Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Automatyki i Informatyki Stosowanej

Praca dyplomowa magisterska

na kierunku Informatyka
w specjalności Systemy Internetowe Wspomagania Zarządzania

Bezpieczeństwo poczty elektronicznej

Adam Mizerski
Numer albumu 225614

promotor
dr Piort Sapiecha

Zielonka 2020

Bezpieczeństwo poczty elektronicznej

Streszczenie

Poczta elektroniczna jest podstawowym narzędziem komunikacji w Internecie. Pomimo kluczowej roli tego systemu we współczesnym świecie, kwestie jego bezpieczeństwa zostały niestety mocno zaniedbane.

W tej pracy przedstawione są najpierw zagrożenia i mające im przeciwdziałać mechanizmy bezpieczeństwa obecnie wykorzystywane w poczcie elektronicznej.

Następnie opisane są i porównane alternatywne aplikacje i protokoły: *DarkMail*, *BitMessage*, *RetroShare*, *Freenet*, *I2P-Bote* i *LemonMail*.

Trzecim elementem pracy jest zaprojektowanie i implementacja systemu pocztowego nazwanego *TigerMail*. Działa on w sieci rozproszonej, wykorzystując *blockchain Ethereum* i rozproszony system plików *Swarm* oraz zapewnia poufność wymiany informacji z zachowaniem *forward secrecy*, wykorzystując protokół *Signal*.

Słowa kluczowe: poczta elektroniczna, *email*, *SMTP*, *STARTTLS*, *SPF*, *DKIM*, *DMARC*, *S/MIME*, *OpenPGP*, *PGP*, *DarkMail*, *BitMessage*, *RetroShare*, *Freenet*, *I2P*, *I2P-Bote*, *LemonMail*, *blockchain*, *Ethereum*, *Swarm*, *forward secrecy*, *Signal*.

Security of electronic mail

Summary

Electronic mail is a basic tool for communication over the Internet. Despite its key role in modern world, its security was left neglected.

This work begins with description of threats and countermeasures currently used in electronic mail.

Next there are presented and compared alternative applications and protocols: *DarkMail*, *BitMessage*, *RetroShare*, *Freenet*, *I2P-Bote* and *LemonMail*.

The third part of this work is design and implementation of mail system named *TigerMail*. It's working in a distributed network, using *Ethereum blockchain* and distributed file system *Swarm*. It provides *forward secrecy* using *Signal* protocol.

Keywords: *email*, *SMTP*, *STARTTLS*, *SPF*, *DKIM*, *DMARC*, *S/MIME*, *OpenPGP*, *PGP*, *DarkMail*, *BitMessage*, *RetroShare*, *Freenet*, *I2P*, *I2P-Bote*, *LemonMail*, *blockchain*, *Ethereum*, *Swarm*, *forward secrecy*, *Signal*.

załącznik nr 3 do zarządzenia
nr 28 /2016 Rektora PW

Politechnika Warszawska

…….....
miejscowość i data

……..
imię i nazwisko studenta

……..
numer albumu

…….…….
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
czytelny podpis studenta

Spis treści

Rozdział 1

Wstęp

Motywacja

Poczta elektroniczna jest podstawowym narzędziem komunikacji w Internecie. W 2017 roku, 6,3 mld kont należących do 3,7 mld użytkowników, wysyłało ponad 269 mld wiadomości dziennie [14].

Pomimo kluczowej roli tego systemu we współczesnym świecie, kwestie jego bezpieczeństwa zostały niestety mocno zaniedbane.

Z tego powodu najpierw zbadałem, opisałem i porównałem mechanizmy bezpieczeństwa wykorzystywane w poczcie elektronicznej z istniejącymi alternatywami. W trakcie pisania tej pracy wymyśliłem i stworzyłem nowy system poczty, oferujący lepsze bezpieczeństwo komunikacji, niż te, które znalazłem i opisałem.

Bezpieczeństwo w kontekście komunikacji przez Internet

Na wstępie należy zadać sobie pytanie: „Czy komunikacja przez sieć jest bezpieczna?". Na to dość ogólne pytanie niełatwo jest odpowiedzieć. Pomocne będzie rozbicie go na kilka bardziej szczegółowych:

- Czy ktoś niepowołany może odczytać treść wiadomości?
  - Czy w tym celu wystarczy pasywnie podsłuchiwać i jeżeli tak, to gdzie?
  - Czy jest możliwe przeprowadzenie ataku na jakiś element infrastruktury (na przykład na serwer pośredniczący), tak by ofiara nie była tego świadoma? Jak daleko idące są skutki udanego ataku?
- Jak wiele można odczytać z samych publicznie widocznych metadanych wiadomości (nadawca, odbiorca, data wysłania, rozmiar itp.), nie znając jej treści?
- Jakie dane są niezbędne do dostępu do konta/tożsamości (hasło, hasła jednorazowe, urządzenie *universal second factor* (*U2F*), klucz prywatny itp.)? Jak atakujący mógłby je odkryć lub zdobyć? Jak daleko idące są skutki wycieku danych dostępowych?
- Czy ktoś może wysłać wiadomość, podszywając się pod kogoś innego, bez kradzieży danych dostępowych? Jakie są możliwe drogi ataku?

Bezpieczeństwo obecnej poczty elektronicznej

Określenie poziomu bezpieczeństwa poczty elektronicznej jest dość problematyczne i odpowiedź na większość powyższych pytań zaczyna się od „to zależy".

Protokoły komunikacji opracowane dla poczty elektronicznej powstały w 1973 roku [7], w sieci *ARPANET* (prekursor Internetu). Wtedy wszyscy użytkownicy sieci byli z założenia zaufani, ponieważ należeli do jednej organizacji. Z tego powodu protokoły nie zawierały żadnych mechanizmów zabezpieczeń. Wszystko było przesyłane jawnym tekstem, a adres nadawcy można było dowolnie edytować.

Od tego czasu potrzeba zapewnienia bezpieczeństwa komunikacji wzrosła od nieistotnej do bardzo poważnej. Protokoły pocztowe były rozszerzane o różne metody zabezpieczeń, takie jak szyfrowanie połączeń *Transport Layer Security* (*TLS*) lub zabezpieczanie przed *email spoofing*, jednak do tej pory powszechność stosowania tych rozwiązań pozostawia wiele do życzenia [18, 21]. Jest to spowodowane tym, że nie ma mechanizmu wymuszającego stosowanie odpowiednich zabezpieczeń. Dla użytkowników najważniejsze jest, żeby wiadomość dotarła do odbiorcy, więc oprogramowanie serwerowe priorytetyzuje dostarczenie wiadomości kosztem bezpieczeństwa transmisji. Jeżeli serwer odbiorcy nie obsługuje pożądanych mechanizmów bezpieczeństwa, to serwer nadawcy wyśle wiadomość bez ich stosowania.

Inną metodę zabezpieczenia poczty oferują protokoły szyfrowania *end-to-end*, takie jak *Secure/Multipurpose Internet Mail Extensions* (*S/MIME*) lub *OpenPGP*. Działają one, ingerując

bezpośrednio w treść wiadomości, dzięki czemu zabezpieczanie komunikacji przez pośredników nie jest już tak istotne. Te również nie cieszą się niestety popularnością, głównie z powodu zbyt wysokiego stopnia skomplikowania dla większości użytkowników [1, 31, 88]. Znanym przypadkiem, pokazującym jak trudne jest rozpoczęcie szyfrowanej komunikacji pocztowej, stała się próba kontaktu Edwarda Snowdena z dziennikarzem Glennem Greenwaldem [26].

Struktura sieci poczty elektronicznej

Poczta elektroniczna jest systemem zdecentralizowanym. Jest to wygodna architektura dla komunikacji asynchronicznej, ponieważ serwery są obecne w sieci przez cały czas i przechowują pocztę dla użytkowników. Użytkownicy muszą jednak obdarzyć serwery dużym zaufaniem, jeśli chodzi o bezpieczeństwo przechowywanych danych i dbanie o prywatność użytkowników. Drugiej kwestii niestety niewiele osób poświęca uwagę – większość darmowych usług pocztowych w zapisach regulaminów wręcz zawiera punkty dotyczące możliwości wykorzystania przez administratorów treści korespondencji do na przykład profilowania i celów marketingowych.

Sieci rozproszone pozwalają pozbyć się problemu niedbających o prywatność użytkowników serwerów. Bardzo dobrze sprawdza się to w przypadku komunikacji synchronicznej, gdzie rozmówcy są jednocześnie obecni w sieci (na przykład w komunikatorach *Tox*[1] i *Briar*[2]). Jednak dla komunikacji asynchronicznej powstaje nowy problem: kto przechowa wiadomość, gdy odbiorcy nie ma w sieci i jaką mamy gwarancję, że to zrobi? W rozwiązaniu tego problemu pomóc może technologia *blockchain*.

Możliwości rozwoju

Mimo swoich wad i faktu, że różne firmy tworzą komunikatory mające być lepszą alternatywą, *email* istnieje do dzisiaj. Zdecentralizowana architektura i wykorzystywanie otwartych protokołów powodują, że nie istnieje firma lub organizacja, która mogłaby zakończyć działanie poczty. Dzięki czemu jest to system stabilny i powszechnie dostępny. Z drugiej strony nie ma jednostki organizacyjnej, która mogłaby zmodernizować protokół, wprowadzając niekompatybilne wstecz zmiany, skutkiem czego każdy, kto chce korzystać z poczty elektronicznej, musi akceptować jej obecną formę. Żaden użytkownik nie może wprowadzić własnych modyfikacji, ponieważ utraci możliwość komunikacji z innymi.

[1] https://tox.chat/
[2] https://briarproject.org/

Do tej pory nie powstała też żadna alternatywa, która rozwiązałaby wszystkie problemy obecnej poczty. Wynikać to może z faktu, że różni użytkownicy poczty mają różne, nierzadko wykluczające się, wymagania i modele zagrożeń. Na przykład dziennikarze śledczy i aktywiści będą wymagać szyfrowania *end-to-end* i ochrony prywatności, w tym metadanych (nadawca, odbiorca, rozmiar itp.). W bankach i korporacjach natomiast ważna może być możliwość przeprowadzenia audytu. Autorzy *„Securing Email"* [14] sugerują, że być może trzeba porzucić ideę jednego rozwiązania, które obejmie wszystkie przypadki.

Krótki przegląd protokołów

W tej pracy przeanalizowałem istniejące protokoły. Poniżej znajduje się krótkie omówienie każdego z nich:

Email (rozdział 2) – Powszechnie używana poczta elektroniczna, wykorzystująca protokoły *Simple Mail Transfer Protocol* (*SMTP*), *Post Office Protocol* (*POP3*) i *Internet Message Access Protocol* (*IMAP*).

***Dark Internet Mail Environment* (*DIME*)** (rozdział 3.1) – System poczty stworzony na podstawie protokołów *SMTP* i *IMAP*. Wprowadza mechanizmy dystrybucji i weryfikacji kluczy oraz wymaga szyfrowania danych. Wiadomość jest podzielona na części, z których każda może być zaszyfrowana innym kluczem, na przykład nagłówki dla serwera a treść dla odbiorcy.

BitMessage (rozdział 3.2) – Sieć rozproszona, w której zaszyfrowana wiadomość jest wysyłana do wszystkich użytkowników. Tylko adresat może ją odszyfrować. Inni użytkownicy dobrowolnie przechowują niedostarczone wiadomości.

RetroShare (rozdział 3.3) – Sieć rozproszona typu *friend-to-friend*, wykorzystująca protokoły *OpenPGP* i *TLS*. W przypadku nieobecności odbiorcy, wiadomości są przechowywane przez zaufanych znajomych.

Freemail (rozdział 3.4) – System pocztowy zbudowany na rozproszonym systemie plików *Freenet*. Wymiana wiadomości polega na zapisywaniu danych w sieci i metadanych pod określonymi adresami.

I2P-Bote (rozdział 3.5) – System pocztowy działający wewnątrz sieci anonimizującej *I2P*. Wysłane wiadomości są zapisywane w *Kademlia DHT* i usuwane po odebraniu.

LemonMail (rozdział 3.6) – System pocztowy wykorzystujący rozproszony system plików *Inter-Planetary File System* (*IPFS*) do zapisywania danych i *blockchain Ethereum* do metadanych.

W ramach tej pracy stworzyłem nowy system pocztowy:

TigerMail (rozdział 4) – Mój system pocztowy, wykorzystujący rozproszony system plików *Swarm* do zapisywania danych i *blockchain Ethereum* do metadanych. Oferuje *forward secrecy* dzięki wykorzystaniu protokołu *Signal*.

Tabela 1.1 zawiera podsumowanie moich badań przeprowadzonych w ramach tej pracy. Więcej szczegółów znajduje się w rozdziale 3.7.

	wartości pożądane	*Email*	*DIME*	*BitMessage*	*RetroShare*	*Freemail*	*I2P-Bote*	*LemonMail*	*TigerMail*
sieć rozproszona		○	○	●	●	●	●	●	●
email spoofing	○	◐	○	○	○	○	○	○	○
komunikacja asynchroniczna	●	●	●	●	●	●	●	◐	●
gwarancja przechowania danych	●	●	●	○	◐	○	○	◐	◐
forward secrecy	●	○	○	○	○	◐	○	○	●
inny użytkownik widzi, że wiadomość została wysłana i przez kogo	○	○	○	●	◐	○	○	●	●
inny użytkownik widzi, do kogo została wysłana wiadomość	○	○	○	○	◐	◐	○	●	○

Tabela 1.1: Porównanie systemów pocztowych

●– cecha obecna, ◐– częściowo, ○– brak.

1.1 Cel pracy

Celem pracy jest stworzenie bezpiecznego systemu asynchronicznej wymiany wiadomości, nazwanego przeze mnie *TigerMail*. Ma on wykorzystywać sieć rozproszoną i zapewniać *forward secrecy*.

W tym celu:

- Przedstawię obecny stan rzeczy, protokoły wykorzystywane do przesyłania poczty elektronicznej, metody ich zabezpieczania i skalę stosowania tych metod (w tym następujących protokołów i standardów: *SASL*, *SMTP*, *POP3*, *IMAP*, *TLS*, *STARTTLS*, *SMTP DANE*, *SMTP STS*, *SPF*, *DKIM*, *DMARC*, *ARC*, *S/MIME*, *OpenPGP*, *Identity-Based Encryption*).
- Przedstawię alternatywne systemy asynchronicznej wymiany wiadomości, ze szczególnym uwzględnieniem zastosowanych mechanizmów szyfrowania i ochrony prywatności (*DIME*, *BitMessage*, *RetroShare*, *Freenet*, *I2P-Bote*, *LemonMail*).
- Stworzę specyfikację wymagań, opracuję koncepcję i architekturę systemu *TigerMail*, w języku naturalnym i *UML*.
- Zaimplementuję aplikację w języku *C++* dla systemu *GNU/Linux*. Przeprowadzę testy opracowanego rozwiązania, w tym: funkcjonalne, bezpieczeństwa i wydajnościowe.

Własny wkład autora

Porównam istniejące rozwiązania problemów bezpieczeństwa w tradycyjnej poczcie elektronicznej oraz w alternatywnych protokołach.

Przedstawię sposób na użycie protokołu *Signal*, w komunikacji asynchronicznej w sieci rozproszonej, z wykorzystaniem technologii *blockchain Ethereum* i rozproszonego systemu plików *Swarm*. Zaimplementuję to jako aplikację o nazwie *TigerMail* i przeprowadzę testy.

1.2 Środowisko projektu

Aplikacja *TigerMail* została wykonana przy pomocy następujących narzędzi:

- System operacyjny: *GNU/Linux*, dystrybucja *openSUSE Tumbleweed* (https://www.opensuse.org/)
- Język programowania: *C++17* (https://isocpp.org/)
- Kompilator: *gcc* (https://gcc.gnu.org/) 10.1.1
- System budowania: *meson* (https://mesonbuild.com/) 0.54.2
- Biblioteki:
  - *Boost* (https://www.boost.org/) 1.71
    - *ASIO* – obsługuje komunikację sieciową, współbieżność, sygnały systemu operacyjnego i zadania uruchamiane czasowo.
    - *filesystem* – w celu generowania unikalnych nazw plików.
    - *log* – obsługuje logowanie zdarzeń w aplikacji.
    - *program options* – parsuje argumenty wiersza poleceń.
    - *unit test framework* – ułatwia implementację testów jednostkowych.
  - *CryptoPP* (https://cryptopp.com/) 8.2.0 – dostarcza funkcje kryptograficzne.
  - *gsl* (https://github.com/Microsoft/gsl) 2.0.0 – zawiera narzędzia ułatwiające programowanie.
  - *jsoncpp* (https://github.com/open-source-parsers/jsoncpp) 1.9.2 – parsuje i generuje dane w formacie *JSON*.
  - *libethereum* (https://github.com/ethereum/aleth/) 1.6.0 – tworzy transakcje dla sieci *Ethereum*.
  - *libsignal-protocol-c* (https://github.com/signalapp/libsignal-protocol-c) 2.3.3 – umożliwia komunikację protokołem *Signal*.
  - *LibreSSL* (https://www.libressl.org/) 3.1.2 – dostarcza funkcje kryptograficzne, w szczególności *Rijndael-256*, której nie ma w *CryptoPP*.
  - *protobuf-lite* (https://developers.google.com/protocol-buffers/) 3.11.4 – umożliwia serializację danych binarnych.
  - *range-v3* (https://ericniebler.github.io/range-v3/) 0.9.1 – zawiera narzędzia ułatwiające programowanie.

