona jest klasa **Labyrinth**, która udostępnia metody realizujące zdefiniowane przypadki powyżej. Warto podkreślić, iż mechanizmy zarządzania epokami zostały ukryte, dzięki czemu biblioteka jest mniej skomplikowana w obsłudze.



Rysunek 9. Diagram przedsawiający klasę **Labyrinth**

Klasa **Labyrinth** posiada prywatny konstruktor, zatem, aby stworzyć obiekt, należy skorzystać ze statycznej metody tworzącej obiekt w zależności od przypadku użycia.

W fazie inicjalizacji skorzystamy z metody `initialize`, podczas odzyskiwania dostępu do epok z metody `fromRecoveryCode`, natomiast w przypadku ponownego logowania na zarejestrowanym urządzeniu z `deserialize`.

Do poprawnego działania biblioteki należy zapewnić komunikację z serwerem. Programista może według swoich potrzeb wybrać bibliotekę oraz standard komunikacji z serwerem, dlatego nie można założyć jej konkretnej implementacji. Aby oddzielić logikę komunikacji z serwerem od implementacji biblioteki, zdefiniowany został interfejs, do którego musi się dostosować programista korzystający z biblioteki. Na listingu 7 zdefiniowany został typ, który stanowi kontrakt, jakie dane ma wysłać klient i zwrócić serwer. Wybrane poszczególne mniejsze typy, które razem stanowią ten kontrakt, zostaną przedstawione w następnej sekcji.

```
1 export type LabyrinthServerClient = OpenFirstEpochServerClient &
2     OpenNewEpochBasedOnCurrentServerClient &
3     JoinEpochServerClient &
4     GetVirtualDeviceRecoverySecretsServerClient &
5     AuthenticateDeviceToEpochServerClient &
6     CheckIfLabyrinthIsInitializedServerClient &
7     AuthenticateDeviceToEpochAndRegisterDeviceServerClient &
8     NotifyAboutDeviceActivityServerClient &
9     CheckIfAnyDeviceExceedInactivityLimitServerClient;
```

Listing 7. Definicja kontraktu klienta z serwerem

3.4 Szczegóły implementacji

W tym rozdziale przedstawione zostaną fragmenty implementacji najważniejszych elementów protokołu z podziałem na przypadki użycia biblioteki w typowej aplikacji.

3.4.1 Pierwsze korzystanie z systemu

W sytuacji pierwszego korzystania z systemu przez użytkownika następuje faza inicjalizacji – biblioteka zajmuje się stworzeniem pierwszej epoki oraz przekazaniem serwerowi reprezentacji urządzenia i urządzenia wirtualnego. Komunikację z serwerem w ramach fazy inicjalizacji realizuje kontrakt przedstawiony na listingu 8.

```
1  export type OpenFirstEpochBody = {
2      virtualDeviceId: string;
3      virtualDeviceEncryptedRecoverySecrets:
       ↪ VirtualDeviceEncryptedRecoverySecretsSerialized;
4      virtualDevicePublicKeyBundle: VirtualDevicePublicKeyBundleSerialized;
5      devicePublicKeyBundle: DevicePublicKeyBundleSerialized;
6      firstEpochMembershipProof: {
7          epochDeviceMac: string;
8          epochVirtualDeviceMac: string;
9      };
10 };
11
12 export type OpenFirstEpochResponse = {
13     deviceId: string;
14     epochId: string;
15 };
16
17 export type OpenFirstEpochServerClient = {
18     openFirstEpoch: (
19         requestBody: OpenFirstEpochBody,
20     ) => Promise<OpenFirstEpochResponse>;
21 };
```

Listing 8. Kontrakt pomiędzy klientem a serwerem podczas tworzenia pierwszej epoki

Kod obsługujący inicjalizację przedstawia listing 9. Metoda `initialize` przyjmuje dwa argumenty:

- **userId** – identyfikator użytkownika, potrzebny jako składnik do zaszyfrowania wirtualnego urządzenia
- **labyrinthServerClient** – implementacja klienta służącego do komunikacji z serwerem.

W ramach implementacji tworzone jest wirtualne urządzenie (linia 8) oraz urządzenie wraz z pierwszą epoką (linia 11).

```
class Labyrinth {
    public static async initialize(
        userId: string,
        labyrinthServerClient: LabyrinthServerClient,
    ) {
        const { virtualDevice, virtualDeviceDecryptionKey, recoveryCode } =
            await VirtualDevice.initialize(userId);

        const { firstEpoch, thisDevice } = await ThisDevice.initialize(
            virtualDevice,
            virtualDeviceDecryptionKey,
            labyrinthServerClient,
        );

        const epochStorage = EpochStorage.createEmpty();
        epochStorage.add(firstEpoch);

        const labyrinthInstance = new Labyrinth(thisDevice, epochStorage);

        return {
            labyrinthInstance,
            recoveryCode,
        };
    }
}
```

Listing 9. Metoda inicjalizująca Labyrinth

Fragment implementacji tworzenia wirtualnego urządzenia przedstawia listing 10. W pierwszej kolejności generowany jest kod odzyskiwania, na którego podstawie generowany jest identyfikator urządzenia oraz klucz potrzebny do zaszyfrowania reprezentacji wirtualnego urządzenia.

```
const recoveryCode = generateRecoveryCode();
const { virtualDeviceId, virtualDeviceDecryptionKey } =
    await deriveVirtualDeviceIdAndDecryptionKey(userId, recoveryCode);
```

Listing 10. Generacja kodu odzyskiwania i wyprowadzenie klucza szyfrującego

Z kolei inicjalizacja urządzenia jest przedstawiona na listingu 11. Urządzenie jest odpowiedzialne za stworzenie pierwszej epoki oraz zaszyfrowanie jej wraz z reprezentacją wirtualnego urządzenia. Czynności te realizowane są w ramach funkcji `openFirstEpoch`. Oprócz tego funkcja wysyła zaszyfrowane dane, korzystając ze zdefiniowanego klienta `openFirstEpochServerClient`.

```
class ThisDevice {
    public static async initialize(
        virtualDevice: VirtualDevice,
        virtualDeviceDecryptionKey: Uint8Array,
        openFirstEpochServerClient: OpenFirstEpochServerClient,
    ) {
        const deviceKeyBundle = DeviceKeyBundle.generate();

        const { deviceId, firstEpoch } = await openFirstEpoch(
            deviceKeyBundle.pub,
            virtualDeviceDecryptionKey,
            virtualDevice,
            openFirstEpochServerClient,
        );

        const thisDevice = new ThisDevice(deviceId, deviceKeyBundle);

        return {
            thisDevice,
            firstEpoch,
        };
    }
}
```