  - *SML* (https://boost-experimental.github.io/sml/) 1.1.0 – umożliwia tworzenie maszyn stanów.
  - *SQLite* (https://sqlite.org/) 3.31.1 – silnik bazy danych działający w aplikacji i zapisujący dane w jednym pliku.
  - *sqlite_modern_cpp* (https://github.com/SqliteModernCpp/sqlite_modern_cpp/) 3.2 – dostarcza wygodny interfejs do biblioteki *SQLite* w języku *C++14*.
  - *vmime* (https://www.vmime.org/) 0.9.2 – parsuje i generuje wiadomości w formacie *Internet Message Format (IMF)* i *MIME*.
- *ip2unix* (https://github.com/nixcloud/ip2unix/) 2.1.3 – pozwala wymusić w programie komunikację przez *Unix Domain Socket*.
- *Thunderbird* (https://www.thunderbird.net/) 68.8.1 – klient poczty.

1.3 Plan pracy

Rozdział 2

Przegląd i analiza protokołów wykorzystywanych w tradycyjnej poczcie elektronicznej.

Rozdział 3

Przegląd i analiza innych projektów dostarczających bezpieczną komunikację asynchroniczną.

Rozdział 4

Opis projektowania, implementacji i testów mojego systemu bezpiecznej poczty *TigerMail*.

Rozdział 5

Podsumowanie pracy.

Rozdział 2

Poczta Elektroniczna

Email jest najstarszą i najpopularniejszą formą komunikacji w Internecie. W tym rozdziale opiszę jego architekturę, wykorzystywane algorytmy i rozwiązania problemów bezpieczeństwa.

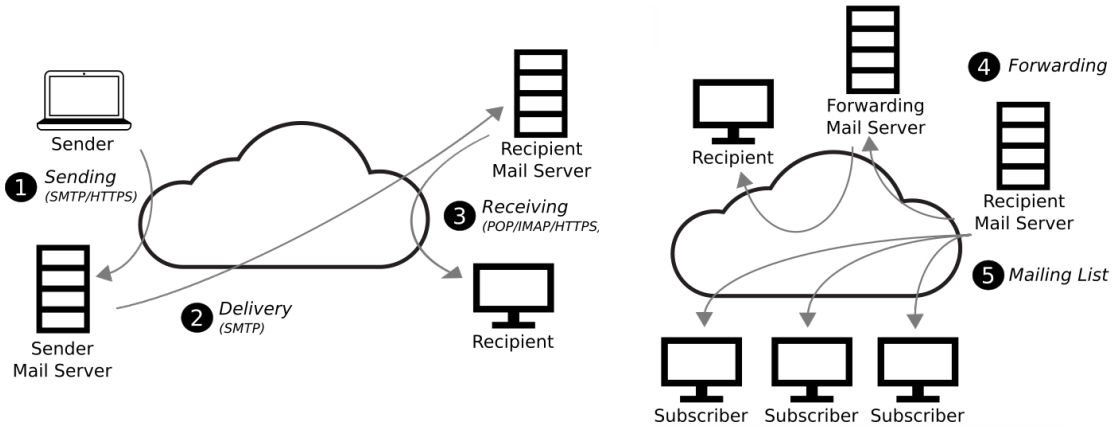
2.1 Architektura systemu pocztowego

Ogólna idea poczty elektronicznej wygląda następująco:

- Każdy użytkownik ma swoją skrzynkę pocztową na serwerze w sieci.
- Użytkownik może wysłać wiadomość na skrzynkę innego użytkownika, używając adresu `nazwa_użytkownika@adres_serwera`.
- Użytkownik wysyła i odbiera wiadomości, komunikując się z serwerem za pomocą programu klienckiego, określanego jako *Mail User Agent* (*MUA*).

Architektura poczty elektronicznej jest nieznacznie bardziej złożona:

- *MUA* wysyła wiadomości przez serwer *Mail Transfer Agent* (*MTA*, pośrednik przekazu poczty), używając protokołu *SMTP* (nr 1 i 2 na rysunku 2.1).
- *MUA* pobiera zawartość skrzynki od *MTA*, używając protokołu *POP3* lub *IMAP* (nr 3 na rysunku 2.1).



Rysunek 2.1: Przegląd protokołów i architektury poczty elektronicznej
Źródło: *[14]*

- Po drodze między nadawcą a odbiorcą może znajdować się nie jeden *MTA*, a cały łańcuch *MTA* pośredniczących (nr 4 na rysunku 2.1). W takim łańcuchu wyróżniamy:
  - *Mail Submission Agent* (*MSA*) jest pierwszym serwerem w łańcuchu *MTA* i z nim bezpośrednio łączy się *MUA* nadawcy. *MSA* odpowiada za uwierzytelnianie użytkowników, może zwracać do *MUA* komunikaty o błędach w wysyłanych wiadomościach lub samodzielnie je poprawiać.
  - *Mail Delivery Agent* (*MDA*) jest ostatnim serwerem w łańcuchu *MTA* i z nim bezpośrednio łączy się *MUA* odbiorcy. *MDA* zajmuje się przechowywaniem zawartości skrzynek odbiorczych oraz odpowiada za uwierzytelnianie użytkowników.
- Niektóre *MUA* pozwalają tylko na odczytywanie poczty z lokalnej kopii skrzynki, pozostawiając pobieranie wiadomości od *MDA* niezależnym programom określanym jako *mail retrieval agent* (*MRA*).

2.2 Podstawowe protokoły poczty elektronicznej

Internet Message Format – IMF

Dokument RFC 5322 [61] opisuje strukturę wiadomości pocztowej. Wiadomość składa się z nagłówków używanych przez programy pocztowe i treści dla odbiorcy.

Standard *Multipurpose Internet Mail Extensions (MIME)* (RFC 2045 i powiązane) rozszerza ten format i pozwala używać innego kodowania niż *US-ASCII*, tworzyć wiadomości wykorzystujące *HTML*, czy dodawać załączniki.

Listing 2.1: Przykładowa wiadomość w formacie *IMF*

```
From: John Doe <jdoe@machine.example>
To: Mary Smith <mary@example.net>
Subject: Saying Hello
Date: Fri, 21 Nov 1997 09:55:06 -0600
Message-ID: <1234@local.machine.example>

This is a message just to say hello.
So, "Hello".
```

Standard nie zawiera mechanizmów podpisywania ani szyfrowania wiadomości. Zostały one dodane dopiero jako rozszerzenia, które opisałem w rozdziale 2.5. Zaszyfrowanie lub podpisanie wiadomości na tym poziomie może ochronić treść przed ewentualnym atakującym, który mógłby mieć kontrolę nad dowolnym z serwerów pośredniczących.

Wiadomości w tym formacie są przesyłane przez protokoły *SMTP*, *POP3* i *IMAP* opisane poniżej.

Simple Authentication and Security Layer – SASL

Simple Authentication and Security Layer (*SASL*) [90] jest protokołem pozwalającym na negocjację metody i przeprowadzenie uwierzytelnienia. Jest zaprojektowany tak, by można było go wykorzystywać jako element w innych protokołach.

Lista zdefiniowanych metod jest dość długa [73], ale serwer może obsługiwać tylko wybrane. Lista obsługiwanych metod jest wysyłana przez serwer w ramach protokołu wykorzystującego *SASL*. Najczęściej dostępne metody (opis tego, jak to sprawdziłem, jest w następnych rozdziałach) to *LOGIN* [54] i *PLAIN* [89]. Polegają one na przesłaniu loginu i hasła jawnym tekstem, co jest akceptowalne, jeżeli połączenie jest szyfrowane [71, rozdział 14].

Listing 2.2: Przykładowa sesja uwierzytelniania *SASL*

```
S: * ACAP (SASL "CRAM-MD5") (STARTTLS)
C: a001 STARTTLS
S: a001 OK "Begin TLS negotiation now"
<TLS negotiation, further commands are under TLS layer>
S: * ACAP (SASL "CRAM-MD5" "PLAIN")
C: a002 AUTHENTICATE "PLAIN"
S: + ""
C: {21}
C: <NUL>tim<NUL>tanstaaftanstaaf
S: a002 OK "Authenticated"
```

Simple Mail Transfer Protocol – SMTP

Simple Mail Transfer Protocol (*SMTP*) [39] pozwala wysłać wiadomość pocztową na serwer (kroki 1, 2 i 4 na rysunku 2.1).

Listing 2.3: Przykładowa sesja *SMTP*

```
S: 220 foo.com Simple Mail Transfer Service Ready
C: EHLO bar.com
S: 250-foo.com greets bar.com
S: 250-8BITMIME
S: 250-SIZE
S: 250-DSN
S: 250 HELP
C: MAIL FROM:<Smith@bar.com>
S: 250 OK
C: RCPT TO:<Jones@foo.com>
S: 250 OK
C: DATA
S: 354 Start mail input; end with <CRLF>.<CRLF>
C: Blah blah blah...
C: ...etc. etc. etc.
C: .
S: 250 OK
C: QUIT
S: 221 foo.com Service closing transmission channel
```

Warto zwrócić uwagę na to, że informacje identyfikujące nadawcę i odbiorców znajdują się zarówno w *SMTP*, w liniach `MAIL FROM:` i `RCPT TO:`, jak i w przesyłanej wiadomości (rozdział 2.2). Serwer *MDA* odbierając wiadomość, musi dodać adres z otrzymanego `MAIL FROM:` do niej jako nagłówek `Return-Path:` w *IMF*.

Adres nadawcy i odbiorców są takie same w obu miejscach, gdy *MUA* wysyła wiadomość do *MSA* (nr 1 na rysunku 2.1). Uwierzytelnianie użytkownika przeprowadzane w tym miejscu zostało dodane jako rozszerzenie, ale w praktyce jest już wszędzie wymagane.

Adres nadawcy w linii `MAIL FROM:` jest inny od tego w *IMF*, na przykład, gdy wiadomość jest wysyłana przez serwer listy dyskusyjnej (nr 5 na rysunku 2.1). Jako adres nadawcy w *SMTP* podawany jest adres zwrotny, gdzie mogą być dostarczane informacje o błędach w doręczeniu (adres nadawcy w wiadomości *IMF* pozostaje niezmieniony). Na przykład na liście `opensuse@opensuse.org` dla wiadomości o identyfikatorze 1234 (nadanym przez serwer) i uczetnika z adresem `alice@example.com` adresem zwrotnym jest `opensuse+bounces-1234-alice=example.com@opensuse.org`.

Adresy odbiorców w liniach `RCPT TO:` mogą różnić się od tych w *IMF*. Na przykład, gdy serwer listy dyskusyjnej wysyła wiadomość do użytkowników, to w *SMTP* podane są ich adresy, a w *IMF* adresatem pozostaje lista.

Jak widać na powyższych przykładach możliwość wystąpienia rozbieżności pomiędzy adresami w *SMTP* i *IMF* jest pożądana. Niestety jednocześnie pozwala to nadawcom niechcianej poczty wpisać dowolny, zmyślony adres nadawcy. Takie działanie nazywa się *email spoofing* i metody przeciwdziałania temu opisałem w rozdziale 2.4.

Dokument RFC 4954 [71] opisuje wykorzystanie *SASL* w *SMTP*. Metody najczęściej używane w publicznie dostępnych serwerach to *LOGIN*, *PLAIN* i *XOAUTH2*, przy czym *LOGIN* i *XOAUTH2* są oznaczone jako przestarzałe. Sprawdziłem to, używając programu `openssl` do nawiązania szyfrowanego połączenia i polecenia `EHLO foo`. W odpowiedzi serwer wysyła między innymi linię zaczynającą się od `250 AUTH`, która dalej zawiera dostępne metody autentykacji (listing 2.4).

Listing 2.4: Sprawdzenie dostępnych metod uwierzytelniania w *SMTP*

```
$ openssl s_client -starttls smtp -connect smtp.example.com:587
albo
$ openssl s_client -connect smtp.example.com:465
S: [...]
S: 250 8BITMIME
C: EHLO foo
S: [...]
S: 250 AUTH LOGIN PLAIN
S: [...]
```

Wyniki dla kilku wybranych serwerów:

- `smtp.gmail.com` – *LOGIN*, *PLAIN*, *XOAUTH2*, *PLAIN-CLIENTTOKEN*, *OAUTHBEARER*, *XOAUTH*,
- `smtp.wp.pl` – *LOGIN*, *PLAIN*,
- `smtp.poczta.onet.pl` – *PLAIN*, *LOGIN*, *XOAUTH2*,
- `smtp.yandex.ru` – *LOGIN*, *PLAIN*, *XOAUTH2*,
- `smtp.mail.yahoo.com` – *PLAIN*, *LOGIN*, *XOAUTH2*, *OAUTHBEARER*,
- `poczta.superhost.pl` – *PLAIN*, *LOGIN*,

`poczta.superhost.pl` nie jest powszechnie używanym, publicznym serwerem pocztowym, ale znajduje się na nim moja prywatna poczta i dlatego dodałem go do zestawienia.

Serwery `outlook.com` i `smtp.mail.me.com` (*icloud*) po wysłaniu `EHLO foo` nie odpowiadały.

Mechanizm szyfrowania połączenia *Opportunistic TLS* (*STARTTLS*), dodany do *SMTP* jako rozszerzenie, opisałem w rozdziale 2.3.

Post Office Protocol – POP3

Post Office Protocol (POP3) [64] pozwala pobierać wiadomości pocztowe z serwera.

Listing 2.5: Przykładowa sesja *POP3*

```
S: +OK POP3 server ready <1896.697170952@dbc.mtview.ca.us>
C: APOP mrose c4c9334bac560ecc979e58001b3e22fb
S: +OK mrose's maildrop has 2 messages (320 octets)
C: STAT
S: +OK 2 320
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
C: RETR 1
S: +OK 120 octets
S: <the POP3 server sends message 1>
S: .
C: DELE 1
S: +OK message 1 deleted
C: RETR 2
S: +OK 200 octets
S: <the POP3 server sends message 2>
S: .
C: DELE 2
S: +OK message 2 deleted
C: QUIT
S: +OK dewey POP3 server signing off (maildrop empty)
```

Głównym założeniem protokołu *POP3* jest istnienie jednego klienta, który pobiera wszystkie wiadomości z serwera i przechowuje u siebie. Serwer ma być tylko pośrednikiem przechowującym przysłane wiadomości, do czasu aż klient je pobierze i usunie.

POP3 obsługuje następujące metody uwierzytelniania:

- Jawne podanie hasła (komendy *USER* i *PASS*).
- Obliczenie skrótu *md5* z połączenia:
  - wartości z nagłówka `+OK POP3 server ready`, która za każdym razem jest inna,
  - hasła,

  Dla przykładu powyżej hasłem jest `tanstaaf`, więc wartość wysłana przy `APOP` jest wynikiem `md5sum("<1896.697170952@dbc.mtview.ca.us>tanstaaf")`.
- Protokół *SASL* [72].

Najczęściej dostępne metody to *USER* (opcja pierwsza) i *PLAIN* (z *SASL*). Sprawdziłem to, używając programu `openssl` do nawiązania szyfrowanego połączenia i polecenia `CAPA`, na które

serwer między innymi zwraca dostępne metody (listing 2.6). Polecenie *CAPA* nie musi działać, ponieważ jest opcjonalnym rozszerzeniem protokołu *POP3* [25].

Listing 2.6: Sprawdzenie dostępnych metod uwierzytelniania w *POP3*

```
$ openssl s_client -starttls pop3 -connect pop3.example.com:110
albo
$ openssl s_client -connect pop3.example.com:995
S: [...]
S: +OK Server ready.
C: CAPA
S: +OK
S: [...]
S: USER
S: SASL PLAIN LOGIN
```

Wyniki dla kilku wybranych serwerów:

- `pop.gmail.com` – *USER*, *PLAIN*, *XOAUTH2*, *OAUTHBEARER*,
- `pop3.wp.pl` – *USER*, *PLAIN*, *LOGIN*,
- `pop3.poczta.onet.pl` – *USER*, *PLAIN*, *LOGIN*, *XOAUTH2*,
- `pop3.yandex.ru` – *USER*,
- `pop.mail.yahoo.com` – *USER*, *PLAIN*, *XOAUTH2*,
- `poczta.superhost.pl` – *USER*, *PLAIN*, *LOGIN*,

Serwer `outlook.com` nie odpowiadał na polecenie *CAPA*, `mail.me.com` (*icloud*) nie obsługuje protokołu *POP3*, żaden ze sprawdzanych serwerów nie podał nagłówka pozwalającego na wykorzystanie metody *APOP*.

Mechanizm szyfrowania połączenia *STARTTLS*, dodany do *POP3* jako rozszerzenie, opisałem w rozdziale 2.3.

Internet Message Access Protocol – IMAP

Internet Message Access Protocol (*IMAP*) [34] pozwala synchronizować zawartość skrzynki pocztowej pomiędzy serwerem i jednocześnie podłączonymi wieloma urządzeniami.

IMAP obsługuje następujące uwierzytelniania:

- Login i hasło.
- Protokół *SASL*.

Najczęściej dostępną metodą jest *PLAIN*. Można to sprawdzić, używając polecenia `openssl` (listing 2.7).

Listing 2.7: Sprawdzenie dostępnych metod uwierzytelniania w *IMAP*

```
$ openssl s_client -starttls imap -connect imap.example.com:143
albo
$ $ openssl s_client -connect imap.example.com:993
S: [...]
S: * OK Example.com server ready.
C: a001 CAPABILITY
S: CAPABILITY IMAP4rev1 ... AUTH=PLAIN AUTH=LOGIN
S: a001 OK
```

Wyniki dla kilku wybranych serwerów:

- `imap.wp.pl` – *PLAIN*,
- `imap.poczta.onet.pl` – *PLAIN*, *LOGIN*, *XOAUTH2*,
- `imap.mail.yahoo.com` – *PLAIN*, *XYMCOOKIEB64*, *XOAUTH2*, *OAUTHBEARER*,
- `smtp.mail.me.com` (*icloud*) – *ATOKEN*, *PLAIN*,
- `poczta.superhost.pl` – *PLAIN*, *LOGIN*,

Serwery `imap.gmail.com`, `imap.yandex.ru` i `smtp.outlook.com` nie odpowiadały na polecenie *CAPABILITY*.

Mechanizm szyfrowania połączenia *STARTTLS*, opisany w rozdziale 2.3), jest częścią *IMAP*.

Hypertext Transfer Protocol – HTTP

Hypertext Transfer Protocol (*HTTP*) nie jest protokołem pocztowym, lecz służy do udostępniania stron internetowych. Wymieniam go jednak tutaj, ponieważ wiele osób korzysta z interfejsów www do serwerów pocztowych (na rysunku 2.1 jest zaznaczony w krokach 1 i 3). W takiej konfiguracji serwer www jako *MUA* komunikuje się z serwerami *SMTP* i *IMAP* [76, 87], natomiast przez *HTTP* udostępnia użytkownikowi interfejs graficzny.

2.3 Protokoły szyfrowania połączeń

Transport Layer Security – TLS

Transport Layer Security (*TLS*) [59] jest protokołem zapewniającym szyfrowane połączenie pomiędzy serwerem i klientem. Po nawiązaniu połączenia można przez niego przesyłać dowolne dane, dlatego wykorzystuje się go jako warstwę pod innymi protokołami aplikacyjnymi. Pozwala wykorzystywać wiele algorytmów kryptograficznych, w tym *RSA*, *ECDSA*, *RSASSA-PSS*, *EdDSA*, *RSASSA-PSS*, *SHA-256*, *SHA-384*, *SHA-512*, *SHA-1*.

Przy nawiązywaniu połączenia *TLS* klient zawsze przeprowadza uwierzytelnianie serwera. W tym celu sprawdza jego certyfikat X.509, który zawiera klucz publiczny, podpisy wydane innymi certyfikatami i metadane takie jak termin ważności, nazwa organizacji czy domeny *Domain Name System* (*DNS*), dla których certyfikat może być użyty. Opcjonalnie serwer może też sprawdzić certyfikat klienta.

Zaufanie do certyfikatów jest oparte na urzędach certyfikacji (*certificate authority*). Są to organizacje publiczne lub prywatne firmy, które zajmują się weryfikacją klientów i podpisywaniem ich certyfikatów. Każdy klient posiada certyfikaty urzędów certyfikacji, które uznaje za zaufane.

Po nawiązaniu połączenia, protokół *TLS* gwarantuje, że przesyłane dane są niemożliwe do odczytania przez osoby trzecie. Jednocześnie też nikt nie może zmodyfikować przesyłanych danych tak, żeby komunikujące się strony tego nie wykryły.

Wcześniejsze wersje standardu nazywały się *Secure Sockets Layer* (*SSL*) i tę nazwę nadal można spotkać w kontekście szyfrowanych połączeń lub nazw bibliotek.

Opportunistic TLS – STARTTLS

Opportunistic TLS (*STARTTLS*) to ogólna nazwa dla rozszerzeń protokołów *SMTP*, *POP3*, *IMAP* i innych pozwalających na włączenie szyfrowania *TLS* w pierwotnie nieszyfrowanym połączeniu [55, 28].

Wadą tego rozwiązania jest możliwość degradacji bezpieczeństwa – atakujący, wykonując atak *man-in-the-middle*, może przechwycić polecenie rozpoczynające *STARTTLS* i nie przesłać go dalej. W ten sposób połączenie pozostaje nieszyfrowane.

Dlatego obecnie rekomendowaną metodą jest używanie protokołów nazywanych *POP3S*, *IMAPS* i *SMTPS*, które działają na innych portach i analogicznie jak *Hypertext Transfer Protocol Secure* (*HTTPS*), wymagają szyfrowania przez *TLS* [53].

SMTP Security via Opportunistic DNS-Based Authentication of Named Entities – SMTP DANE

SMTP Security via Opportunistic DNS-Based Authentication of Named Entities (*SMTP DANE*) [17] umożliwia ochronę przed atakiem *man-in-the-middle* przechwytującym negocjację *STARTTLS* lub modyfikującym rekordy *MX* w zapytaniach do *DNS*.

DANE polega na wykorzystaniu protokołu *Domain Name System Security Extensions* (*DNSSEC*) do zabezpieczenia zapytań *DNS* oraz na opieraniu zaufania do certyfikatów *TLS* na rekordach *DNS TLSA*.

DNSSEC, nie zdobyło szerokiej popularności i nawet jest krytykowane jako gorsze rozwiązanie od obecnej infrastruktury certyfikatów *TLS* [3].

SMTP Strict Transport Security – SMTP STS

SMTP Strict Transport Security (*SMTP STS*) [49], podobnie jak *SMTP DANE*, umożliwia ochronę przed atakiem *man-in-the-middle* przechwytującym negocjację *STARTTLS*, jednak, zamiast polegać na protokole *DNSSEC*, wykorzystuje rekordy *TXT* w *DNS* i *HTTPS*.

Istnienie rekordu w *DNS* sygnalizuje jedynie obecność danych *STS*. Właściwe dane są udostępniane przez serwer protokołem *HTTPS*. Dla serwera pocztowego o adresie `example.com` rekord *TXT* powinien znajdować się w domenie `_mta-sts.example.com`, natomiast właściwe dane powinny być dostępne pod adresem `https://mta-sts.example.com/.well-known/mta-sts.txt`.

Listing 2.8: Przykładowe dane *SMTP STS*

```
version: STSv1
mode: enforce
mx: mail.example.com
mx: *.example.net
mx: backupmx.example.com
max_age: 604800
```

Serwery pocztowe, wymienione w danych *SMTP STS* muszą obsługiwać *STARTTLS* i w przypadku problemów z nawiązaniem szyfrowanego połączenia klient powinien się rozłączyć, zamiast kontynuować bez szyfrowania.

2.4 Przeciwdziałanie email spoofing

Email spoofing to wysyłanie wiadomości, w której adres nadawcy jest sfałszowany.

Wysłanie takiej wiadomości jest możliwe, ponieważ adres nadawcy jest zapisany w *IMF* i *SMTP* nie musi go weryfikować. Co więcej, taka możliwość jest pożądana w przypadku list mailingowych, co opisałem w rozdziale 2.1 w części o protokole *SMTP*.

Sender Policy Framework – SPF

Sender Policy Framework (*SPF*) [38] jest mechanizmem pozwalającym określić serwery pocztowe uprawnione do wysyłania poczty użytkowników w danej domenie.

SPF polega na zapisaniu w *DNS* w domenie adresu nadawcy rekordu *TXT* o określonej strukturze.

Listing 2.9: Przykładowe rekordy *SPF*

```
v=spf1 ip4:192.0.2.0/24 ip4:198.51.100.123 a -all
v=spf1 +mx a:colo.example.com/28 -all
```

W pierwszym przykładzie dozwolone jest wysyłanie poczty z konkretnych adresów *IP* oraz gdy adres *IP* nadawcy zgadza się z rekordem *A* w *DNS*. W drugim przykładzie dozwolone jest wysyłanie z adresu *IP*, który zgadza się z rekordem *MX* lub jest w tej samej podsieci */28* co adres *colo.example.com*. Standard pozwala też tworzyć bardziej zaawansowane reguły.

DomainKeys Identified Mail – DKIM

DomainKeys Identified Mail (*DKIM*) [43] jest protokołem pozwalającym serwerowi pocztowemu podpisywać wysyłane wiadomości.

Klucz publiczny jest publikowany w *DNS* w rekordzie *TXT* w subdomenie `<selektor>._domainkey` (selektor jest parametrem załączonym do podpisu). Format pozwala określać rodzaj klucza i funkcji skrótu. W protokole dozwolone są klucze *RSA* ze skrótami *SHA-1* i *SHA-256*, z adnotacją, że przyszłe wersje protokołu mogą dopuszczać nowe możliwości.

Listing 2.10: Przykładowy rekord *DKIM*

```
"v=DKIM1; p=MIGfMA0GCSqGSIb3[...]PNUYckcQ2QIDAQAB"
```

Serwer pocztowy dodaje do wiadomości w formacie *IMF* nagłówek `DKIM-Signature`, który zawiera podpis treści wiadomości i wybranych nagłówków. Sam nagłówek zawiera wiele par typu klucz-wartość.

Listing 2.11: Przykładowy podpis *DKIM*

```
DKIM-Signature: v=1; a=rsa-sha256; d=example.net; s=brisbane;
   c=simple; q=dns/txt; i=@eng.example.net;
   t=1117574938; x=1118006938;
   h=from:to:subject:date;
   z=From:foo@eng.example.net|To:joe@example.com|
   Subject:demo=20run|Date:July=205,=202005=203:44:08=20PM=20-0700;
   bh=MTIzNDU2Nzg5MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTI=;
   b=dzdVyOfAKCdLXdJOc9G2q8LoXSlEniSbav+yuU4zGeeruD00lszZVoG4ZHRNiYzR
```

Znaczenia niektórych pól: *v* – numer wersji protokołu *DKIM*, *s* – selektor, *h* – lista podpisanych nagłówków, *z* – kopia podpisanych nagłówków (na wypadek, gdyby kolejny serwer pocztowy je zmodyfikował), *bh* – skrót treści wiadomości, *b* – podpis wybranych nagłówków i skrótu treści wiadomości. W tym przykładzie klucz *DKIM* jest opublikowany w *DNS* w domenie `brisbane._domainkey.example.net`.

Domain-based Message Authentication, Reporting, and Conformance – DMARC

Protokół *Domain-based Message Authentication, Reporting, and Conformance* (*DMARC*) [44] nie jest bezpośrednio związany z żadnymi mechanizmami bezpieczeństwa. Ten standard pozwala serwerom pocztowym na wzajemne raportowanie o wiadomościach, które nie przeszły weryfikacji *SPF* lub *DKIM*.

DMARC konfiguruje się przez opublikowanie rekordu *TXT* w subdomenie *_dmarc*. Rekord opublikowany dla domeny *example.com* mówi innym serwerom, co mają robić z wiadomościami przychodzącymi, których adres nadawcy jest w domenie *example.com*.

Listing 2.12: Przykładowy rekord *DMARC*

```
v=DMARC1; p=quarantine; rua=mailto:dmarc-feedback@example.com,
    mailto:tld-test@thirdparty.example.net!10m; pct=25
```

W powyższym przykładzie 25% wiadomości powinny być poddane kwarantannie (na przykład dostarczone do folderu *SPAM*). Zbiorczy raport ma być co dobę wysyłany na adres *dmarc-feedback@example.com* oraz ograniczony do rozmiaru 10MB na adres *tld-test@thirdparty.example.net*.

Authenticated Received Chain – ARC

Authenticated Received Chain (*ARC*) [4] jest protokołem pozwalającym serwerom pośredniczącym (na przykład list dyskusyjnych) dodawać do wiadomości informacje o wyniku weryfikacji przeprowadzonej w momencie jej otrzymania. Wykorzystywane są w tym celu nagłówki podobne do *DKIM*. Standard jest obecnie w fazie projektowania.

Listing 2.13: Przykład nagłówków *ARC*

```
ARC-Seal: i=1; a=rsa-sha256; cv=none; d=lists.example.org; s=dk-lists;
        t=12345; b=TlCCKzgk3TrAa+G77gYYO8Fxk4q/Ml0biqduZJeOYh6+0zhwQ8u/
        lHxLi21pxu347isLSuNtvIagIvAQna9a5A==
ARC-Message-Signature: i=1; a=rsa-sha256; c=relaxed/relaxed; d=
        lists.example.org; h=message-id:date:from:to:subject; s=
```

```
        dk-lists; t=12345; bh=KWSe46TZKCcDbH4klJPo+tjk5LWJnVRlP5pvjXFZYL
        Q=; b=DsoD3n3hiwlrN1ma8IZQFgZx8EDO7Wah3hUjIEsYKuShRKYB4LwGUiKD5Y
        yHgcIwGHhSc/4+ewYqHMWDnuFxiQ==
ARC-Authentication-Results: i=1; lists.example.org; spf=pass
        smtp.mfrom=jqd@d1.example; dkim=pass (512-bit key)
        header.i=@d1.example; dmarc=pass
DKIM-Signature: v=1; a=rsa-sha1; c=relaxed/relaxed; d=d1.example; h=
        message-id:date:from:to:subject; s=origin2015; bh=bIxxaeIQvmOBdT
        AitYfSNFgzPP4=; b=qKjd5fYibKXWWIcMKCgRYuo1vJ2fD+IAQPjX+uamXIGY2Q
        0HjQ+Lq3/yHzG3JHJp6780/nKQPOWt2UDJQrJkEA==
```

Problemem nierozwiązanym przez *ARC* jest kwestia ustalania zaufania pomiędzy serwerami pocztowymi. Jednak w momencie, gdy zaufanie zostanie już ustalone, serwery pocztowe mogą uznawać ważność podpisów *DKIM* i *ARC* innych serwerów.

Message Header Field for Indicating Message Authentication Status

Wiele z mechanizmów opisanych powyżej dodaje swoje nagłówki do wiadomości, w których raportują rezultat swojej weryfikacji. Dokument *RFC* 8601 [42] opisuje nowy nagłówek `Authentication-Results`, który daje jednolity format raportowania dla obecnych i przyszłych mechanizmów.

Przykładowe nagłówki:

Listing 2.14: Przykładowe nagłówki *Authentication-Results*

```
Authentication-Results: example.com;
    auth=pass (cram-md5) smtp.auth=sender@example.net;
    spf=pass smtp.mailfrom=example.net
Authentication-Results: example.com; iprev=pass
    policy.iprev=192.0.2.200
```

Skala wykorzystania

Żeby dowiedzieć się, czy dana domena obsługuje *SPF*, wystarczy sprawdzić rekordy *TXT* tej domeny. Podobnie sprawdzenie dla *DMARC* można wykonać w subdomenie `_dmarc`.

W przypadku *DKIM* sprawdzenie jest trudniejsze, ponieważ trzeba znać wartość selektora, która jest dołączana do wiadomości. Na przykład w przypadku poczty z *gmail.com* ostat-

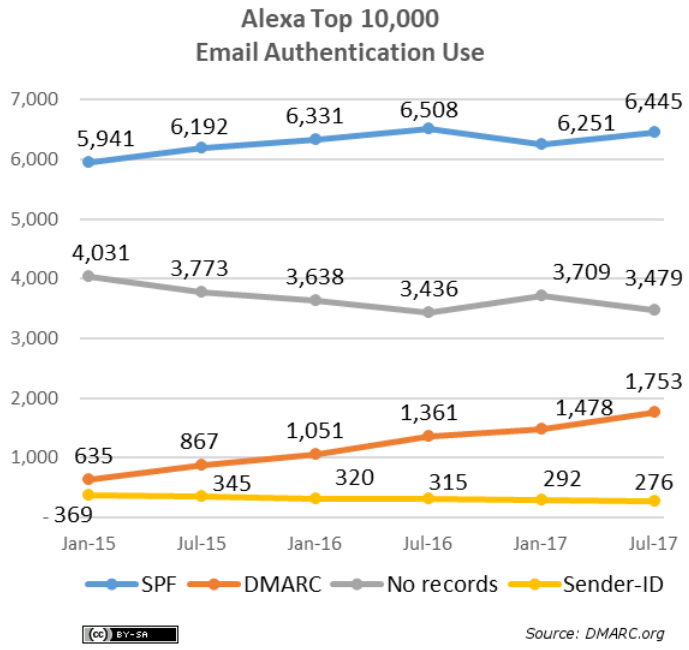
domena	*SPF*	*DMARC*
gmail.com	●	●
wp.pl	●	○
onet.pl	●	○
yandex.ru	●	●
yahoo.com	●	●
mizerski.pl	●	○

Tabela 2.1: Dostępność *SPF* i *DMARC* w wybranych domenach.

nia wiadomość, jaką otrzymałem, zawierała selektor `20161025` (klucz znajduje się w domenie `20161025._domainkey.gmail.com`).

Sprawdziłem dostępność *SPF* i *DMARC* dla kilku wybranych domen. Wyniki znajdują się w tabeli 2.1 po prawej stronie.

Autorzy strony dmarc.org przez jakiś czas badali dostępność w najpopularniejszych domenach według rankingu Alexa. Wykres z wynikami znajduje się na rysunku 2.2.



Rysunek 2.2: Dostępność protokołów *SPF* i *DMARC* wśród 10000 najpopularniejszych domen
Źródło: *https://dmarc.org/stats/alexa-top-sites/*

2.5 Protokoły szyfrowania end-to-end

Przedstawione powyżej mechanizmy skupiają się na zabezpieczeniu komunikacji pomiędzy klientem i serwerem lub pomiędzy serwerami. Problemem jest nadal to, że serwery mają dostęp do całej treści wiadomości, włącznie z możliwością jej modyfikowania.