Listing 11. Inicjalizacja urządzenia

3.4.2 Ponowne logowanie do aplikacji

Podczas ponownego logowania do aplikacji muszą zostać załadowane dane z lokalnej pamięci urządzenia. Na listingu 12 przedstawiony został kod tworzący instancję klasy **Labyrinth**. Po odtworzeniu zapisanych lokalnie danych sprawdzane jest, czy w czasie nieaktywności urządzenia powstała nowa epoka i jeżeli tak, to następuje dołączenie do nowszych epok. Za proces ten odpowiedzialna jest reprezentacja urządzenia (`ThisDevice`). Następnie urządzenie powiadamia serwer o swojej aktywności i odpytuje, czy jakiekolwiek z urządzeń przekroczyło maksymalny czas nieaktywności. Jeżeli taka sytuacja nastąpi, tworzona jest nowa epoka bez unieważnionego urządzenia.

```
class Labyrinth {
    public static async deserialize(
        labyrinthSerialized: LabyrinthSerialized,
        labyrinthServerClient: LabyrinthServerClient,
    ): Promise<Labyrinth> {
        const {
            thisDevice: thisDeviceSerialized,
            epochStorage: epochStorageSerialized,
        } = labyrinthSerialized;
        const epochStorage = EpochStorage.deserialize(epochStorageSerialized);
        const thisDevice = await ThisDevice.deserialize(
            thisDeviceSerialized,
            epochStorage,
            labyrinthServerClient,
        );

        await labyrinthServerClient.notifyAboutDeviceActivity(thisDevice.id);
        await checkIfAnyDeviceExceededInactivityLimitAndOpenNewEpochIfNeeded(
            labyrinthServerClient,
            thisDevice,
            epochStorage,
        );
        return new Labyrinth(thisDevice, epochStorage);
    }
}
```

Listing 12. Tworzenie obiektu **Labyrinth** przy ponownym logowaniu do aplikacji

Dołączanie do nowych epok polega na ciągłym wyprowadzaniu kolejnej epoki na podstawie poprzedniej i **newEpochEntropy** danej epoki. Czynność ta jest powtarzana do momentu, aż dołączymy do najnowszej epoki. Fragment kodu realizujący ten proces jest przedstawiony na listingu 13.

```
const { newestEpochSequenceId } =
    await serverClient.getNewestEpochSequenceId();

let newestKnownEpoch = epochStorage.getNewestEpoch();

while (newestEpochSequenceId !== newestKnownEpoch.sequenceId) {
    newestKnownEpoch = await joinNewerEpoch(
        deviceKeyBundle,
        newestKnownEpoch,
        serverClient,
    );
    epochStorage.add(newestKnownEpoch);
}
```

Listing 13. Łancuchowe odtwarzanie kolejnych epok

Następnie w ramach funkcji `joinNewerEpoch` pobierane jest **encryptedNewEpochEntropy** wraz z identyfikatorem epoki i zestawem kluczy publicznych urządzenia, które tworzyło daną epokę. Dodatkowo weryfikowany jest podpis cyfrowy kluczy, aby uniknąć ich niezgodności. W przypadku niepowodzenia rzucany jest wyjątek `InvalidEpochStorageAuthKey`. Czynności te realizuje fragment kodu przedstawiony na listingu 14.

```
const newerEpochSequenceId = (
        BigInt(newestKnownEpoch.sequenceId) + 1n
    ).toString();
const {
    epochId: newerEpochId,
    encryptedEpochEntropy: encryptedNewerEpochEntropy,
    senderDevicePublicKeyBundle,
} = await joinEpochWebClient.getNewerEpochJoinData(newerEpochSequenceId);

const senderDevice = DevicePublicKeyBundle.deserialize(
    senderDevicePublicKeyBundle,
);
const isValidEpochStorageAuthKey = senderDevice.deviceKeyPub.verify(
    senderDevice.epochStorageAuthKeySig,
    Uint8Array.of(0x31),
    senderDevice.epochStorageAuthKeyPub.getX25519PublicKeyBytes(),
);
if (!isValidEpochStorageAuthKey) {
    throw new InvalidEpochStorageAuthKey();
}
```

Listing 14. Pobranie sekretów do dołączenia do nowej epoki

Po udanej weryfikacji klucza odszyfrowywane jest **encryptedNewEpochEntropy**, które pozwala na odzyskania sekretów nowej epoki. Fragment przedstawia listing 15.

```
const [newerEpochChainingKey, newerEpochDistributionPreSharedKey] =
    await kdf_two_keys(
        newestKnownEpoch.rootKey,
        null,
        asciiStringToBytes(
            `epoch_chaining_${newestKnownEpoch.sequenceId}
            _${newestKnownEpoch.id}`,
        ),
    );

const newerEpochEntropy = await labyrinth_hpke_decrypt(
    deviceKeyBundle.pub.epochStorageKeyPub,
    deviceKeyBundle.priv.epochStorageKeyPriv,
    senderDevice.epochStorageAuthKeyPub,
    newerEpochDistributionPreSharedKey,
    asciiStringToBytes(`epoch_${newerEpochSequenceId}`),
    BytesSerializer.deserialize(encryptedNewerEpochEntropy),
);
const newerEpochRootKey = await kdf_one_key(
    newerEpochEntropy,
    newerEpochChainingKey,
    asciiStringToBytes('epoch_root_key'),
);

return {
    id: newerEpochId,
    sequenceId: newerEpochSequenceId,
    rootKey: newerEpochRootKey,
} as Epoch;
```

Listing 15. Dołączenie do nowej epoki

3.4.3 Logowanie z nowego urządzenia

Po zalogowaniu do aplikacji z nowego urządzenia należy odzyskać sekrety wszystkich epok utworzonych do tej pory. W tym celu wykorzystywany jest kod odzyskiwania, który pozwala na odzyskanie wirtualnego urządzenia wraz z pierwszą epoką.

Kolejne czynności potrzebne na odzyskanie dostępu do historii wiadomości są wykonywane przez funkcję `fromRecoveryCode` przedstawioną na listingu 16. Najpierw odtwarzana jest pierwsza epoka wraz z wirtualnym urządzeniem, a następnie na podstawie pierwszej epoki odzyskiwane są kolejne z wykorzystaniem wirtualnego urządzenia (linia 16). W linii 23 tworzona jest reprezentacja

urządzenia, która jednocześnie uwierzytelnia swoją przynależność do najnowszej epoki. Na koniec urządzenie powiadamia serwer o swojej aktywności i sprawdza, czy którekolwiek z innych urządzeń jest nieaktywne. Jeżeli tak to tworzona zostanie nowa epoka bez urządzenia nieaktywnego.

```
class Labyrinth {
    public static async fromRecoveryCode(
        userId: string,
        recoveryCode: string,
        labyrinthServerClient: LabyrinthServerClient,
    ): Promise<Labyrinth> {
        const { virtualDevice, epoch } = await VirtualDevice.fromRecoveryCode(
            userId,
            recoveryCode,
            labyrinthServerClient,
        );

        const epochStorage = EpochStorage.createEmpty();
        epochStorage.add(epoch);

        await joinAllEpochs(
            virtualDevice.keyBundle,
            epochStorage,
            labyrinthServerClient,
        );

        const newestRecoveredEpoch = epochStorage.getNewestEpoch();
        const thisDevice = await ThisDevice.fromRecoveryCode(
            newestRecoveredEpoch,
            labyrinthServerClient,
        );

        await labyrinthServerClient.notifyAboutDeviceActivity(thisDevice.id);
        await checkIfAnyDeviceExceededInactivityLimitAndOpenNewEpochIfNeeded(
            labyrinthServerClient,
            thisDevice,
            epochStorage,
        );
        return new Labyrinth(thisDevice, epochStorage);
    }
}
```

Listing 16. Odzyskiwanie epok na nowym urządzeniu

Część metody `VirtualDevice.fromRecoveryCode` odzyskującej pierwszą epokę i reprezentację wirtualnego urządzenia jest przedstawiona na listingu 17. Najpierw wyprowadzane są:

- `virtualDeviceId` – identyfikator wirtualnego urządzenia, który posłuży do pobrania zaszyfrowanej reprezentacji z serwera,
- `virutalDeviceDecryptionKey` – klucz potrzebny do odszyfrowania wirtualnego urządzenia i pierwszej epoki.