Rozwiązaniem jest wykorzystanie szyfrowania *end-to-end*, czyli takiego, gdzie operacje kryptograficzne są wykonywane na *MUA* użytkowników. Dzięki temu podejściu treść wiadomości jest zabezpieczona, nawet jeżeli którykolwiek serwer lub połączenie nie są.

Secure/Multipurpose Internet Mail Extensions – S/MIME

Secure/Multipurpose Internet Mail Extensions (*S/MIME*) jest protokołem zabezpieczania wiadomości pocztowych, wykorzystującym infrastrukturę klucza publicznego. Wersja czwarta standardu jest opisana w dokumentach *RFC* 8550 i 8551 [65, 66]. Podobnie jak *TLS*, wykorzystuje certyfikaty *X.509* i *Certificate Revocation Lists* (*CRL*), opisane w dokumencie *RFC* 5280 [13].

Użytkownicy *S/MIME* posiadają swoje certyfikaty, podpisane przez urzędy certyfikacji (*certificate authority*). Zaufanie do certyfikatu, podobnie jak w *TLS*, opiera się na zaufaniu do urzędu certyfikacji.

Format wiadomości Cryptographic Message Syntax – CMS

Wiadomości wykorzystujące standard *S/MIME* są kodowane w formacie *Cryptographic Message Syntax* (*CMS*), który jest opisany w dokumencie *RFC* 5652 [29]. Operacje kryptograficzne opisane są ogólnie, bez wymieniania konkretnych algorytmów, na zasadzie wskazania danych wejściowych oraz miejsca i sposobu zapisu wyników.

Format *CMS* przewiduje osobne algorytmy dla sumy kontrolnej i podpisu wiadomości. Wiadomość może zawierać podpisane atrybuty (*SignedAttributes*), takie jak typ zawartości (*content-type*) lub data złożenia podpisu (*signing-time*), podpisywane w trakcie tworzenia wiadomości. Jeżeli wiadomość zawiera jakiekolwiek podpisane atrybuty, to jednym z nich musi być suma kontrolna treści wiadomości (*message-digest*) – w szczególności może to być jedyny zawarty atrybut. Jeżeli wiadomość zawiera atrybuty wymagające podpisania, to podpis jest tworzony dla samych atrybutów, ponieważ jest wśród nich suma kontrolna wiadomości. W przeciwnym wypadku podpis jest tworzony dla samej treści wiadomości.

Szyfrowanie wiadomości odbywa się w dwóch krokach: najpierw generowany jest klucz symetryczny, którym szyfrowana jest treść, następnie kluczami asymetrycznymi odbiorców szyfrowany jest klucz symetryczny. Dzięki temu dodanie wielu adresatów nie powoduje znacznego zwiększenia rozmiaru przesyłanych danych.

Algorytmy kryptograficzne

Algorytmy używane do zabezpieczania treści wiadomości w *S/MIME* są wymienione w dokumencie *RFC* 8551 i przedstawiłem je w tabeli 2.2.

	Wymagane	Opcjonalne
Suma kontrolna *DigestAlgorithmIdentifier*	– *SHA-256* – *SHA-512* (*RFC* 5754)	
Podpis *SignatureAlgorithmIdentifier*	– *ECDSA* z krzywą eliptyczną *P-256* i funkcją skrótu *SHA-256* – *EdDSA* z krzywą eliptyczną *curve25519* w trybie *PureEdDSA* (*RFC* 8419) – *RSA PKCS #1 v1.5* z funkcją skrótu *SHA-256* (*RFC* 3370)	– *RSASSA-PSS* z funkcją skrótu *SHA-256* (*RFC* 4056)
Szyfr asymetryczny *KeyEncryptionAlgorithmIdentifier*	– *ECDH ephemeral-static* z krzywą eliptyczną *P-256* (*RFC* 5753) – *ECDH ephemeral-static* z krzywą eliptyczną *X25519* z użyciem *HKDF-256* (*RFC* 8418) – *RSA PKCS #1 v1.5* (*RFC* 3370)	– *RSA Optimal Asymmetric Encryption Padding* (*RSAES-OAEP*) (*RFC* 3560)
Szyfr symetryczny *ContentEncryptionAlgorithmIdentifier*	– *AES-128 GCM* i *AES-256 GCM* (*RFC* 5084) – *AES-128 CBC* (*RFC* 3565)	– *ChaCha20-Poly1305* (*RFC* 7905)

Tabela 2.2: Algorytmy kryptograficzne wykorzystywane w *S/MIME*.

Dla algorytmów *RSA* muszą być obsługiwane klucze o rozmiarach pomiędzy 2048 i 4096 bitów. Większe są akceptowalne, ale nie powinny być mniejsze.

W dokumencie *RFC* 8550, w sekcji 4.3 do podpisywania certyfikatów i *CRL* wymienione są te same algorytmy co do podpisywania wiadomości.

Przechowywanie kluczy prywatnych

W dokumencie *RFC* 8550, który opisuje sposób użycia certyfikatów *X.509* w *S/MIME*, w rozdziale 6. napisane jest, że sposób zabezpieczenia klucza prywatnego jest poza jego zakresem. Wynika z tego, że zależy to wyłącznie od implementacji klienta.

Przykładowo program pocztowy *Thunderbird* zabezpiecza klucze tak samo, jak hasła, czyli przy użyciu hasła głównego (*master password*) [33]. W szczególności, jeżeli hasło nie jest ustawione, przechowywany klucz nie będzie zabezpieczony.

Innym rozwiązaniem jest zakodowanie klucza na prywatnego na kluczu sprzętowym lub karcie mikroprocesorowej. Używane są dwa standardy: *NIST Special Publication 800-73-4*[1] oraz *PKCS #11*[2]. Zastosowanie takiej karty pozwala na zwiększenie bezpieczeństwa klucza, ponieważ nie można go wydobyć z karty, a wszystkie operacje kryptograficzne są wykonywane na karcie. Pozwala to dodatkowo na łatwiejsze i bezpieczniejsze używanie klucza na wielu urządzeniach.

Dostarczanie certyfikatów

Standard *S/MIME* umożliwia dołączanie dowolnej ilości certyfkatów do wiadomości. Ponieważ zaufanie do certyfikatu opiera się na podpisie zaufanego urzędu certyfikacji, certyfikat może być dostarczony w wiadomości niezabezpieczonej.

Dokument *RFC* 2798 [74, roz. 2.8] opisuje możliwość dostarczania certyfikatów *S/MIME* przez usługę katalogową *Lightweight Directory Access Protocol* (*LDAP*). Dzięki temu może być używany na przykład w firmach wykorzystujących *LDAP*, ale tylko do zabezpieczenia komunikacji wewnętrznej.

Unieważnianie kluczy

Unieważnianie kluczy odbywa się przez publikację list certyfikatów unieważnionych (*Certificate Revocation Lists*) podpisanych przez urząd certyfikacji. *CRL* może być też dołączone do wiadomości razem z certyfikatami.

Certyfikaty *X.509* mogą zawierać odnośniki do *CRL* [13, roz. 4.2.1.13] w postaci *Uniform Resource Identifier* (*URI*, *RFC* 3986). W szczególności pliki z *CRL* mogą być udostępniane przez protokoły *HTTP*, *FTP* lub *LDAP*. Jeżeli taki odnośnik jest obecny, klient pocztowy przed każdym użyciem certyfikatu powinien sprawdzić, czy nie został on unieważniony.

[1] https://csrc.nist.gov/publications/detail/sp/800-73/4/final
[2] https://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html

OpenPGP

OpenPGP [63, 16] jest standardem stworzonym przez twórców oprogramowania *Pretty Good Privacy*. Opisuje on formaty i protokoły używane do szyfrowania i podpisywania wiadomości.

Używanie *OpenPGP* opiera się na infrastrukturze klucza publicznego i sieci zaufania (*web of trust*). Uproszczony schemat budowania sieci zaufania:

- Alicja i Bob generują swoje klucze prywatne i powiązane z nimi klucze publiczne. Do kluczy dołączają swoje tożsamości składające się z imienia, nazwiska, adresu *email* i opcjonalnie pseudonimu (można mieć więcej niż jedną tożsamość, dzięki temu można do jednego klucza dodać adres *email* prywatny, z uczelni, z pracy, itp).
- Alicja i Bob wymieniają się swoimi kluczami publicznymi. Metoda transportu jest dowolna ponieważ z założenia nie jest uznawana za bezpieczną (po drodze atakujący może podmienić klucz).
- Alicja i Bob weryfikują je, najlepiej spotykając się twarzą w twarz, poprzez sprawdzenie „odcisków palca" (*fingerprint*) kluczy (szczegóły o odciskach znajdują się poniżej). Alicja po sprawdzeniu, że otrzymała poprawny klucz Boba, ustawia w programie swój poziom zaufania (do wyboru są: brak zaufania, częściowe i pełne) i podpisuje jego klucz publiczny swoim kluczem prywatnym. Bob robi to samo z kluczem Alicji.
- Alicja i Bob odsyłają sobie podpisane klucze.
- Bob i Carol powtarzają powyższą procedurę.
- Carol chce się skontaktować z Alicją. Alicja przysyła jej swój klucz publiczny, na którym znajduje się już podpis Boba. Dzięki temu Alicja i Carol nie muszą się spotykać i weryfikować swoich kluczy, ponieważ mają zbudowany łańcuch zaufania przez klucz Boba.

Algorytmy kryptograficzne

Algorytmy używane w *OpenPGP* są wymienione w dokumencie *RFC* 4880 [16] w rozdziale 9 i przedstawiłem je w tabeli 2.3.

Kolejne dokumenty *RFC* dodają nowe algorytmy: 5581 [70] dodaje algorytm szyfrowania symetrycznego *Camellia* z kluczami długości 128, 192 i 256 bitów, 6637 [35] dodaje klucze asymetryczne i algorytmy *Elliptic Curve Diffie-Hellman* (*ECDH*) i *Elliptic Curve Digital Signature Algorithm* (*ECDSA*), wykorzystujące krzywe eliptyczne *P-256*, *P-384* i *P-521*.

	Wymagane	Opcjonalne	Przestarzałe
Klucze asymetryczne	*DSA* (tylko podpisy) *Elgamal* (tylko szyfrowanie)	*RSA*	*RSA* (tylko podpisy) *RSA* (tylko szyfrowanie)
Klucze symetryczne	*TripleDES*	*IDEA* *CAST5* *Blowfish* *AES-128* *AES-192* *AES-256* *Twofish*	
Kompresja		*ZIP* *ZLIB* *BZip2*	
Suma kontrolna	*SHA-1*	*RIPE-MD/160* *SHA-256* *SHA-384* *SHA-512* *SHA-224*	*MD5*

Tabela 2.3: Algorytmy kryptograficzne używane w *OpenPGP*.

Klucz główny i podklucze

Klucz *OpenPGP* składa się z klucza głównego i dowolnej liczby podkluczy. Podklucze są zawsze podpisane przez klucz główny.

Klucze mogą mieć przypisane różne role (jedną lub więcej). Najważniejsze z nich to:

- certyfikacja (podpisywanie innych kluczy),
- szyfrowanie,
- podpisywanie,
- uwierzytelnianie.

Oprócz tego implementacje nakładają dodatkowe ograniczenia, na przykład *GnuPG* [81] nie pozwala na stworzenie podklucza do certyfikacji.

Dzięki podkluczom możliwe jest na przykład:

- Zabezpieczenie klucza głównego przez ograniczenie jego roli do tworzenia podkluczy i przechowywanie go poza maszyną używaną do codziennej pracy.
- Udostępnienie wielu osobom w organizacji podklucza do odszyfrowywania wiadomości wysłanych na ogólny adres (na przykład info@firma.com). Klucz do podpisywania wiadomości z takiego adresu może być udostępniony ograniczonej liczbie osób albo w ogóle nie istnieć.

Odcisk palca klucza

Klucz publiczny OpenPGP, oprócz samego klucza kryptograficznego, zawiera także informacje o właścicielu (tożsamości), w tym dane osobowe czy np. zdjęcie, i podpisy złożone przez inne klucze. To powoduje, że plik z kluczem może być duży i zmieniać się w czasie (dodawane mogą być nowe tożsamości i podpisy). Żeby ułatwić weryfikację przynależności klucza publicznego do prywatnego, używa się „odcisku palca".

W OpenPGP czwarta wersja odcisku palca jest zdefiniowana jako skrót *SHA-1* ciągu następujących danych [16, roz. 12.2]:

- stała liczba `0x99` (1 bajt),
- rozmiar w bajtach pozostałych danych (2 bajty),
- liczba `4` – numer wersji odcisku (1 bajt),
- data stworzenia klucza (4 bajty),
- identyfikator algorytmu klucza (1 bajt),
- klucz publiczny (format i rozmiar zależą od algorytmu).

Na przykład odcisk mojego klucza to A903 6A19 7959 3292 CC5B 7BBC CCF4 9B90 CAF7 07C1.

Ważne jest, żeby przy weryfikacji porównywać cały odcisk. Programy często używają ID klucza (ostatnie 64 bity odcisku, np. CCF4 9B90 CAF7 07C1) lub nawet krótkiego ID (ostatnie 32 bity odcisku, np. CAF7 07C1) do identyfikacji kluczy. Problem w tym, że bardzo łatwo jest wygenerować nowy klucz o zadanym krótkim ID [80]. Według `https://evil32.com/` generowanie takiego klucza zajmuje 4 sekundy. Na dowód autorzy wygenerowali prawie 24 tysiące fałszywych kluczy [20].

Serwery kluczy

W celu ułatwienia wymiany kluczy publicznych używa się serwerów kluczy. Można na nich publikować klucze publiczne, wyszukiwać po ID lub tożsamościach i przeglądać powiązania między kluczami (podpisy).

Serwery aktualizują dane między sobą. Dzięki temu wystarczy opublikować swój klucz na jednym z nich – klucz zostanie wysłany na pozostałe automatycznie.

Zamiast wybierać konkretny serwer, można użyć adresu `http://pool.sks-keyservers.net/`, który przekieruje na jeden z serwerów.

Serwery kluczy służą jedynie do ułatwienia wymieniania się kluczami. W żadnym stopniu nie zwiększają bezpieczeństwa, np. nie weryfikują tożsamości zapisanej w kluczu. Po pobraniu klucza z serwera trzeba zweryfikować poprawność odcisku palca, jak jest to opisane w poprzednim podrozdziale.

Unieważnianie kluczy

W sytuacji, gdy klucz prywatny wpadnie w niepowołane ręce i nie może być już uznawany za zaufany, właściciel może opublikować specjalną wiadomość oznajmującą unieważnienie klucza. Można też stworzyć ją, zanim będzie potrzebna i zachować na później (na przykład na wypadek, gdy klucz prywatny przechowywany na kluczu sprzętowym zostanie zgubiony). W sytuacji kompromitacji klucza prywatnego taką wiadomość można opublikować na serwerze kluczy. Można też próbować wysłać ją do wszystkich osób, które mogą go jeszcze wykorzystywać.

Głównym problemem z unieważnianiem kluczy jest związany dostarczenie informacji do wszystkich użytkowników posiadających klucz publiczny, ponieważ nie ma żadnego mechanizmu wspierającego taką ewentualność. Publikacja wiadomości o unieważnieniu na serwerze kluczy zwiększa szansę, nie daje gwarancji, ponieważ nie ma narzuconego na użytkowników lub oprogramowanie obowiązku odpytywania serwerów o aktualizacje kluczy.

Termin ważności

Przy tworzeniu klucza można ustalić jego termin ważności lub ustawić ważność bezterminową. Można to zmienić później w dowolnym momencie. Po każdej takiej zmianie klucz publiczny trzeba ponownie opublikować. Kiedy termin ważności klucza minie, wszyscy posiadacze klucza publicznego muszą go zaktualizować. Pozwala to na wymuszenie regularnego aktualizowania kluczy przez użytkowników, zwiększając szansę na dostarczenie w razie konieczności wiadomości unieważniającej.

Serwis keybase.io

Podstawowym problemem przy korzystaniu z *OpenPGP* jest kwestia zaufania. W założeniu *OpenPGP* działa tak, że jeżeli Alicja chce rozmawiać z Bobem, to najpierw muszą się spotkać twarzą w twarz i potwierdzić swoje adresy e-mail i odciski kluczy. Daje to najwyższy poziom zaufania, ale jest uciążliwe, a czasami niewykonalne (jeśli skontaktować się chcą osoby mieszkające na przykład po dwóch stronach oceanu).

Z pomocą mogą przyjść różne strony internetowe. Jeżeli Bob ma konta w serwisach społecznościowych, repozytoriach kodu lub własną stronę internetową, to może na nich umieścić publiczne wiadomości potwierdzające jego tożsamość, podpisane jego kluczem. Mallory, który chciałby znaleźć się w komunikacji pomiędzy Alicją i Bobem, musiałby podmienić wiadomości we wszystkich tych serwisach, co jest bardzo trudne (zakładając, że Bob stosuje dobre praktyki bezpiecznego logowania, których Mallory nie może złamać).

Serwis keybase.io [36] dostarcza narzędzia automatyzujące tworzenie, znajdowanie i weryfikowanie takich wiadomości.

Przykładowa wiadomość potwierdzająca moją tożsamość, zamieszczona pod adresem https://etam-software.eu/.well-known/keybase.txt:

```
==================================================================
https://keybase.io/etam
------------------------------------------------------------------

I hereby claim:

  * I am an admin of https://etam-software.eu
  * I am etam (https://keybase.io/etam) on keybase.
  * I have a public key with fingerprint A903 6A19 7959 3292 CC5B  7BBC CCF4 9B90
    CAF7 07C1

To do so, I am signing this object:

{
  "body": {
    "key": {
      "eldest_kid":
          "0101101d1dd2da5307c400305ffcaa7aab698624bbcd79395fd13353d730a49daca90a",
      "fingerprint": "a9036a1979593292cc5b7bbcccf49b90caf707c1",
      "host": "keybase.io",
      "key_id": "ccf49b90caf707c1",
      "kid":
          "0101101d1dd2da5307c400305ffcaa7aab698624bbcd79395fd13353d730a49daca90a",
      "uid": "680a0f679ffe4d10d0cab6faf54f0919",
      "username": "etam"
    },
    "service": {
      "hostname": "etam-software.eu",
      "protocol": "https:"
    },
    "type": "web_service_binding",
    "version": 1
  },
  "ctime": 1539122284,
  "expire_in": 157680000,
  "prev": "eaf691a2d29b2f86ebcb126966b4242e3636e8c1eb23d1cbc87b491c4921470d",
  "seqno": 24,
  "tag": "signature"
}

which yields the signature:

-----BEGIN PGP MESSAGE-----

owGtUl1oFUcUvldjEtMmBBEptrF2qQiS5O7M7N9c0kIJWB8qMVRQY8xm/va6Jrl7
s7s3epsGay3aNj6ESBuUKEJDSYiiIPEPIyiimEgUNVHS1giKxgf1oRiMLaWzoUIf
+thhYJhzvu873zmcruL5scK4/c6Twv5Va47ER5/S2ObhD712hXo8pyTblSYx94hm
LoLQbnK5klRUoAJ5OeAccqIj1WSaqiJVdxxGiEkINbBlQI1Sxk2MsO5wgJCOuIlU
omFOGMEqUcoVx02nhJ/x3XQoZWUQGQRgE+sYQQwZO6kpNRhzNEyxyohjylJAErd6
QcSQ5igJRKXryZj82HP2/gP/P/vOzskZlkpUxzCx4wiNA5XLitRwiKNrjooBjoCB
8NOkRUi0CEmLOlGuyEiby0Q01KiLf2UrAs8JtxNfVIqs5GZ8L/SY1yyzW8MwEyQj
dpjLRPDtgtr/CNnUTXM5R8loE37gemklCSSShW6kDHSEAYTQ0soVsSPj+sJ2I4Ru
```

```
SvvyRHVEW+SAOAYGZBPkEFPoWIagjAJoYMOgGtSgQAYyhMWAoBBxwCizTKphwDQM
gWaqXIl6a017ShLKWiFJSdHATaVJmPWF0lH0XRzlxeKFsbIlH+R9QYvGuw8uzPHB
neLN8i2YF21erGhh6ZvIi6Dgr49ff13SZz0YWY6up7z1uxOnGlf2rBi73lg31jn+
cvHmX5p6R7rW3Nj1TFs+VlC2rW+85LbbW3Fo9OFFQH6zYN4P+uTOgU9a7MRU9vN7
Ly8Xt/w0NKk76LOi2Xa2/9Kr3reyyxbUvlvGq8qqvjxWWbx3x5aG+PCvu0u8E/mt
6/Zc6e8avrO65vzjQX/v1b7TJ35cW1/TeCj16bNdbz/CdehC7XuLlnZUZ75K2D9b
J3uq2Z6GR/2LJ75/f9/9wy/W69n67qN/nO18OF09cHd2w/GbqarCZbfOeSUzF54f
yH+cGNrXmX8mN/Hn6tu5a/bBnhsb66Z+/7ZvKtY91BOkPmpOfDNdOrOh9G8=
=mvxS
-----END PGP MESSAGE-----

And finally, I am proving ownership of this host by posting or
appending to this document.

View my publicly-auditable identity here: https://keybase.io/etam

==================================================================
```

Protokół Autocrypt

Autocrypt [5] jest specyfikacją dla klientów poczty (*MUA*) opisującą sposób stopniowego wprowadzania szyfrowania do poczty elektronicznej.

Podstawą działania *autocrypt*-a jest dołączanie klucza publicznego nadawcy w nagłówku każdej wiadomości. W ten sposób użytkownicy automatycznie wymieniają się kluczami, bez pomocy serwerów kluczy. Specyfikacja dalej określa na przykład, kiedy można włączyć szyfrowanie wiadomości lub jak odpowiadając na wiadomość uniknąć przesyłania jawnym tekstem treści, która w poprzedniej wiadomości była zaszyfrowana.

Obecnie obowiązująca, pierwsza wersja specyfikacji całkowicie pomija budowę sieci zaufania. Ochrona przed aktywnymi napastnikami, którzy potrafią modyfikować przychodzące wiadomości, ma zostać dodana w przyszłych wersjach.

Przechowywanie kluczy prywatnych

Dokument *RFC* 4880 nie definiuje, jak klucze prywatne mają być zabezpieczone. Opisuje jednak mechanizm *String-to-Key* (*s2k*), który pozwala wygenerować klucz symetryczny przy użyciu funkcji skrótu na podstawie hasła, opcjonalnie *soli* i liczby iteracji, który może być wykorzystany do zabezpieczenia klucza prywatnego.

Pakiet *GnuPG* [81] w wersji 2.2.20 przechowuje klucze, domyślnie szyfrując je algorytmem *AES-128* w trybie *CBC* i zabezpiecza integralność skrótem *SHA-1*. Opcjonalnie może być użyty tryb *OCB* z 12-bajtowym *nonce*, który zapewnia jednocześnie poufność i integralność. Klucz szyfrujący jest generowany mechanizmem *s2k* z funkcją skrótu *SHA-1*, 16-bajtową *solą* i liczbą iteracji skalibrowaną na czas 100 milisekund.

Oprócz tego możliwe jest użycie kart mikroprocesorowych. Używany jest do tego standard *OpenPGP Card* [24]. Tak samo jak w *S/MIME*, zastosowanie takiej karty pozwala na zwiększenie bezpieczeństwa klucza, ponieważ nie można go wydobyć z karty, a wszystkie operacje kryptograficzne są wykonywane na karcie. Pozwala to dodatkowo na łatwiejsze i bezpieczniejsze używanie klucza na wielu urządzeniach.

Problemy

OpenPGP, mimo że ma już 20 lat (dokument RFC 2440 został opublikowany w 1998 roku), nadal jest używany przez nielicznych entuzjastów technologii i bezpieczeństwa. Niektórzy, mimo wieloletniego zaangażowania, tracą nadzieję i opisują fundamentalne problemy, które ich zdaniem powodują, że *OpenPGP* nie ma szans na zdobycie popularności [1, 31, 88].

Najczęściej wymieniane problemy:

- Łatwo o degradację bezpieczeństwa. Na przykład gdy Alicja wyśle zaszyfrowaną wiadomość do Boba, Bob może wysłać niezaszyfrowaną odpowiedź, w której znajdzie się cytowana treść wiadomości Alicji.
- OpenPGP zabezpiecza tylko treść wiadomości. Pozostałe metadane, takie jak nadawca, odbiorca, data, pozostają jawne.
- Brak *forward secrecy*. Gdy atakujący zdobędzie klucz, może odszyfrować całą przeszłą i przyszłą komunikację ofiary.
- Jawność sieci zaufania. Z publicznych serwerów kluczy można dowiedzieć się, kto podpisał czyj klucz i kiedy. To duże zagrożenie dla prywatności.
- Problem z długotrwałym utrzymaniem bezpieczeństwa klucza prywatnego. Używanie OpenPGP stoi w sprzeczności z dobrymi praktykami zabezpieczania dostępu hasłami. Zgodnie z nimi hasła należy co jakiś czas zmieniać i nie używać tego samego w różnych usługach. Żeby móc korzystać z jednej usługi z wielu urządzeń, na każdym trzeba mieć kopię klucza. Oprócz tego powszechną praktyką jest używanie tego samego klucza do wielu usług. Teoretycznie można używać podkluczy w celu ochrony klucza głównego, jednak żaden z powszechnie stosowanych graficznych interfejsów nie udostępnia takiej funkcji.

Podsumowanie

Tak jak poczta elektroniczna, *OpenPGP* ma już swoje lata. Mimo wielu wad i braku szerokiego rozpowszechnienia jest, moim zdaniem, najlepszym z dostępnych rozwiązań.

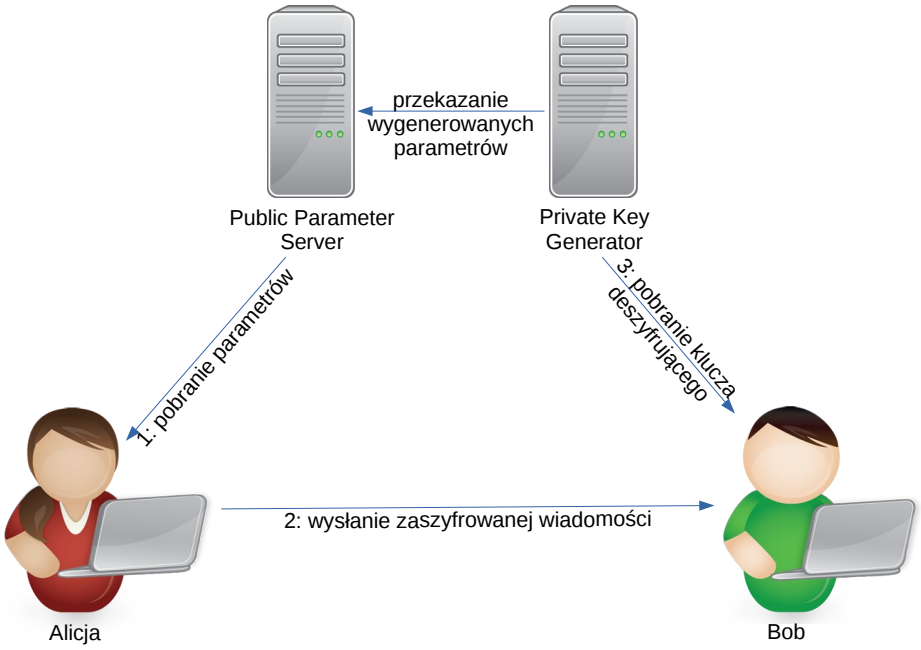
Identity-Based Encryption

Identity-Based Encryption jest kryptosystemem, w którym nadawca może zaszyfrować wiadomość, wykorzystując jakąś publiczną informację o odbiorcy, na przykład jego adres *email*. Przykładem implementacji takiego systemu jest *Voltage SecureMail* [85].

Architektura

Architektura systemu używającego *Identity-Based Encryption* jest opisana w dokumencie *RFC* 5408 [67]. Dokument *RFC* 5409 [69] opisuje sposób jego użycia z formatem zapisu wiadomości *CMS* [29].

W tym systemie przesłanie wiadomości składa się z następujących kroków (rysunek 2.3):



Rysunek 2.3: Wysłanie wiadomości w systemie *Identity-Based Encryption*.

1. Alicja pobiera publiczne parametry z serwera *Public Parameter Server* (*PPS*). Połączenie musi być zabezpieczone protokołem *TLS* 1.2 lub nowszym. Parametry są zakodowane w formacie *ASN.1*. Zawierają między innymi termin ważności, co pozwala na okresową ich zmianę.
2. Alicja tworzy klucz publiczny, którym zaszyfruje wiadomość. Składa się on z danych identyfikujących parametry, adresu *email* Boba oraz znacznika czasu, po którym Bob będzie mógł odszyfrować wiadomość. Alicja szyfruje wiadomość tym kluczem i wysyła do Boba.
3. Bob odbiera wiadomość od Alicji. Jeżeli Alicja użyła klucza publicznego, którego Bob jeszcze nie spotkał, kontaktuje się z serwerem *Private Key Generator* (*PKG*), żeby pobrać odpowiadający mu klucz prywatny. Połączenie musi być zabezpieczone protokołem *TLS* 1.2 lub nowszym. Serwer musi obsługiwać metodę uwierzytelniania *HTTP Basic Auth*, ale może też obsługiwać inne. Dane przesyłane w zapytaniu i odpowiedzi wykorzystują język *XML* i są opisane w dokumencie *RFC* 5408.

Kryptografia

System oparty jest na czterech funkcjach:

- *params, master-key = setup(k)* – funkcja tworząca publiczne parametry (`params`) i klucz używany przez *Private Key Generator* (*PKG*).
- *priv_key = extract(params, master-key, ID)* – funkcja tworząca klucz deszyfrujący.
- *ciphertext = encrypt(params, ID, plaintext)* – funkcja szyfrująca.
- *plaintext = decrypt(params, ciphertext, priv_key)* – funkcja deszyfrująca.

Dokumenty *RFC* 5091 i 6508 [68, 27] opisują algorytmy nazwane *Boneh-Franklin*, *Boneh-Boyen* i *Sakai-Kasahara* (od nazwisk ich twórców), które realizują powyższe funkcje.

Podsumowanie

Taki system ma zastosowanie w dużych organizacjach, gdzie administrator *PKG* może zarządzać dostępem użytkowników do kluczy. Z jednej strony można w razie potrzeby uniemożliwić użytkownikowi odszyfrowywanie nowych wiadomości, z drugiej daje to możliwość przeprowadzenia audytu lub dostęp do zaszyfrowanej zawartości programom antywirusowym.

Rozdział 3

Przegląd alternatywnych rozwiązań

W tym rozdziale opiszę alternatywne protokoły komunikacji asynchronicznej. Tak jak w poprzednim, opiszę ich architekturę, wykorzystywane algorytmy i rozwiązania problemów bezpieczeństwa.

3.1 Protokół Dark Internet Mail Environment – DIME

Dark Internet Mail Environment (*DIME*) [48] jest projektem założonym przez Ladara Levisona, właściciela firmy Lavabit.

W latach 2004-2013 Lavabit dostarczał usługi poczty elektronicznej chroniącej prywatności użytkowników. Poczta przychodząca była szyfrowana kluczem publicznym użytkownika i tylko użytkownik mógł odszyfrować i odczytać oczekujące na niego wiadomości [46]. Gdy w 2013 roku sąd nakazał Ladarowi wydanie kluczy *SSL* organom ścigania, w związku z dochodzeniem w sprawie Edwarda Snowdena, ten zdecydował o zamknięciu usługi [2].

DIME jest opracowywanym przez Ladara standardem poczty, który, chociaż nie jest kompatybilny ze stosowanymi obecnie standardami, jest dostatecznie do nich podobny, żeby umożliwić łatwą i szybką adaptację. Specyfikacja jest jeszcze w wielu miejscach niekompletna, ale uznałem, że nawet w obecnej postaci projekt jest warty opisania w tej pracy.

Sygnety

Nowością wprowadzaną przez *DIME* jest mechanizm dystrybucji i weryfikacji kluczy. Każdy serwer i użytkownik posiadają klucze asymetryczne *Ed25519* [6] do podpisywania i *secp256k1* [10] do szyfrowania danych. Klucze publiczne dostępne są w postaci sygnetów (ang. *signet*), które zawierają dodatkowe metadane i podpisy.

Sygnet serwera (nazywany sygnetem organizacji) jest powiązany z jego domeną (np. example.com). W sygnecie mogą być zawarte dodatkowe informacje, takie jak dane adresowe i kontaktowe firmy, adres do zgłaszania nadużyć, czy adres interfejsu www. Dane zawarte w sygnecie podpisane są tylko powiązanym kluczem prywatnym.

Klucz publiczny sygnetu serwera musi być opublikowany w *DNS* w rekordzie *DIME* albo w rekordzie *TXT* w subdomenie `_dime`. Dane zapisane w *DNS* są w formacie *management record*, w którym oprócz klucza publicznego (*primary organization key*, w skrócie *pok*) mogą się znaleźć dodatkowe informacje, takie jak podpis certyfikatu *TLS* kluczem sygnetu, dodatkowe domeny zawierające *management record*, dodatkowe adresy serwerów *DMTP*, czy termin ważności sygnetu.

Listing 3.1: Przykładowy *management record*

```
1 tls=VvkMypjiECY3vZg/2xbBMd/Sftgr9N31YG4NdWrtM2bQnE+hFSfwOODlfyIB2C8uoskDMmX6bOtino
    VLrmGOBA pok=QD8JiZS92RbtQFMZeTTkqHyAczoSgNYvgBCZLkPuOyQG pol=mixed
    sub=strict dx=dmtp.example.tld syn=mirror.example.tld ref=1 exp=30 ver=1
```

Weryfikacja sygnetów i certyfikatów TLS

Sygnet serwera i jego certyfikat *TLS* przed użyciem muszą być zweryfikowane. Weryfikacja opiera się na kilku elementach, przy czym nie wszystkie muszą być na raz obecne:

- Zabezpieczenie *management record* przez *DNSSEC*.
- Podpis certyfikatu *TLS* w *management record*.
- Weryfikacja certyfikatu *TLS* przez urząd certyfikacji.
- Weryfikacja certyfikatu *TLS* przez protokół *Online Certificate Status Protocol (OCSP)*.

Wymagania weryfikacji przedstawiłem w tabeli 3.1.

Ponieważ *DNSSEC* nie jest powszechnie używany, jako alternatywę autor proponuje globalny rejestr udostępniany z serwerów nadzorowanych przez organizację *Dark Mail Alliance*. Jednak to rozwiązanie jest dopiero projektowane i nie ma podanych szczegółów.