Następnie w linii 8 pobierana jest zaszyfrowana reprezentacja wirtualnego urządzenia i pierwszej epoki. Ostatnim istotnym krokiem jest odszyfrowanie wirtualnego urządzenia i epoki w linii 13. Dzięki

```
1  const { virtualDeviceId, virtualDeviceDecryptionKey } =
2      await deriveVirtualDeviceIdAndDecryptionKey(userId, recoveryCode);
3
4  const {
5      epochId,
6      virtualDeviceEncryptedRecoverySecrets,
7      expectedVirtualDevicePublicKeyBundle,
8  } = await webClient.getVirtualDeviceRecoverySecrets({
9      virtualDeviceId: BytesSerializer.serialize(virtualDeviceId),
10 });
11
12 const { virtualDeviceKeyBundle, epochWithoutId } =
13     await decryptVirtualDeviceRecoverySecrets(
14         virtualDeviceDecryptionKey,
15         VirtualDeviceEncryptedRecoverySecrets.deserialize(
16             virtualDeviceEncryptedRecoverySecrets,
17         ),
18         VirtualDevicePublicKeyBundle.deserialize(
19             expectedVirtualDevicePublicKeyBundle,
20         ),
21     );
22
23 const epoch = {
24     id: epochId,
25     ...epochWithoutId,
26 };
27
28 const virtualDevice = new VirtualDevice(
29     virtualDeviceId,
30     virtualDeviceKeyBundle,
31 );
32
33 return {
34     virtualDevice,
35     epoch,
36 };
```

Listing 17. Odzyskanie pierwszej epoki i wirtualnego urządzenia

wirtualnemu urządzeniu i pierwszej epoce, możemy odzyskać kolejne epoki, jeśli istnieją. Proces zachodzi w ten sam sposób co w poprzedniej podsekcji z wykorzystaniem funkcji `joinAllEpochs`, jedyną różnicą jest podanie jako argument wirtualnego urządzenia zamiast urządzenia.

Rozdział 4

Testy

W niniejszym rozdziale zostaną przedstawione testy biblioteki. Z uwagi na jeden z celów, jakim było stworzenie intuicyjnej biblioteki, stworzyłem aplikację czatu tekstowego wykorzystującą ją w praktyce. W ten sposób mogłem zweryfikować, czy rozwiązanie jest wygodne w użytkowaniu i działa poprawnie w scenariuszach typowych dla tego typu aplikacji.

4.1 Architektura aplikacji

Aplikacja składa się z trzech warstw:

- aplikacji klienckiej – napisana w **TypeScript** z wykorzystaniem biblioteki **ReactJS**,
- serwera – napisanego w języku **Java** z wykorzystaniem szkieletu aplikacji **Spring Boot**,
- bazy danych – wybrany silnik bazodanowy do długoterminowego przechowywania danych to **PostgreSQL**, z kolei do tymczasowego przechowywania wiadomości wysłanych do nieaktywnych użytkowników wykorzystano silnik **Redis**.

4.1.1 Aplikacja kliencka

Aplikacja kliencka realizuje funkcjonalności takie jak:

- logowanie,
- stworzenie konwersacji z użytkownikami,
- wysłanie wiadomości,
- odebranie wiadomości w czasie rzeczywistym,
- odebranie wiadomości po nieaktywności w systemie,
- obsługę protokołu **Labyrinth**.

Aplikacja zarządza stanem, a w szczególności na bieżąco zapisuje sekrety lokalnie, aby móc je ponownie wykorzystać przy kolejnym uruchomieniu. Dodatkowo aplikacja implementuje interfejs **LabyrinthServerClient** (opisany w podsekcji 3.3), po to, aby biblioteka mogła komunikować się z serwerem według potrzeb. Listing 18 przedstawia przykładową obsługę fazy protokołu, jaką jest otworzenie nowej epoki na podstawie poprzedniej. Na etapie implementacji według kontraktu

wykorzystywane są już konkretne rozwiązania i technologię. Komunikacja odbywa się poprzez Hypertext Transfer Protocol (HTTP) z wykorzystaniem biblioteki axios[1].

```
const labyrinthServicePrefix = '/api/labyrinth-service';
/* implementacja pozostałych pomniejszych interfejsów */
const openNewEpochBasedOnCurrentServerClient:
↪  OpenNewEpochBasedOnCurrentServerClient =
    {
        getDevicesInEpoch: async (epochId) =>
            (
                await httpClient.get<GetDevicesInEpochResponse>(
                    `${labyrinthServicePrefix}/epochs/${epochId}/devices`,
                )
            ).data,
        openNewEpochBasedOnCurrent: async (currentEpochId, requestBody) =>
            (
                await httpClient.post<OpenNewEpochBasedOnCurrentResponse>(
                    `${labyrinthServicePrefix}/epochs/open-based-on-current/
                        ${currentEpochId}`,
                    requestBody,
                )
            ).data,
    };
/* implementacja pozostałych pomniejszych interfejsów */
export default const labyrinthWebClientImpl = {
    //
    ...openNewEpochBasedOnCurrentServerClient,
    //
} as LabyrinthServerClient;
```

Listing 18. Przykład implementacji komunikacji w ramach otwierania nowej epoki

4.1.2 Serwer

Główną rolą serwera jest obsługa funkcjonalności aplikacji klienckiej. Architektura serwera została zaprojektowana według wzorca modularnego monolitu. W ramach serwera wyróżniono cztery moduły aplikacji:

- moduł przechowujący informacje o użytkownikach,
- moduł uwierzytelniania i autoryzacji,
- moduł czatu pozwalający na przekazywanie wiadomości i obsługę konwersacji,
- moduł obsługujący operacje związane z protokołem **Labyrinth**.

[1] Axios – biblioteka do żądań HTTP

Moduł obsługi protokołu **Labyrinth** nie jest skomplikowany, ponieważ jego jedyną funkcją jest zapis danych wysyłanych przez aplikację kliencką oraz ich odczyt na żądanie.