DNSSEC	podpis *TLS* przez sygnet	podpis *TLS* przez *CA*	*OCSP*	wynik
✔	✔			✔
✔	brak	✔	✔	✔
✔	brak	✗		✗
✔	✗			✗
✗	✔	✔		✔
✗	✔	✗		✗
✗	brak	✔	✔	✔
✗	brak	✗		✗

Tabela 3.1: Wymagania weryfikacji w *DIME*. [48, strona 95]

Format wiadomości D/MIME

Podstawową zmianą wprowadzoną przez *DIME* jest podział przesyłanej wiadomości na fragmenty, które mogą być zaszyfrowane różnymi kluczami. Dzięki temu np. serwer poczty wychodzącej może tylko odczytać informacje pozwalające na przekazanie wiadomości do serwera poczty odbiorcy. Serwer poczty odbiorcy dowie się tylko, do którego użytkownika adresowana jest wiadomość. Natomiast żaden z serwerów pośredniczących nie odczyta treści wiadomości.

Protokoły DMTP i DMAP

Protokół *DMTP* jest oparty o *SMTP*. Główne zmiany to obowiązkowe szyfrowanie połączenia przez *TLS*, dodanie komend pozwalających pobierać i weryfikować sygnety, oraz wyłączenie uwierzytelniania użytkownika. Zamiast hasła, użytkownik jest uwierzytelniany na podstawie podpisu zawartego w wiadomości w formacie *D/MIME*.

Protokół *DMAP* nie został jeszcze sformalizowany w dokumentacji. Z krótkiego opisu wynika, że będzie oparty o *IMAP*, będzie wymagał szyfrowania połączenia, a uwierzytelnianie użytkownika nie będzie polegało na wysłaniu niezabezpieczonego hasła do serwera. Oprócz tego protokół zostanie rozszerzony o operacje zarządzania sygnetami.

3.2 Aplikacja BitMessage

Bitmessage [86] jest systemem wymiany wiadomości opartym o sieć *p2p*, w której użytkownicy są identyfikowani przez klucz publiczny Secp256k1 [10].

W *Bitmessage* adresem jest skrót `ripemd160(sha256(klucz_publiczny))` z metadanymi o protokole, zakodowany w base58 [8], z prefiksem `BM-`. Sieć pozwala pobrać klucz publiczny dla danego adresu. W celu ochrony przed atakami typu Sybil [84] wymagane jest, żeby pierwszy bajt w otrzymanym skrócie był równy zero.

Przykładowy adres: `BM-2nTX1KchxgnmHvy9ntCN9r7sgKTraxczzyE`.

Wysłana wiadomość jest zaszyfrowana kluczem odbiorcy i nie zawiera jawnie podanego adresu. Wiadomości są przekazywane do wszystkich użytkowników sieci (gdy liczba użytkowników rośnie, sieć automatycznie dzieli się na „strumienie" pozwalające na lepszą skalowalność). Tylko odbiorca jest ją w stanie odszyfrować. Odbiorca może odesłać potwierdzenie otrzymania wiadomości.

Użytkownicy sieci przechowują przekazywane wiadomości przez 2 dni. Jeżeli odbiorca nie jest podłączony do sieci w momencie jej wysłania, to przez ten czas ma szansę ją otrzymać. Po tym czasie nadawca, jeżeli nie otrzymał potwierdzenia odbioru, może ponownie wysłać wiadomość.

Aby powstrzymać pojawianie się spamu w sieci, wymagane jest, by każda wiadomość zawierała dowód pracy (*proof-of-work*) [9]. Poziom trudności jest zapisany na stałe w programie (może się zmieniać przy wydawaniu nowych wersji) i wybrany tak, by jego stworzenie zajmowało około 4 minut.

Oprócz komunikacji bezpośredniej można też publikować wiadomości bez adresata jako ogłoszenia. Wiadomości są szyfrowane kluczem stworzonym na podstawie klucza publicznego nadawcy. Wystarczy znać adres nadawcy, dodać go do listy adresów obserwowanych, by móc odszyfrowywać jego ogłoszenia [12].

Kanały (*Chan*) [11] działają jak zdecentralizowana lista dyskusyjna. Każdy użytkownik kanału posiada kopię klucza prywatnego, deterministycznie generowanego na podstawie hasła. Na kanale można wysyłać wiadomości, podając prywatny adres jako nadawcę lub adres kanału dla zachowania anonimowości.

3.3 Aplikacja RetroShare

RetroShare [62] jest siecią rozproszoną typu *friend-to-friend*, czyli taką, w której użytkownicy łączą się tylko, gdy sobie ufają i wymienili się wcześniej swoimi kluczami. Umożliwia też komunikację ze znajomymi znajomych. Wykorzystuje protokoły *TLS* i *OpenPGP*.

W ramach jednej aplikacji udostępnia wiele metod komunikacji, takich jak chat, poczta, forum, współdzielenie plików itp. W wersji 0.6 dodany został *General eXchange System* (*GXS*)[1], który umożliwia przekazywanie danych z wykorzystaniem innych zaufanych użytkowników jako pośredników. Między innymi poczta korzysta z tego podsystemu.

General eXchange System przekazuje pakiety pomiędzy zaufanymi użytkownikami. Podobnie jak w protokole *IP*, *GXS* utrzymuje tablicę routingu, dzięki której wie, przez kogo ma wysyłać dane. Niedostarczone wiadomości przechowywane są przez maksymalnie 2 dni.

Przesyłana wiadomość jest cała zaszyfrowana kluczem odbiorcy. Jedyna jawna informacja, to adres, na jaki ma być dostarczona.

Głównym źródłem zagrożenia bezpieczeństwa w komunikacji przez *RetroShare* jest podejście użytkowników do kwestii zaufania. Obecnie, ponieważ *RetroShare* nie ma wielu użytkowników, można znaleźć fora (na przykład `https://www.reddit.com/r/retroshare`), gdzie różne osoby publikują swoje klucze, żeby inni mogli dodawać ich jako znajomych. To podważa podstawowe założenie, że znajomi są zaufani i otwiera drogę do ataków na przykład odmowy dostępu (*denial of service*).

3.4 Aplikacja Freemail

Freemail[22] jest systemem pocztowym działającym jako rozszerzenie rozproszonego systemu plików *Freenet*.

Freenet

Freenet [23] jest rozproszonym systemem plików. Przechowywane w nim dane są dzielone na fragmenty o rozmiarze 32kB i przechowywane przez innych użytkowników sieci, z zachowaniem redundancji.

[1] `https://retroshareteam.wordpress.com/2015/06/08/version-0-6-is-out/`

Dane mogą być umieszczane w sieci i pobierane z wykorzystaniem różnych rodzajów kluczy:

***Content Hash Key* (*CHK*)** – Podstawowa metoda przechowywania niezmiennych danych. Dane są zaszyfrowane indywidualnym kluczem symetrycznym i identyfikowane skrótem kryptograficznym już zaszyfrowanych danych.

Klucz pozwalający pobrać dane składa się z wyżej opisanego skrótu kryptograficznego, klucza deszyfrującego i informacji o tym, jakie algorytmy kryptograficzne zostały wykorzystane.

Obecnie używane algorytmy to *SHA-256* do obliczania skrótu i *AES-256* w trybie *CTR* z kodem *HMAC-SHA256* do szyfrowania. Wspierany też jest tryb *PCFB* [77] dla kompatybilności wstecznej.

Klucz deszyfrujący jest znany tylko osobie pobierającej dane. Dzięki temu zawartość jest chroniona przed pozostałymi użytkownikami, którzy je przechowują.

Listing 3.2: Format klucza *CHK*

```
1 CHK@file hash,decryption key,crypto settings
```

Listing 3.3: Przykładowy klucz *CHK*

```
1 CHK@SVbD9~HM[...]PJZLL5NX5Brs,bA7qLNJR7IXR[...]K~18kSi6bbNQ,AAEA--8
```

***Signed Subspace Key* (*SSK*)** – Pozwala przechowywać dane, które można aktualizować. Dla pliku generowany jest klucz asymetryczny *DSA* o parametrach $(L, N) == (2048, 256)$. Skrót *SHA-256* klucza publicznego jest używany do identyfikacji i pozwala pobrać dane. Dane są podpisane kluczem prywatnym. Dodatkowo podobnie jak w *CHK*, dane są szyfrowane indywidualnym 256 bitowym kluczem *RSA* w trybie *PCFB* z kodem *HMAC-SHA256*.

Klucz pozwalający pobrać dane składa się ze skrótu klucza publicznego identyfikującego dane, klucza deszyfrującego, informacji o tym, jakie algorytmy kryptograficzne zostały wykorzystane, nazwy nadanej przez autora i numeru wersji. Można, używając jednego klucza, opublikować kilka plików o różnych nazwach.

Listing 3.4: Format klucza *SSK*

```
1 SSK@public key hash,decryption key,crypto settings/user selected name-version
```

Listing 3.5: Przykładowy klucz *SSK*

```
1 SSK@GB3wuHmt[...]o-eHK35w,c63EzO7u[...]3YDduXDs,AQABAAE/mysite-4
```

***Updateable Subspace Key* (*USK*)** – Jest tym samym co *SSK*, z tą różnicą, że klient automatycznie próbuje znaleźć i pobrać najnowszą wersję danych, zaczynając od podanego numeru wersji.

Przy pobieraniu danych, numer wersji może być podany jako liczba dodania albo ujemna. Jeżeli numer wersji jest podany jako liczba dodatnia, to oprogramowanie zwróci ostatnio pobrane dane, a następnie w tle zacznie sprawdzać, czy istnieje nowsza wersja. Jeżeli nowsza wersja zostanie znaleziona, to zostanie zwrócona użytkownikowi przy następnej próbie pobrania danych. Jeżeli numer wersji jest podany jako liczba ujemna, to oprogramowanie od razu sprawdzi obecność nowszej wersji, i, jeżeli taka istnieje, to pobierze i zwróci użytkownikowi nowsze dane.

Listing 3.6: Format klucza *USK*

```
1 USK@public key hash,decryption key,crypto settings/user selected name/number/
```

Listing 3.7: Przykładowy klucz *USK*

```
1 USK@GB3wuHmt[...]o-eHK35w,c63EzO7u[...]3YDduXDs,AQABAAE/mysite/5/
```

***Keyword Signed Key* (*KSK*)** – Dane są identyfikowane wybraną nazwą i nie są chronione. Każdy może je nadpisać.

Listing 3.8: Przykładowy klucz *KSK*

```
1 KSK@myfile.txt
```

Web of Trust

Web of Trust [56] jest podsystemem odpowiedzialnym za określanie zaufania pomiędzy użytkownikami w sieci. Użytkownicy w *WoT* identyfikowani są przez klucze *USK*, które jednocześnie służą do publikowania i pobierania informacji o użytkownikach. *WoT* pozwala publikować dodatkowe informacje w postaci klucz-wartość.

Użytkownicy mogą budować zaufanie przez rozwiązywanie *CAPTCHA*. Alicja w swojej informacji *Wot* publikuje *CAPTCHA* do rozwiązania przez innych użytkowników. Bob rozwiązuje *CAPTCHA*. Na podstawie rozwiązania tworzy klucz *KSK*, pod którym zapisuje swoje dane. Alicja regularnie sprawdza, czy ktoś rozwiązał jej *CAPTCHA*, próbując pobrać dane z kluczy *KSK*. Alicja pobiera dane Boba i dodaje go do swojej listy zaufanych użytkowników. Oprócz tego można też ręcznie dodać innego użytkownika do listy zaufanych.

Freemail

Freemail [22] jest systemem pocztowym opartym o *Freenet* i podsystem *Web of Trust*. Użytkownicy *Freemail* identyfikowani są kluczami *WoT*.

We *Freemail* wysyłanie wiadomości polega na umieszczaniu danych we *Freenet* a odbieranie na aktywnym sprawdzaniu obecności nowych danych w sieci.

Stworzenie skrzynki pocztowej polega na opublikowaniu pliku *mailsite* pod tym samym kluczem *USK* co dane *WoT*, ale z inną nazwą. Plik *mailsite* zawiera losowo wygenerowany, 4096 bitowy klucz publiczny *RSA* i losowo wygenerowaną wartość *rtsksk* składającą się z 32 małych liter (znaki z zakresu 97-122 w *ASCII*).

Gdy Alicja chce rozpocząć rozmowę z Bobem, tworzy specjalną wiadomość *Request To Send* (*RTS*), która służy do stworzenia kanału komunikacyjnego (*channel*) i zawiera:

- klucz *USK* do pliku *mailsite* nadawcy,
- adres odbiorcy,
- losowo wygenerowany klucz prywatny *SSK* kanału,
- losowe wartości *fetchslot* i *sendslot* (opis poniżej).
- termin ważności.

Wiadomość *RTS* jest podpisana kluczem Alicji i zaszyfrowana kluczem Boba z pliku *mailsite*. Następnie jest zapisana pod kluczem *KSK* stworzonym na podstawie wartości *rtsksk* opublikowanej przez Boba, daty, i liczby naturalnej (program zaczyna od zera i szuka pierwszego niewykorzystanego klucza *KSK*).

W ten sposób Alicja i Bob uzyskują wspólny klucz *SSK*, pod którym Alicja i Bob umieszczają kolejne wiadomości. Jako nazwy pliku Alicja używa „i" (od *initiator*), a Bob „r" (od *recipient*). Jako pierwszego numeru wersji Alicja używa wartości *fetchslot*, a Bob *sendslot*. Kolejne wiadomości są zapisywane pod numerami wersji uzyskanymi z wartości *SHA-256* poprzedniego numeru wersji. W ten sposób uzyskiwane jest częściowe *forward secrecy* (częściowe, ponieważ nie chroni przed atakującym, który zapisał wysłane w przeszłości wiadomości i zdobył klucz później).

3.5 Aplikacja I2P-Bote

I2P-Bote [32] jest systemem pocztowym działającym wewnątrz sieci *Invisible Internet Project* (*I2P*) [82].

Sieć *I2P*, podobnie jak sieć *TOR*, jest siecią rozproszoną, w której, dla zachowania anonimowości użytkowników, połączenia są tunelowane przez wielu pośredników i szyfrowane wielowarstwowo. W przeciwieństwie do *TOR*, *I2P* nie ma na celu umożliwiania anonimowego dostępu do Internetu, choć udostępnia i taką możliwość. Zamiast tego skupia się na anonimowej komunikacji pomiędzy użytkownikami.

W *I2P-Bote* adresem jest para kluczy publicznych: jeden do szyfrowania danych, drugi do weryfikacji podpisów. Tworząc adres, trzeba wybrać jedną z czterech dostępnych par algorytmów: *ElGamal-2048/DSA-1024*, *ECDH-256/ECDSA-256* (domyślnie sugerowane), *ECDH-521/ECDSA-521* i *NTRUEncrypt-1087/GMSS-512*.

Wysyłane wiadomości są dzielone na fragmenty o rozmiarze 30kb, szyfrowane kluczem odbiorcy i zapisywane w *Kademlia DHT* u pozostałych użytkowników sieci z zachowaniem redundancji. Następnie klucze *DHT* fragmentów są umieszczane w *Kademlia DHT* w jednym pliku nazywanym indeksem, pod kluczem równym sumie kontrolnej *SHA-256* adresu odbiorcy. *Kademlia DHT* pozwala zapisać wiele wartości pod jednym kluczem, więc nie ma problemu, gdy wysłanych jest wiele wiadomości.

Dane są przechowywane przez co najmniej 100 dni. Każda wartość przechowywana w *Kademlia DHT* zawiera skrót *SHA-256* wiadomości autoryzującej usunięcie danych, która jest tworzona przez nadawcę i zaszyfrowana przekazywana odbiorcy. Dzięki temu odbiorca po odebraniu wiadomości może usunąć dane z *Kademlia DHT*.

Nadawca domyślnie nie komunikuje się bezpośrednio z użytkownikami przechowującymi dane. Zamiast tego komunikacja odbywa się przez wielu pośredników w celu zachowania anonimowości użytkowników. Liczbę pośredników można wybrać z zakresu od 0 do 3 (domyślnie 2). Komunikacja odbiorcy przez pośredników jeszcze nie została zaimplementowana. Dane przesyłane pomiędzy użytkownikami są zaszyfrowane losowo generowanymi kluczami *AES-256*, które są dołączane do wiadomości i zaszyfrowane kluczem asymetrycznym. Pakiety są szyfrowane wielowarstwowo, to znaczy pośrednik po odebraniu i odszyfrowaniu wiadomości uzyskuje wiadomość zaszyfrowaną i zaadresowaną do następnego odbiorcy.

3.6 Aplikacja LemonMail

LemonMail [47] jest systemem pocztowym wykorzystującym *blockchain Ethereum* do przechowywania metadanych o wiadomościach oraz rozproszonego systemu plików *IPFS* [79] do przechowywania treści wiadomości.

W *IPFS* dane są identyfikowane przez skrót kryptograficzny zawartości. Dane w sieci rozsyła się podobnie jak w *BitTorrent*: węzeł w sieci, który posiada kopię danych, udostępnia je innym zainteresowanym.

W *LemonMail* użytkownicy identyfikowani są przez adres *Ethereum*, czyli przez klucz publiczny. Istnieje możliwość rejestracji nazw w kontrakcie *Ethereum*.