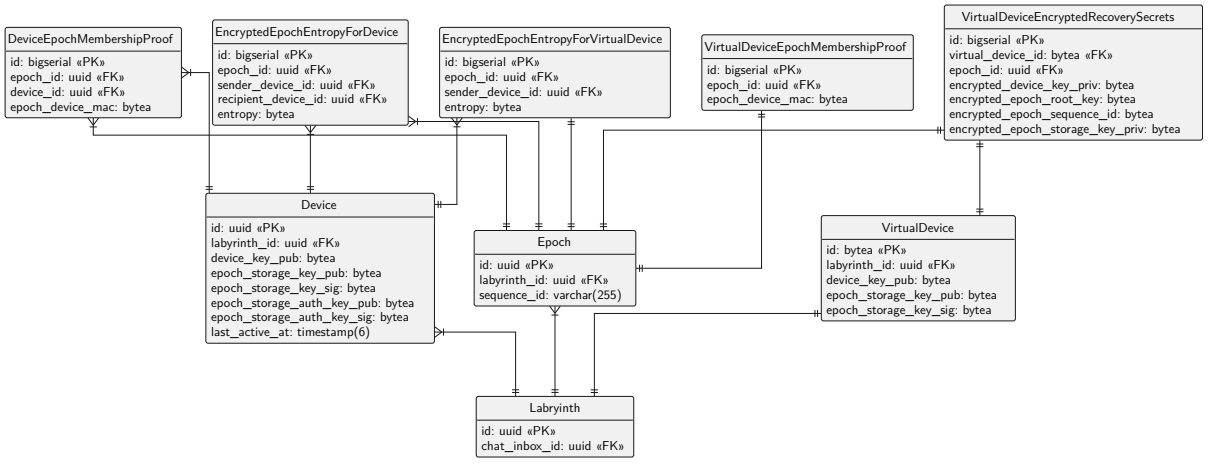
Aby zapewnić, że każda z wiadomości zostanie poprawnie zapisana, zastosowano strategię opisaną scenariuszem:

1. Użytkownik wysyła wiadomość do konwersacji.
2. Serwer odbiera wiadomość i zapisuje ją tymczasowo dla każdego użytkownika w bazie **Redis**.
3. Serwer przekazuje wiadomości do użytkowników w konwersacji.
4. Aplikacja kliencka odbiera wiadomość, zaszyfrowuje ją używając biblioteki obsługującej **Labyrinth**, a następnie wysyła do serwera do przechowania długoterminowego.
5. Serwer po udanym zapisie zaszyfrowanej wiadomości do bazy długoterminowej usuwa jej odpowiednik z bazy tymczasowej.

Dzięki temu pewne jest, że żadna z wiadomości nie zostanie stracona.

4.1.3 Baza danych

Do poprawnego działania protokołu **Labyrinth** serwer musi pozwalać na zapis danych. W tym celu zaprojektowano bazę danych do danych potrzebnych w protokole Labyrinth składającą się z 9 tabel. Generowane są one przy pomocy skryptu **SQL**, a ostateczna postać tabel została przedstawiona na diagramie Entity Relationship Diagram (ERD) 10 w notacji Information Engineering (IE) opisanej w publikacji [8], nazywaną potocznie notacją „kurzych łapek".



Rysunek 10. Diagram ERD przedstawiający tabele służące do przechowywania danych protokołu i ich relacje

Jak widać, mimo prostej logiki serwera, ilość zapisywanych danych jest bardzo duża. Wiąże się to z czasochłonną implementacją prawidłowej obsługi zapisu i odczytu, dlatego przed zdecydowaniem się na korzystanie z protokołu Labyrinth należy wziąć ten czynnik pod uwagę.

4.2 Testy jednostkowe

Weryfikacja poprawności podstawowych funkcji kryptograficznych odbyła się przy pomocy testów jednostkowych. Biblioteka wielokrotnie korzysta z tych funkcji, dlatego szczególnie ważna jest ich prawidłowa implementacja. Szczególnie istotny był test weryfikujący poprawność implementacji funkcji `labyrinth_hpke_encrypt` i `labyrinth_hpke_decrypt`, które nie były zapewnione przez żadną z wykorzystywanych bibliotek. Na listingu 19 przedstawiono test wraz z komentarzami opisującymi przebieg testu. Należy zauważyć, że w trakcie szyfrowania i odszyfrowania wykorzystywane są jedynie publiczne klucze drugiej strony komunikacji, zgodnie z ideą szyfrowania asymetrycznego.

```
// psk jest współdzielone pomiędzy odbiorcą a nadawcą przed szyfrowaniem
const psk = random(KEY_LENGTH_BYTES);

// para kluczy szyfrujących odbiorcy
const { publicKey: recipient_enc_pub, privateKey: recipient_enc_priv } =
    generate_key_pair();
// para kluczy uwierzytelniających nadawcy
const { publicKey: sender_auth_pub, privateKey: sender_auth_priv } =
    generate_key_pair();
const aad = random(8);

// kod realizujący szyfrowanie po strony nadawcy
const plaintext = asciiStringToBytes('Hello Alice!');
const ciphertext = await labyrinth_hpke_encrypt(
    recipient_enc_pub,
    sender_auth_pub,
    sender_auth_priv,
    psk,
    aad,
    plaintext,
);

// kod realizujący odszyfrowanie po strony odbiorcy
const decrypted_plaintext = await labyrinth_hpke_decrypt(
    recipient_enc_pub,
    recipient_enc_priv,
    sender_auth_pub,
    psk,
    aad,
    ciphertext,
);

expect(decrypted_plaintext).toEqual(plaintext);
```

Listing 19. Test funkcji `labyrinth_hpke_encrypt` i `labyrinth_hpke_decrypt`

4.3 Testy End-To-End

Testy E2E pozwalają zweryfikować działanie aplikacji z perspektywy użytkownika końcowego, symulując rzeczywiste scenariusze użytkowania. Sprawdzają one, czy wszystkie elementy systemu, od interfejsu po komunikację z serwerem, współpracują prawidłowo.

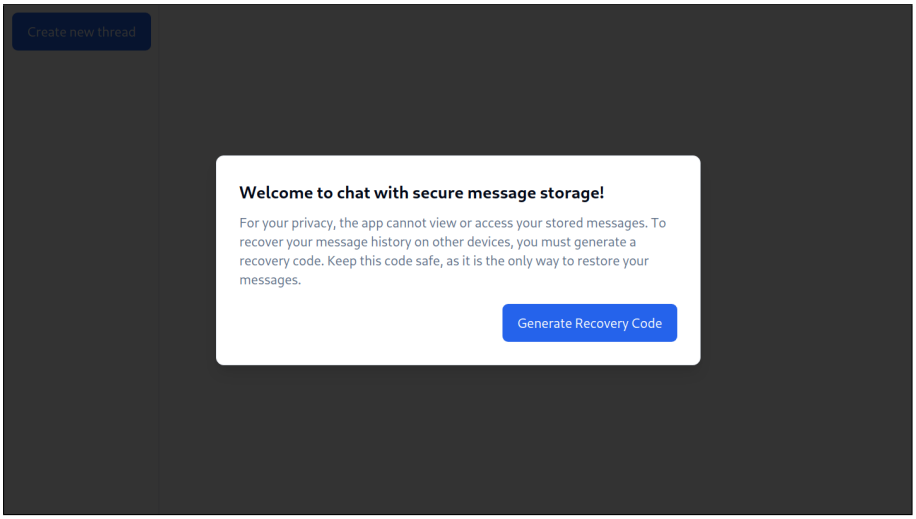
Do pisania testów wykorzystałem bibliotekę Cypress, która umożliwia testowanie aplikacji bezpośrednio w przeglądarce poprzez symulację działań użytkownika, takich jak klikanie w przyciski czy uzupełnianie pól tekstowych. Jedną z jej istotnych zalet jest wizualizacja podejmowanych akcji, co pozwala na łatwe wychwytywanie błędów podczas uruchamiania testów.

Cypress automatyzuje czasochłonne czynności, takie jak logowanie czy resetowanie danych między testami, znacząco usprawniając proces weryfikacji. Podczas tworzenia aplikacji automatyczne testy były znacznie wygodniejsze niż ręczne sprawdzanie po każdej zmianie. W późniejszych etapach projektu testy te pomagały unikać błędów wynikających z niezamierzonych zmian w innych częściach kodu.

4.3.1 Pierwsze logowanie i odzyskiwanie dostępu do pierwszej epoki

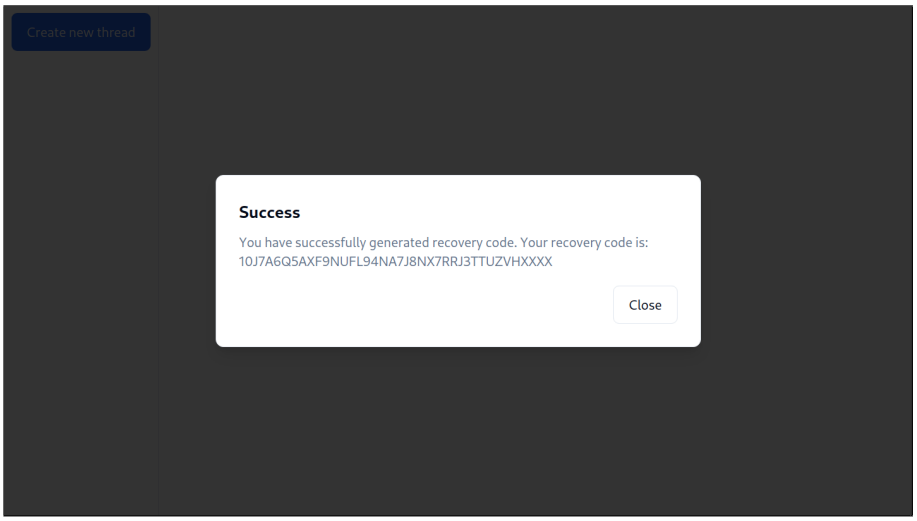
Celem tego testu było sprawdzenie poprawności generowania kodu odzyskiwania, który ma posłużyć do przywrócenia dostępu na innym urządzeniu. Poniżej opisano poszczególne kroki tego etapu.

Na początku użytkownik loguje się do systemu i przechodzi na stronę wiadomości. Aplikacja wyświetla okno powitalne, umożliwiające wygenerowanie kodu odzyskiwania. Okno przedstawione jest na rysunku 11.



Rysunek 11. Okno powitalne aplikacji

Po kliknięciu przycisku generowania kodu aplikacja rozpoczyna proces inicjalizacji. Po jego zakończeniu użytkownikowi wyświetlany jest komunikat z kodem odzyskiwania przedstawionym na rysunku 12. Można zauważyć, że ostatnie 4 znaki wygenerowanego kodu nie są losowe. Protokół Labyrinth przewiduje w tym miejscu kod korekcji błędów, który został pominięty w implementacji.



Rysunek 12. Okno z wygenerowanym kodem odzyskiwania

Kod ten jest następnie zapisywany w celu późniejszego wykorzystania podczas przywracania pierwszej epoki. Dodatkowo zapisywane są dane epoki, które zostaną wykorzystane do porównania z danymi odzyskanymi po zakończeniu procesu odzyskiwania.

Powyższe kroki realizuje kod przedstawiony na listingu 20.

```
const chosenUser: UserPool = 'user_not_in_labyrinth';
cy.login(chosenUser);
cy.visit('/messages');
getDialogExpectTitleAndButtonName(
    'Welcome to chat with secure message storage!',
    'Generate Recovery Code',
)
    .find('button')
    .click();

getDialogExpectTitleAndButtonName('Success', 'Close').as(
    'success-dialog',
);
cy.get('@success-dialog')
    .find('p')
    .invoke('text')
    .then(extractRecoveryCode)
    .as('recovery-code');

getLabyrinthEpochStateFromLocalStorage().as('labyrinth-before');
```

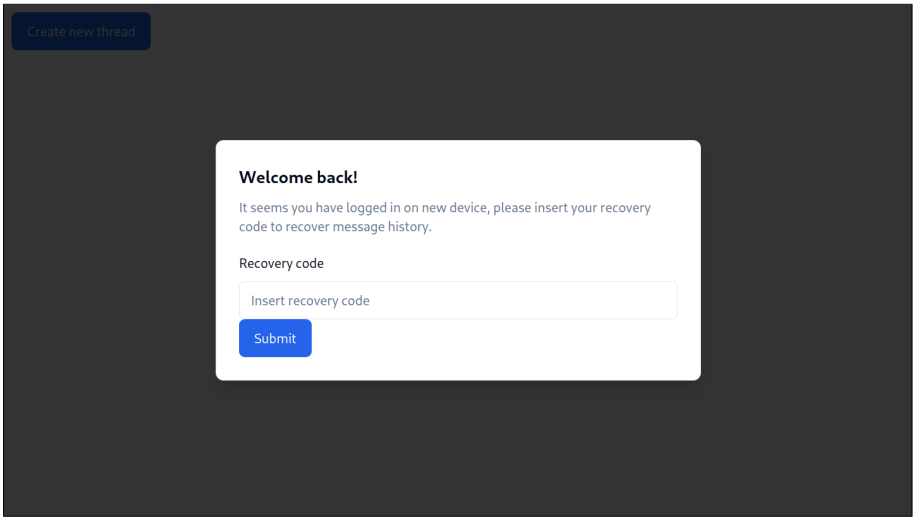
Listing 20. Fragment testu realizujący pierwsze logowanie do aplikacji

W kolejnym etapie symulowane jest logowanie użytkownika na nowym urządzeniu. W tym celu wyczyszczono dane przeglądarki, co pozwala odtworzyć sytuację, gdy użytkownik loguje się do aplikacji na innym urządzeniu.

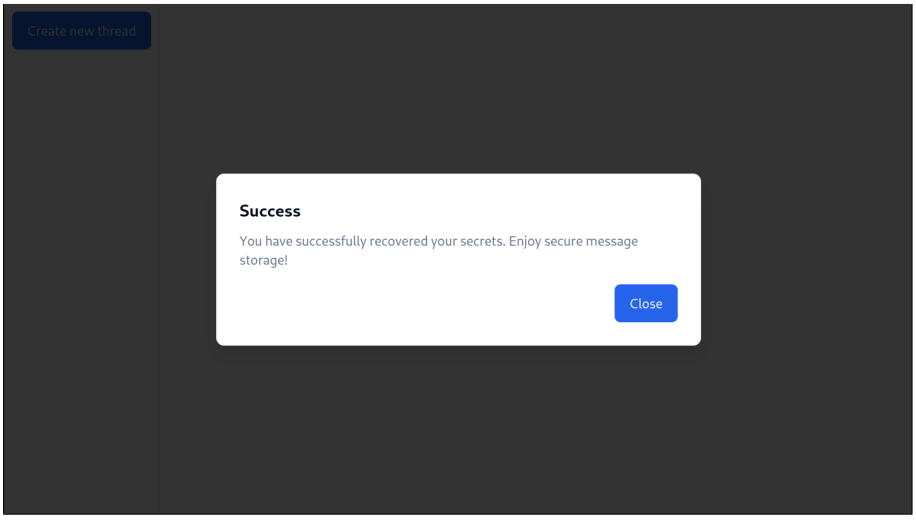
Po zalogowaniu do aplikacji wyświetlane jest okno z prośbą o podanie kodu odzyskiwania. Okno zostało przedstawione na rysunku 13.