Dane przechowywane w IPFS

Wysłanie wiadomości zaczyna się od umieszczenia w *IPFS* pliku z obiektem *json* o strukturze przedstawionej w listingu 3.9.

Listing 3.9: Format danych zapisanych w *IPFS*

```
{
    "toAddress": adres ethereum odbiorcy,
    "encryptedReceiverData": emailData (opis ponizej) zaszyfrowany kluczem
        ethereum odbiorcy,
    "encryptedSenderData": emailData zaszyfrowany kluczem nadawcy,
    "attachments": [
        {
            "fileName": nazwa pliku,
            "fileData": zawartosc
        } zaszyfrowane kluczem odbiorcy,
        ...
    ],
    "inReplyTo": opcjonalny identyfikator wiadomosci (opis ponizej), do ktorej ta
        jest odpowiedzia
}
```

emailData to obiekt *json* o strukturze przedstawionej w listingu 3.10.

Listing 3.10: Format obiektu *emailData*

```
{
    "from": adres ethereum nadawcy,
    "to": adres ethereum odbiorcy,
    "subject": temat wiadomosci,
    "body": tresc wiadomosci,
    "time": znacznik czasu
}
```

Identyfikator wiadomości to wartość skrótu *SHA-3* identyfikatora danych z *IPFS*.

Metadane przechowywane w *Ethereum*

Wysłanie wiadomości polega na umieszczeniu w kontrakcie na *Ethereum* zdarzenia z następującymi danymi:

emailId	identyfikator wiadomości,
from	adres nadawcy,
to	adres odbiorcy,
ipfsHash	identyfikator danych w *IPFS*,
inReplyToId	opcjonalny identyfikator wiadomości, do której ta jest odpowiedzią,
inReplyToIpfsHash	opcjonalny identyfikator danych w *IPFS* wiadomości, do której ta jest odpowiedzią.

Podsumowanie

LemonMail jako jedyny w tym zestawieniu używa technologii *blockchain* do rozwiązania problemu komunikacji asynchronicznej. Dodatkowo problem kojarzenia czytelnej nazwy z kluczem też został rozwiązany: podstawą jest klucz, dla którego można zarejestrować unikalną nazwę w globalnym rejestrze.

Jednak moim zdaniem ten projekt ma sporo wad:

- Umieszcza w *Ethereum* więcej informacji, niż to jest potrzebne, co może stanowić zagrożenie dla prywatności komunikacji.
- Do pełnej asynchroniczności potrzebna jest dodatkowa pomoc w przechowywaniu danych w *IPFS*. Autorzy są tego świadomi i nawet uruchomili specjalny węzeł, który to robi[2]. Rozwiązanie dla tego problemu ma dostarczyć projekt *FileCoin* (`https://filecoin.io/`).
- Wiadomość jest przechowywana w jednym pliku zaszyfrowana kluczem nadawcy i odbiorcy. Żeby wiadomość zniknęła z *IPFS*, muszą ją skasować obydwie osoby. Możliwy jest następujący scenariusz: Alicja i Bob wymieniają się wiadomościami. Alicja usuwa wiadomości ze swojej skrzynki. Mallory atakuje komputer Alicji i zdobywa jej klucz *Ethereum*. Mallory może od Boba pobrać i odczytać wiadomości wysłane przez Alicję.

[2] `https://www.reddit.com/r/ethereum/comments/68tyhn/lemonmail_dapp_a_decentralized_email_service/dh1mvyz`

3.7 Podsumowanie

Lista wymienionych powyżej narzędzi nie jest wyczerpująca. W trakcie pisania tej pracy znalazłem jeszcze inne, na przykład *ZeroMail*[3] i *MailChain*[4]. Po pobieżnym zapoznaniu się z ich architekturą stwierdziłem, że nie wniosłyby do mojej pracy niczego istotnie nowego.

Tabela 3.2 przedstawia zestawienie różnych cech przedstawionych wcześniej systemów pocztowych.

wartości pożądane	*Email*	*DIME*	*BitMessage*	*RetroShare*	*Freemail*	*I2P-Bote*	*LemonMail*
sieć rozproszona	○	○	●	●	●	●	●
email spoofing	○	◐[1]	○	○	○	○	○
komunikacja asynchroniczna	●	●	●	●	●	●	◐[2]
gwarancja przechowania danych	●	●	●	○	◐[3]	○	○	◐[4]
forward secrecy	●	○	○	○	○	◐[5]	○	○
inny użytkownik widzi, że wiadomość została wysłana i przez kogo	○	○	○	●[6]	◐[7]	○	○	●
inny użytkownik widzi, do kogo została wysłana wiadomość	○	○	○	○	◐[8]	◐[9]	○	●

Tabela 3.2: Porównanie istniejących systemów pocztowych

●– cecha obecna, ◐– częściowo, ○– brak.

1. Opisane w rozdziale 2.4 metody przeciwdziałania *email spoofing* nie muszą być wdrożone i przestrzegane.
2. W *IPFS* dane muszą zostać pobrane i udostępniane przez inny węzeł w sieci, żeby były dostępne po odłączeniu się nadawcy.
3. Pośrednicy teoretycznie powinni być zaufanymi znajomymi, którym zależy na jakości i bezpieczeństwie komunikacji innych użytkowników sieci.
4. W ramach projektu *IPFS* powstaje narzędzie *Filecoin*, które w przyszłości pozwoli na płacenie za przechowywanie i udostępnianie danych.

[3] https://github.com/HelloZeroNet/ZeroMail
[4] https://mailchain.xyz/

5. Wykorzystanie funkcji skrótu *SHA-256* do wyliczania nowego numeru wersji pliku z wiadomością nie pozwala atakującemu na znalezienie starych wiadomości. Jednak cały czas jest używany ten sam klucz szyfrujący, więc jest możliwe odczytanie pobranych wcześniej wiadomości.

6. Jest to trochę ograniczone przez podział sieci na *strumienie*.

7. Użytkownik może zobaczyć wiadomości przekazywane pomiędzy zaufanymi znajomymi, jeżeli zostanie wybrany na pośrednika, ale nie dowie się, kto jest nadawcą. Obecnie łatwo jest znaleźć się w sieci „znajomych" z obcymi ludźmi, ze względu na publikowanie przez użytkowników kluczy na ogólnodostępnych serwerach.

8. Wiadomość zawiera jawnie adres odbiorcy, ale przekazywana tylko przez zaufanych znajomych.

9. Przy tworzeniu kanału komunikacyjnego, wiadomości *RTS* są umieszczane w przewidywalnym miejscu.

Jak widać, zbadane przeze mnie rozwiązania różnią się od siebie i każde ma swoje mocne i słabe strony. Dzięki temu porównaniu wpadłem na pomysł nowego systemu pocztowego, opisanego w następnym rozdziale, który wykorzysta niektóre z mocnych stron omówionych projektów.

Rozdział 4

Aplikacja TigerMail

TigerMail to zaprojektowany przeze mnie system pocztowy.

W tym rozdziale najpierw określę wymagania, jakie *TigerMail* ma spełniać. Następnie przedstawię wykorzystywane narzędzia i algorytmy, architekturę aplikacji, model zagrożeń i analizę bezpieczeństwa. Dalej znajdzie się opis implementacji, a na koniec opis przeprowadzonych testów.

4.1 Specyfikacja wymagań

Chcę, żeby zaprojektowany przeze mnie system spełniał następujące wymagania:

Obsługuje asynchroniczną wymianę wiadomości. Dzięki temu nadawca i odbiorca mogą się niezależnie, w różnych momentach podłączać i odłączać od sieci tak, jak w obecnie używanej poczcie elektronicznej.

Korzysta z sieci rozproszonej. W sieci zdecentralizowanej serwery działają w imieniu ich użytkowników. To oznacza, że użytkownik musi mieć zaufanie do właściciela serwera, że dba o bezpieczeństwo przechowywanych danych i że przetwarza je zgodnie z opublikowaną polityką prywatności. Niestety niewiele osób czyta politykę prywatności i przejmuje się jej zapisami, przez co użytkownicy popularnych, darmowych skrzynek pocztowych płacą swoją prywatnością za dostarczane usługi [58].

Sieci rozproszone składają się tylko z użytkowników. Nie mają wyróżnionych elementów, którym trzeba ufać bardziej niż innym. Co więcej, są projektowane z założeniem, że nikomu nie można ufać i w związku z tym, muszą mieć odpowiednie zabezpieczenia.

Identyfikuje użytkowników przez klucz kryptograficzny. Podstawowym problemem w korzystaniu z *OpenPGP* jest upewnienie się, że właściciel klucza jest jednocześnie właścicielem konta pocztowego. W sytuacji, gdy adres i klucz są tym samym, problem znika.

Nowym problemem jest to, że nikt nie będzie w stanie zapamiętać i powtórzyć swojego adresu. Rozwiązanie tego problemu jest poza zakresem tej pracy, ale jest krótko opisane w rozdziałach 3.6 i 5.

Ujawnia innym użytkownikom minimum o tym, kto, komu, kiedy i co wysłał. W sieci rozproszonej urządzenia mogą pośredniczyć w komunikacji pomiędzy innymi, więc jakieś dane muszą być ujawnione. Ten punkt zwraca uwagę na to, żeby w miarę możliwości maksymalnie ograniczyć ilość ujawnianych informacji.

Szyfruje wiadomości algorytmem zapewniającym *forward secrecy*. Jeżeli ktoś o złych zamiarach zapisał komunikację atakowanej osoby, a potem zdobył jej klucz prywatny, to dzięki *forward secrecy* nie odszyfruje starych wiadomości.

4.2 Narzędzia i algorytmy

Następujące narzędzia i algorytmy będą wykorzystane przy budowie *TigerMail*:

4.2.1 Blockchain Ethereum

Blockchain Ethereum [19, 75], działa w sieci publicznej i rozproszonej. Jest wyjątkowy wśród *blockchainów*, ponieważ nie został stworzony tylko i wyłącznie jako platforma do przekazywania płatności, ale pozwala na programowanie własnych kontraktów i przechowywanie dowolnych danych.

Zapisywanie transakcji w blokach

Blockchain zapisuje wszystkie wykonane transakcje w (jak nazwa wskazuje) łańcuchu bloków. Jeden blok zawiera wiele transakcji oraz sumę kontrolną poprzedniego bloku. Z tego wynikają następujące właściwości *blockchaina*:

- Operacje mają kolejność chronologiczną.
- Nie można zmienić operacji zapisanych w starych blokach bez modyfikowania następnych bloków.
- Dane zapisane w *blockchainie* zostają w nim na zawsze.

Tworzenie nowych bloków nazywane jest „wykopywaniem", a użytkownicy, którzy się tym zajmują „górnikami". Pozostali użytkownicy ogłaszają swoje operacje, które są umieszczane w blokach przez „górników". Łańcuch bloków może się rozgałęziać. Żeby temu przeciwdziałać, wykorzystany jest algorytm *proof-of-work*. Powoduje on, że stworzenie nowego bloku wymaga dużej ilości mocy obliczeniowej i zajmuje określoną w przybliżeniu ilość czasu. Najdłuższy łańcuch jest uznawany za obowiązujący.

Każda transakcja, niezależnie czy jest to zwykły przelew, czy wywołanie metody kontraktu, wiąże się z wykonaniem jakiegoś programu w *Ethereum Virtual Machine* (*EVM*). Każda wykonywana instrukcja (na przykład dodanie dwóch zmiennych, zapisanie wartości do stanu kontraktu, wywołanie innej funkcji) ma określony koszt w jednostce nazywanej *Gas*. Każda transakcja zawiera informacje o limicie *Gasu* (dzięki temu programy nie mogą się wykonywać w pętli bez końca), wybraną cenę *Gasu* w walucie *Ether* oraz fundusz na pokrycie kosztu wykonania programu.

„Górnik" wykonuje program w *EVM*, pobiera opłatę zgodnie z ilością wykorzystanego *Gasu* i jego ceny, a resztę zwraca zlecającemu. „Górnik" może dowolnie wybierać transakcje do umieszczenia w bloku, dlatego transakcje z wysoką ceną za jednostkę *Gasu* są „wykopywane" szybciej od pozostałych.

Zapisywanie danych w kontrakcie

Istnieją dwa miejsca, w których można zapisywać dane w kontrakcie:

- Zmienne stanu, które można odczytywać i modyfikować w metodach kontraktu.
- Zdarzenia (*events*), które tworzy się w kontrakcie, ale odczytuje tylko w aplikacji przez *Remote Procedure Call* (*RPC*). Tworzenie zdarzeń kosztuje dużo mniej *Gasu* od zapisywania danych w zmiennych stanu. Jedno zdarzenie może zawierać wiele parametrów. Parametry mogą być indeksowane (maksymalnie 3), dzięki czemu można wyszukiwać zdarzenia o konkretnych parametrach.

Listing 4.1: Przykładowy kontrakt *Ethereum*

```
contract ValueStore {
    mapping(address => uint) values;

    event ValueStored (
        address indexed from,
        uint value
    );

    function storeValue(uint value) public {
        values[msg.sender] = value;
        emit ValueStored(msg.sender, value);
    }
```

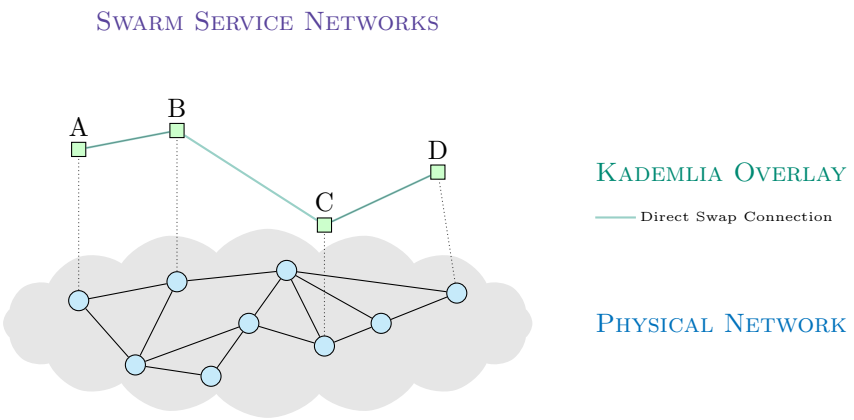
```
13
14      function getValue(address from) public returns (uint) {
15          return values[from];
16      }
17  }
```

W przykładzie w listingu 4.1 `values` jest zmienną stanu, natomiast `ValueStored` jest zdarzeniem z dwoma parametrami. Metodą `getValue` można poznać tylko aktualną wartość zapisaną z konkretnego adresu. Przeglądając zdarzenia, można prześledzić całą historię zmian dla wszystkich adresów, lub ograniczyć wyszukiwanie do jednego adresu.

4.2.2 Rozproszony system plików Swarm

Swarm [78] jest rozproszonym systemem plików. Dane przechowywane w nim są identyfikowane przez ich sumę kontrolną (*Keccak 256*), nazywaną *Swarm Hash*. Dzięki temu pobierając dane, od razu można się upewnić, że są poprawne. Użytkownicy w sieci *Swarm* są identyfikowani sumą kontrolną (*Keccak 256*) klucza publicznego *Ethereum*.

Dane umieszczane w *Swarm* są dzielone na fragmenty (*chunks*) o wielkości 4096 bajtów. *Swarm* wykorzystuje rozproszoną tablicę mieszającą *Kademlia DHT* do przechowywania fragmentów danych. *Kademlia DHT* na podstawie sum kontrolnych danych i kluczy użytkowników w sposób deterministyczny przyporządkowuje dane do użytkowników w sieci i zapewnia redundancję. Dzięki temu *Swarm* pozwala wysłać do niego dane i odłączyć się od sieci. Oprócz tego *Kademlia DHT* jest wykorzystywana do organizacji połączeń w sieci i przekazywania danych (rysunek 4.1).



Rysunek 4.1: *Swarm* używa *Kademlia DHT* do organizacji dodatkowej warstwy połączeń sieciowych.
Źródło: *Viktor Trón, Aron Fischer: Generalised swap swear and swindle games*
https://www.overleaf.com/project/59c298da5b3a6e3b7ec80d56

Istnieje również możliwość stworzenia identyfikatora do zmieniających się danych, nazywanego *Swarm Feed*. Taki identyfikator uzyskuje się przez umieszczenie w *Swarm* specjalnego obiektu zawierającego „temat" (czyli dowolne 32 bajty) i adres *Ethereum* użytkownika. Umieszczenie danych w *Swarm Feed* wymaga podpisu kluczem *Ethereum*. Dzięki temu wielu użytkowników może publikować dane na ten sam „temat".

Obecnie *Swarm* nie gwarantuje, jak długo dane będą przechowywane, ani kiedy będą usunięte. Każdy klient sieci zajmujący się przechowywaniem danych utrzymuje kolejkę fragmentów. Fragmenty są przesuwane na początek kolejki, gdy są dodawane albo przez kogoś pobierane. W momencie, gdy udostępniana przestrzeń dyskowa jest zapełniona i nowy fragment jest dodawany, ostatni element z kolejki jest usuwany. Podczas testowania aplikacji *TigerMail* zauważyłem, że dane są usuwane po kilku tygodniach.

Bardzo podobnym narzędziem jest *IPFS* [79], jednak nie wykorzystałem go, ze względu na to, iż aby udostępniać w nim dane, trzeba być podłączonym do sieci. Nie ma możliwości automatycznego wysłania danych do sieci, tak żeby były dostępne po odłączeniu.