Po wpisaniu kodu i zatwierdzeniu go, aplikacja rozpoczyna proces odzyskiwania pierwszej epoki. Na koniec wyświetlane jest okno informujące nas o sukcesie operacji. Okno przedstawia rysunek 14.

Na zakończenie testu weryfikowane jest, czy dane epok pierwotnie stworzone na urządzeniu pierwszym są takie same jak odzyskane. Kod realizujący ten etap przedstawia listing 21.



Rysunek 13. Okno z miejscem na kod odzyskiwania



Rysunek 14. Okno informujące o sukcesie operacji odzyskiwania pierwszej epoki

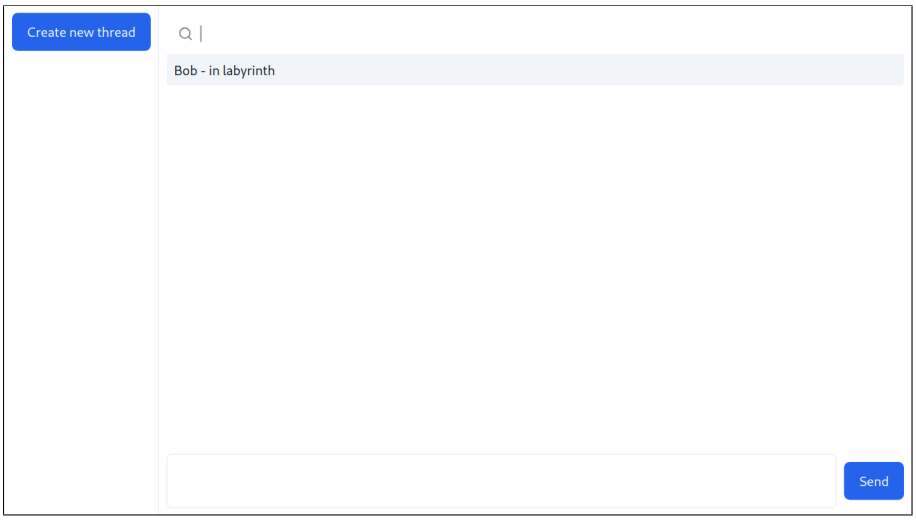
```
cy.changeToNewDevice();
cy.login(chosenUser);
cy.visit('/messages');

getDialogExpectTitleAndButtonName('Welcome back!', 'Submit').as(
    'welcome-back-dialog',
);
cy.get<string>('@recovery-code').then((recoveryCode) => {
    return cy
        .get('@welcome-back-dialog')
        .find('input')
        .type(recoveryCode);
});
cy.get('@welcome-back-dialog').find('button').click();

getLabyrinthEpochStateFromLocalStorage().as('labyrinth-after');

cy.get<string>('@labyrinth-before').then((labyrinthBefore) => {
    cy.get<string>('@labyrinth-after').then((labyrinthAfter) => {
        expect(labyrinthBefore).to.equal(labyrinthAfter);
    });
});
```

Listing 21. Fragment testu realizujący odzyskiwanie dostępu do pierwszej epoki

4.3.2 Wysyłanie i odbieranie wiadomości

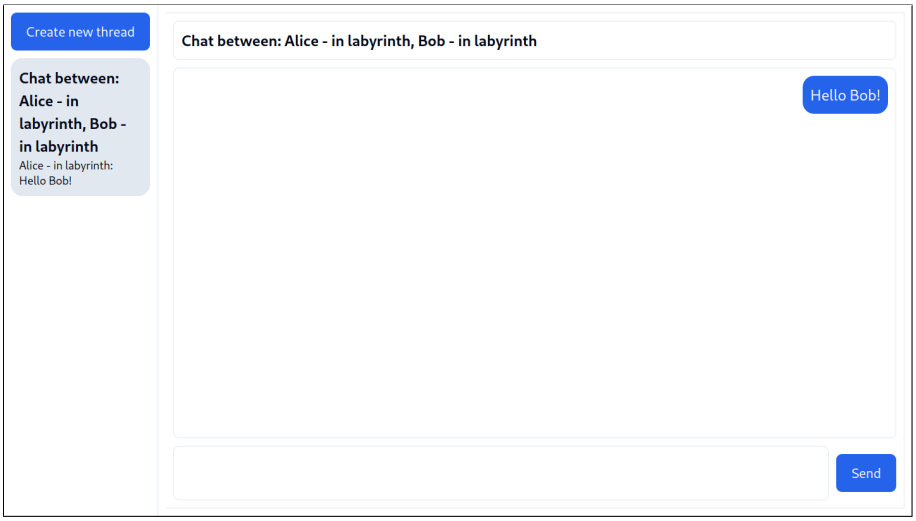
Celem testu jest utworzenie konwersacji z innym użytkownikiem, wymiana wiadomości, a następnie weryfikacja, czy historia wiadomości zostanie załadowana po odświeżeniu strony. Istotą testu jest sprawdzenie, czy wiadomości, które zostały zaszyfrowane i wysłane na serwer, są później poprawnie odszyfrowywane po ich pobraniu.

Po udanej fazie inicjalizacji użytkownik ma możliwość wysyłania wiadomości do innych użytkowników. Na początku należy stworzyć konwersację z wybranymi osobami. Strona aplikacji, na której użytkownik rozpoczyna ten proces, przedstawiona jest na rysunku 15. Użytkownik klika przycisk „Create new thread", a następnie wybiera osoby z listy znajomych, które mają zostać dodane do konwersacji.



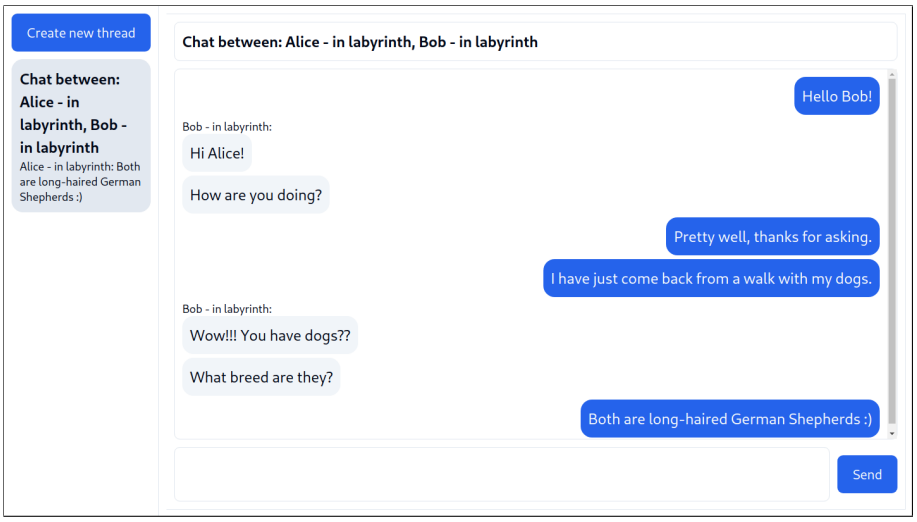
Rysunek 15. Okno wyboru uczestników konwersacji

Po dokonaniu wyboru użytkownik musi napisać pierwszą wiadomość, aby utworzyć konwersację. Kliknięcie przycisku do wysłania wiadomości tworzy nową konwersację, a pierwsza wiadomość zostaje przekazana do pozostałych użytkowników. Widok po stworzeniu nowej konwersacji przedstawiony jest na rysunku 16.



Rysunek 16. Widok po stworzeniu nowej konwersacji

Następnie użytkownicy kontynuują wysyłanie wiadomości, a test sprawdza ich prawidłowe wyświetlanie. Po odświeżeniu strony, wiadomości są pobierane z serwera w postaci zaszyfrowanej, a następnie odszyfrowywane i wyświetlane. Widok po odświeżeniu strony przedstawia rysunek 17.



Rysunek 17. Widok po wymianie wiadomości i odświeżeniu strony

Test kończy się sukcesem, jeśli historia wiadomości zostanie zachowana i prawidłowo wyświetlona po odświeżeniu strony. Całość testu przedstawia listing 22.

```
cy.loadLabyrinthForUser(alice);
cy.login(alice);

cy.visit('/messages');

const userMessagePairs: [UserPool, string][] = [
    [alice, 'Hello Bob!'],
    [bob, 'Hi Alice!'],
    [bob, 'How are you doing?'],
    [alice, 'Pretty well, thanks for asking.'],
    [alice, 'I have just come back from a walk with my dogs.'],
    [bob, 'Wow!!! You have dogs??'],
    [bob, 'What breed are they?'],
    [alice, 'Both are long-haired German Shepherds :)'],
];
createThread([bob], userMessagePairs);
checkChatThreadContent([alice, bob], [userMessagePairs[0]]);

performSendingMessages(alice, [bob], userMessagePairs.slice(1));
checkChatThreadContent([alice, bob], userMessagePairs);

cy.reload();
checkChatThreadContent([alice, bob], userMessagePairs);
```

Listing 22. Test weryfikujący wysyłanie wiadomości oraz ich zapis w postaci zaszyfrowanej

4.3.3 Tworzenie nowych epok i odzyskiwanie ich na nowym urządzeniu

Celem testu jest wysłanie do serwera zaszyfrowanej wiadomości podczas pierwszej epoki, a następnie odczekanie czasu maksymalnej nieaktywności urządzenia, co powoduje utworzenie nowej epoki. Po tym użytkownik wysyła kolejną wiadomość, tym razem podczas drugiej epoki. Na koniec testu weryfikowane jest, czy po wpisaniu kodu odzyskiwania urządzenie jest w stanie odszyfrować obie wiadomości – z pierwszej i drugiej epoki. Najważniejszy fragment testu przedstawia listing 23. W celu weryfikacji, w ramach której epoki wysyłane są wiadomości, wykorzystano funkcję `cy.intercept` z biblioteki Cypress. Funkcja ta pozwala na przechwycenie żądania odpowiedzialnego za zapis wiadomości na serwerze. Z ciała żądania można wydobyć wartość **epochId**, która reprezentuje epokę,

```
cy.intercept({
    method: 'POST',
    url: `/api/chat-service/threads/*/messages`,
}).as('message-sent-to-storage');

sendMessageAs(bob, 'Message sent in first epoch');

cy.wait(`@message-sent-to-storage`).then((interception) => {
    const usedEpochId = interception.request.body.epochId;
    getEpochIdBySequenceIdFromLocalStorage('0').then((firstEpochId) => {
        expect(usedEpochId).to.equal(firstEpochId);
    });
});

changeToNewDeviceAndInsertRecoveryCode();
waitMaxInactivityTime();

sendMessageAs(carol, 'Message sent in second epoch');

cy.wait(`@message-sent-to-storage`).then((interception) => {
    const usedEpochId = interception.request.body.epochId;
    getEpochIdBySequenceIdFromLocalStorage('1').then(
        (secondEpochId) => {
            expect(usedEpochId).to.equal(secondEpochId);
        },
    );
});

changeToNewDeviceAndInsertRecoveryCode();
cy.get('div[data-cy="thread-previews-container"]')
    .should('contain', 'Message sent in first epoch')
    .should('contain', 'Message sent in second epoch');
```

Listing 23. Test realizujący szyfrowanie w różnych epokach

podczas której zaszyfrowana została wiadomość. W liniach 11 i 24 porównywane są identyfikatory epok, co umożliwia potwierdzenie, że pierwsza wiadomość została zaszyfrowana w pierwszej epoce, a druga – w drugiej epoce. Test kończy się sukcesem, jeżeli na nowym urządzeniu korzystając z kodu odzyskiwania, zostaną poprawnie odczytane wiadomości z obu epok.

Rozdział 5

Podsumowanie

W ramach pracy stworzono komunikator tekstowy, który realizuje wszystkie założone przy projektowaniu funkcjonalności. Kluczowym elementem projektu było opracowanie biblioteki implementującej najważniejsze założenia protokołu Labyrinth, które pozwalały na przechowywanie wiadomości po stronie serwera, a także ich bezpieczny odczyt i odzyskiwanie sekretów szyfrujących na nowych urządzeniach. Jednym z najistotniejszych, a zarazem najbardziej złożonych elementów protokołu, okazał się mechanizm rotacji sekretów, który pozwala na unieważnianie nieaktywnych urządzeń.

Kod aplikacji, jak i biblioteki, został udostępniony jako otwartoźródłowy w serwisie GitHub pod adresem: https://github.com/sebastianp265/e2ee-chat-with-labyrinth.

5.1 Bezpieczeństwo protokołu

Bezpieczeństwo stanowiło kluczowy aspekt projektu. Analiza dokumentacji protokołu Labyrinth nie ujawniła luk, które mogłyby być wykorzystane przez osoby niepowołane.

Największym zagrożeniem pozostaje możliwość przechwycenia kodu odzyskiwania w połączeniu z dostępem do bazy danych. W takiej sytuacji atakujący mógłby odtworzyć wszystkie epoki i uzyskać pełny dostęp do wiadomości. Z tego powodu w opisie protokołu wielokrotnie podkreślano konieczność unikania przechowywania kodu odzyskiwania oraz danych wirtualnego urządzenia.

Warto jednak pamiętać, że żadne rozwiązanie kryptograficzne nie jest całkowicie odporne na zagrożenia wynikające z fizycznego dostępu do urządzenia użytkownika. Atakujący, który uzyska bezpośredni dostęp do urządzenia, może ominąć zabezpieczenia i odczytać wiadomości w postaci jawnej, tak jak są one widoczne dla użytkownika.

5.2 Dalsze kierunki rozwoju

Projekt, mimo spełnienia głównych założeń, pozostawia przestrzeń do dalszego rozwoju, zarówno w zakresie rozszerzenia funkcjonalności, jak i zwiększenia poziomu bezpieczeństwa.

5.2.1 Przesyłanie załączników

Interesującym kierunkiem rozwoju mogłaby być implementacja przesyłania plików. Protokół Labyrinth, omawiając sposób przechowywania załączników, wprowadza funkcję nazywaną Oblivious Revocable Function (ORF). Ma ona na celu ograniczenie powiązania między załącznikami przechowywanymi na serwerze a konkretnymi użytkownikami. Temat ten jest szczególnie ciekawy ze względu na zastosowane rozwiązania kryptograficzne, a także z uwagi na publikację pracowników Facebooka, którzy opisali szczegóły matematyczne tego mechanizmu [7].

5.2.2 Odzyskiwanie sekretów starszych epok

Protokół Labyrinth opisuje również mechanizm odzyskiwania sekretów starszych epok, jednak jego implementacja została pominięta z uwagi na brak wystarczających szczegółów w dokumentacji oraz zbyt szeroki zakres pracy.

Moja analiza wskazała, że możliwość odtwarzania starszych epok mogłaby umożliwić przenoszenie wirtualnego urządzenia do nowszych epok, co poprawiłoby wydajność, szczególnie w przypadku dużej liczby epok. Użytkownik mógłby wówczas dołączyć tylko do najnowszej epoki, aby pobrać najnowsze wiadomości. W razie potrzeby możliwe byłoby dołączyć także do starszych epok, aby uzyskać dostęp do wcześniejszych wiadomości. Takie rozwiązanie mogłoby znacząco przyspieszyć działanie aplikacji, zwłaszcza przy pierwszym uruchomieniu na nowym urządzeniu, gdzie dostęp do najnowszych wiadomości jest kluczowy.

5.2.3 Usprawnienie procesu odzyskiwania sekretów

Odzyskiwanie sekretów za pomocą kodu odzyskiwania zapewnia wysoki poziom bezpieczeństwa, ale może być uciążliwe dla użytkowników, zwłaszcza w sytuacjach, gdy dostęp do kodu jest utrudniony. Można wprowadzić bardziej przyjazny mechanizm, taki jak odzyskiwanie sekretów na nowym urządzeniu z wykorzystaniem fizycznego dostępu do urządzenia, które zostało wcześniej zainicjalizowane w protokole. Protokół Labyrinth sugeruje użycie jednorazowego kodu wyświetlanego na zainicjalizowanym urządzeniu, który użytkownik może przepisać na nowym urządzeniu w celu odzyskania sekretów. Niestety, Meta nie precyzuje, jak miałaby działać taka funkcja, pozostawiając wiele aspektów do dalszej analizy tego typu rozwiązań kryptograficznych.

5.2.4 Dodanie szyfrowania E2E

Obecna implementacja ogranicza się do realizacji protokołu Labyrinth, który koncentruje się na bezpiecznym przechowywaniu wiadomości po stronie serwera. Aby zapewnić pełną poufność komunikacji, konieczne byłoby wdrożenie szyfrowania typu E2E. W tym celu można wykorzystać sprawdzone rozwiązania, takie jak protokół Signal, który od lat jest uznawany za wzór w dziedzinie bezpiecznej komunikacji.

Bibliografia

[1] Dance, G. J., LaForgia, M. i Confessore, N., „As Facebook Raised a Privacy Wall, It Carved an Opening for Tech Giants", *The New York Times*, grud. 2018, https://www.nytimes.com/2018/12/18/technology/facebook-privacy.html, dostęp uzyskano 2024-12-21.

[2] Jon Millican, R. R., „Building end-to-end security for Messenger", grud. 2023, https://engineering.fb.com/2023/12/06/security/building-end-to-end-security-for-messenger/, dostęp uzyskano 2024-12-23.

[3] Josefsson, S. i Liusvaara, I., *Edwards-Curve Digital Signature Algorithm (EdDSA)*, RFC 8032, sty. 2017. DOI: 10.17487/RFC8032. adr.: https://www.rfc-editor.org/info/rfc8032.

[4] Krawczyk, H. i Eronen, P., *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*, RFC 5869, maj 2010. DOI: 10.17487/RFC5869. adr.: https://www.rfc-editor.org/info/rfc5869.

[5] Kulkarni, N. D. i Bansal, S., „Utilizing the Facade Design Pattern in Practical Phone Application Scenario", *Journal of Artificial Intelligence & Cloud Computing*, t. 1, nr. 1, s. 15–22, 2022. adr.: https://www.onlinescientificresearch.com/articles/utilizing-the-facade-design-pattern-in-practical-phone-application-scenario.pdf.

[6] Langley, A., Hamburg, M. i Turner, S., *Elliptic Curves for Security*, RFC 7748, sty. 2016. DOI: 10.17487/RFC7748. adr.: https://www.rfc-editor.org/info/rfc7748.

[7] Lewi, K., Millican, J., Raghunathan, A. i Roy, A., *Oblivious Revocable Functions and Encrypted Indexing*, Cryptology ePrint Archive, Paper 2022/1044, 2022. adr.: https://eprint.iacr.org/2022/1044.

[8] Łacheciński, S., „Notacje modelowania w projektowaniu relacyjnych baz danych Modelling notations in the design of relational databases", t. 34, s. 5–21, grud. 2013.

[9] Mamun, A., Rahman, S., Shaon, T. i Hossain, M. A., „Security Analysis of AES and Enhancing its Security by Modifying S-Box with an Additional Byte", *International journal of Computer Networks & Communications*, t. 9, s. 69–88, mar. 2017. DOI: 10.5121/ijcnc.2017.9206.

[10] Meta, *The Labyrinth Encrypted Message Storage Protocol*, https://engineering.fb.com/wp-content/uploads/2023/12/TheLabyrinthEncryptedMessageStorageProtocol_12-6-2023.pdf, dostęp uzyskano 2024-12-23, grud. 2023.

[11] Meyer, M., Reith, S. i Campos, F., *On hybrid SIDH schemes using Edwards and Montgomery curve arithmetic*, Cryptology ePrint Archive, Paper 2017/1213, 2017. adr.: https://eprint.iacr.org/2017/1213.

[12] Urząd Komunikacji Elektronicznej, „Raport o stanie rynku telekomunikacyjnego w 2023 roku", Urząd Komunikacji Elektronicznej, spraw. tech., 2024.

Wykaz skrótów i symboli

AAD Additional Authenticated Data 13, 14

AES Advanced Encryption Standard 13

E2E End-To-End 10, 41, 50

ERD Entity Relationship Diagram 39

GCM Galois/Counter Mode 13

HKDF HMAC-based key derivation function 14

HTTP Hypertext Transfer Protocol 38

IE Information Engineering 39

KDF Key Derivation Function 13

ORF Oblivious Revocable Function 50

P2P Point-To-Point 9

PSK Pre-Shared Key 14

SMS Short Message Service 9