Obydwa projekty *Swarm* i *IPFS* niezależnie pracują nad mechanizmami pozwalającymi na opłacanie przechowywania i udostępniania danych przez innych użytkowników oraz weryfikację zobowiązań.

4.2.3 Protokół Signal

Signal jest protokołem zapewniającym szyfrowanie z *forward secrecy*. Jest wykorzystywany przez komunikatory takie jak *Signal*, *WhatsApp*, *Wire* i *Facebook Messenger*. Bezpieczeństwo tego protokołu zostało potwierdzone w niezależnym audycie [15]. Składa się z kilku algorytmów:

Algorytm XEdDSA

Algorytm *XEdDSA* [57] wykorzystuje krzywe eliptyczne do podpisywania danych. Dla konkretnych krzywych jest nazywany *XEd25519* lub *XEd448*. Jest zaprojektowany tak, żeby móc używać tych samych kluczy, które są używane w algorytmach *X25519* i *X448* [45].

Algorytmy *X25519* i *X448* są używane w funkcji *Elliptic Curve Diffie-Hellman* (*ECDH* lub krócej *DH*) [60]. Dla dwóch par kluczy krzywych eliptycznych (d_A, Q_A) i (d_B, Q_B), gdzie d to klucz prywatny a Q publiczny, *ECDH* pozwala wyliczyć wspólny sekret wykorzystując parę (d_A, Q_B) lub (d_B, Q_A). Dzięki wykorzystaniu tych algorytmów możliwe jest użycie tych samych kluczy do podpisywania danych i wyliczania wspólnego sekretu do szyfrowania danych.

Protokół Extended Triple Diffie-Hellman (X3DH)

Protokół *X3DH* [52] rozwiązuje problem zapewnienia *forward secrecy* przy rozpoczęciu konwersacji. Polega on na opublikowaniu zestawu kluczy publicznych:

- Klucz tożsamości (*identity key*) IK.
- Klucz wstępny, podpisany kluczem tożsamości (*signed prekey*) SPK.
- Zero lub więcej kluczy jednorazowych (*prekey*) $OPK^1, OPK^2, ...$

Użytkownik co jakiś czas publikuje nowe klucze SPK i OPK i usuwa stare (usuwanie starych kluczy może być nieznacznie opóźnione na wypadek odebrania wiadomości zaszyfrowanej starym kluczem).

Protokół *X3DH* wykorzystuje funkcje:

- *Elliptic Curve Diffie-Hellman* (opisana w rozdziale powyżej).
- *HMAC-based Extract-and-Expand Key Derivation Function* (*HKDF* lub krócej *KDF*) [41], która na podstawie klucza i opcjonalnej *soli* wylicza dowolną ilość nowych danych pseudolosowych, które mogą być użyte jako klucze kryptograficzne. Operacja jest deterministyczna i nieodwracalna.

Pierwsza wiadomość jest zaszyfrowana kluczem symetrycznym SK wyliczonym na podstawie kluczy:

- Prywatnego klucza tożsamości nadawcy IK_A,
- Publicznego klucz tożsamości odbiorcy IK_B,
- Publicznego podpisanego klucza wstępnego odbiorcy SPK_B,
- (opcjonalnie) publicznego klucza jednorazowego odbiorcy OPK_B,
- wylosowanego klucza tymczasowego nadawcy (*ephemeral key*) EK_A.

Klucz symetryczny SK jest wyliczany w następujący sposób (jeżeli klucz SPK_B nie jest używany, to wartość *DH4* nie jest obliczana i używana):

```
DH1 = DH(IK_A, SPK_B)
DH2 = DH(EK_A, IK_B)
DH3 = DH(EK_A, SPK_B)
DH4 = DH(EK_A, OPK_B)
SK = KDF(DH1 || DH2 || DH3 || DH4)
```

Wysłana wiadomość składa się z:

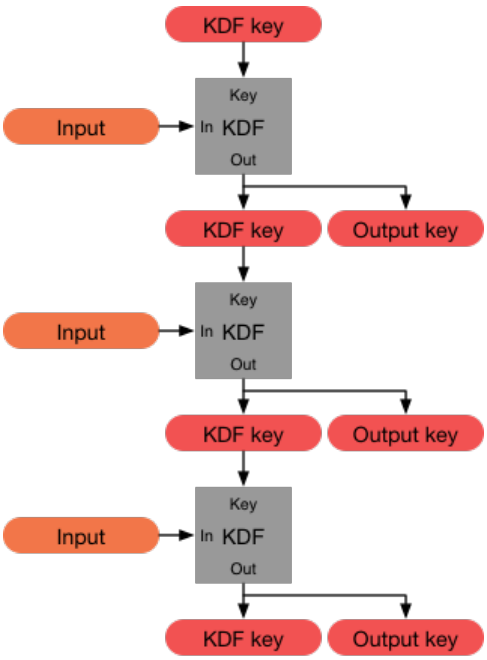
- Klucza publicznego tożsamości nadawcy IK_A,
- publicznego klucza tymczasowego nadawcy EK_A,

- identyfikatorów użytych kluczy wstępnych,
- treści wiadomości zaszyfrowanej algorytmem dostarczającym *authenticated encryption with associated data (AEAD)*, kluczem SK, z kluczami $IK_A||IK_B$ użytymi jako *associated data*.

Protokół Double Ratchet

Protokół *Double Ratchet* [50] zapewnia *forward secrecy* w wymianie wiadomości. Składa się z dwóch mechanizmów: *Symmetric-key ratchet* i *Diffie-Hellman ratchet*.

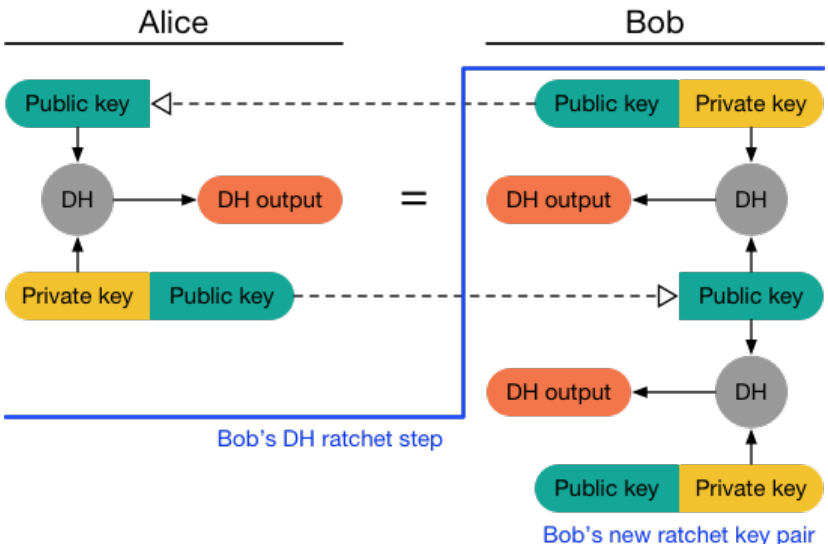
Symmetric-key ratchet jest odpowiedzialny za generowanie kluczy dla kolejnych wiadomości, gdy druga strona nie przysyła odpowiedzi. Wykorzystuje funkcję *KDF* w sposób łańcuchowy, czyli oprócz klucza wiadomości generowany jest klucz wejściowy do następnego kroku (rysunek 4.2).



Rysunek 4.2: Łańcuch *KDF*
Źródło: *[50]*

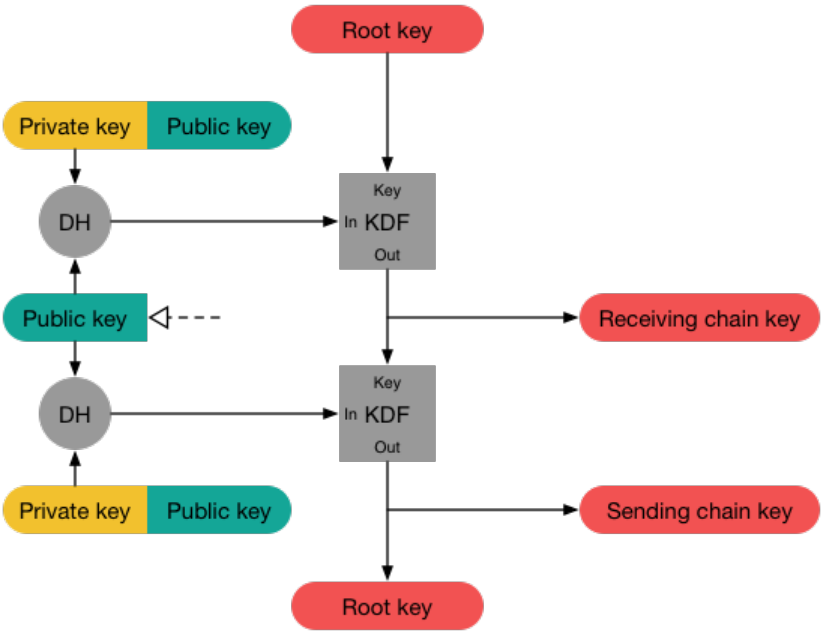
Jeden użytkownik do rozmowy z drugim używa trzech łańcuchów kluczy: główny (nazywany *root*), wysyłający i odbierający. Łańcuch główny jest ten sam dla obu rozmówców, natomiast wysyłający posiadany przez jednego jest równy kluczowi odbierającemu drugiego.

Diffie-Hellman ratchet jest odpowiedzialny za tworzenie identycznych kluczy u Alicji i Boba, wymagając przesyłania tylko kluczy publicznych wygenerowanych kluczy asymetrycznych. W każdym kroku Alicja i Bob generują nową parę kluczy asymetrycznych i wysyłają klucz publiczny drugiej stronie. Używając funkcji *ECDH*, uzyskują identyczne klucze (rysunek 4.3).



Rysunek 4.3: Krok w wymianie kluczy *Diffie-Hellman ratchet*.
Źródło: *[50]*

Klucze uzyskane w powyższym mechanizmie są wykorzystywane do tworzenia nowych kluczy łańcuchów wysyłających i odbierających przez podanie ich jako *sól* do łańcucha głównego (rysunek 4.4).



Rysunek 4.4: Generowanie nowych kluczy łańcuchów wysyłających i odbierających z wykorzystaniem *Diffie-Hellman ratchet*.
Źródło: *[50]*

Algorytm Sesame

Algorytm *Sesame* [51] jest odpowiedzialny za zarządzanie sesjami rozmów. Rozwiązuje następujące problemy:

- Alicja i Bob mogą mieć więcej niż jedno urządzenie. Gdy Alicja wysyła wiadomość do Boba, *Sesame* zajmuje się wysyłaniem wiadomości do wszystkich urządzeń, zarówno Boba, jak i pozostałych urządzeń Alicji.
- Alicja i Bob mogą dodawać i usuwać urządzenia. *Sesame* zajmuje się tworzeniem nowych sesji i usuwaniem starych.
- Alicja i Bob mogą jednocześnie rozpocząć komunikację ze sobą. *Sesame* przeciwdziała powstaniu dwóch niezależnych sesji.
- Alicja i Bob mogą skasować swoje zapisane sesje lub przywrócić kopię zapasową wykonaną w przeszłości.

4.3 Architektura aplikacji

W tym rozdziale opisany jest algorytm wymiany wiadomości, schemat komunikacji pomiędzy procesami oraz uzasadnienie, jak aplikacja odpowiada na zagrożenia i spełnia wymienione powyżej wymagania.

Komunikacja pomiędzy użytkownikami

W sieci rozproszonej, jeżeli Alicja chce wysłać do Boba wiadomość, gdy Bob jest niedostępny, a Alicja też może się w każdej chwili rozłączyć, ktoś trzeci musi tę wiadomość przez jakiś czas przechować. Moim pomysłem na rozwiązanie tego problemu jest zapisanie jej u wszystkich użytkowników na zawsze. Architektura *blockchain* pasuje do tej roli idealnie.

Jednak zapisanie wiadomości w *blockchainie* jest kosztowne (im więcej danych zawiera transakcja, tym więcej trzeba za nią zapłacić) i niebezpieczne (bo zostaje zapisana na zawsze). Dlatego zapisywany będzie tylko *Swarm Hash* jako zdarzenie. Będzie zaszyfrowany, żeby pobrać dane mógł tylko odbiorca.

Do szyfrowania wiadomości użyty będzie protokół *Signal*. Klucze publiczne (tożsamości i wstępne) będą publikowane jako *Swarm Feed* pod określonym tematem.

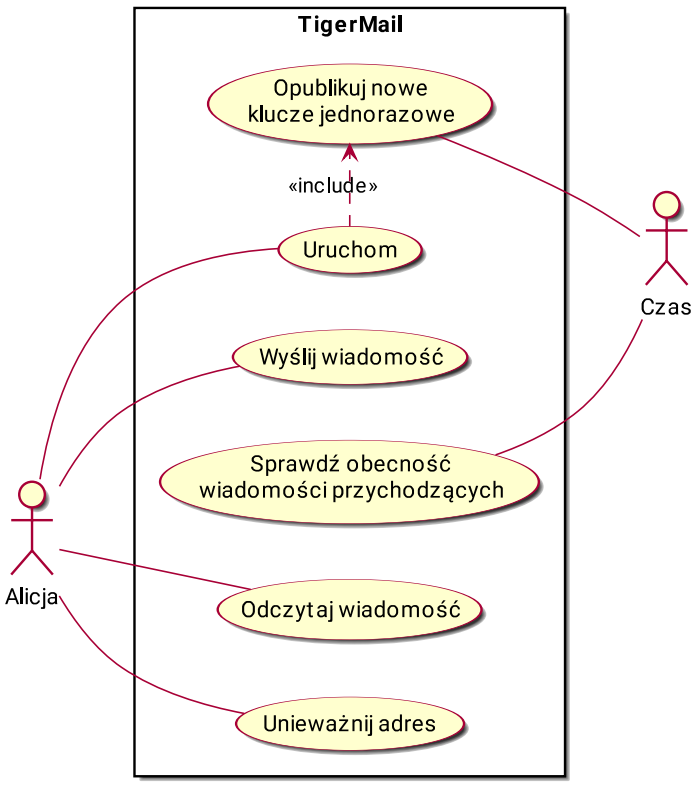
Odnośnik zapisany w *Ethereum* będzie zaszyfrowany innym kluczem, w zależności od sytuacji:

- Pierwsza wiadomość: podpisany klucz wstępny odbiorcy SPK.
- Wiadomość w sesji: klucz, którym była zaszyfrowana ostatnia wiadomość.

Każdy użytkownik będzie obserwował *blockchain* i próbował odszyfrować każdą informację o wysłanej wiadomości. Zastosowanie szyfrowania z kodem uwierzytelniania $HMAC$ pozwoli ustalić, czy użyty klucz był właściwy.

W razie wykrycia kompromitacji klucza będzie możliwe umieszczenie w *Ethereum* specjalnej informacji unieważniającej adres. Taka informacja będzie publiczna i zapisana na stałe, więc na pewno dotrze do wszystkich zainteresowanych.

Rysunek 4.5 przedstawia diagram przypadków użycia aplikacji *TigerMail*.



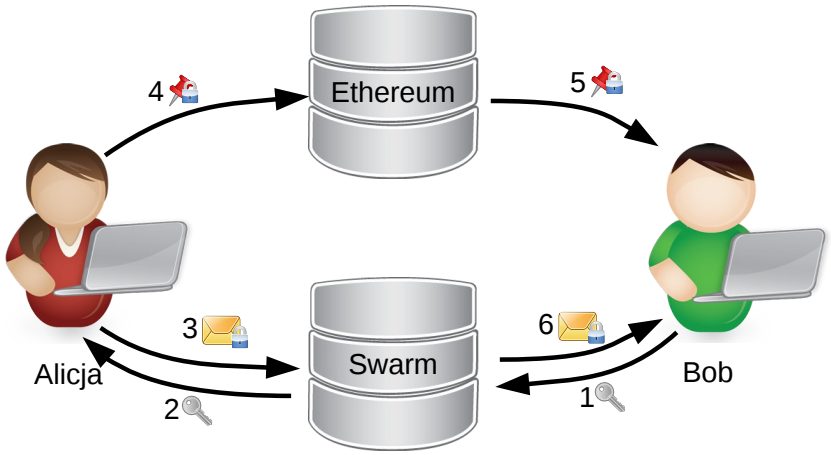
Rysunek 4.5: Przypadki użycia TigerMail

Sekwencja operacji przy przesłaniu wiadomości

Poniżej znajduje się sekwencja operacji przy przesłaniu wiadomości (rysunek 4.6). Dwa pierwsze punkty są wykonywane tylko przed przesłaniem pierwszej wiadomości.

1. Bob publikuje swoje klucze jednorazowe.
2. Alicja pobiera klucze Boba.
3. Alicja szyfruje wiadomość kluczem jednorazowym Boba i umieszcza w *Swarm*.
4. Alicja zapisuje zaszyfrowany *Swarm Hash* w *Ethereum* jako zdarzenie.
5. Bob pobiera zdarzenia z *Ethereum* i próbuje odszyfrować wszystkie *Swarm Hash*. Udaje mu się z tym wysłanym przez Alicję.
6. Bob pobiera wiadomość z *Swarm*.

Dalsza komunikacja odbywa się podobnie jak powyżej w punktach od 3 do 6, z tą różnicą, że wiadomości nie są szyfrowane kluczami jednorazowymi pobranymi ze *Swarm*, tylko ustalanymi przez protokół *Double Ratchet* (opis w rozdziale 4.2.3).

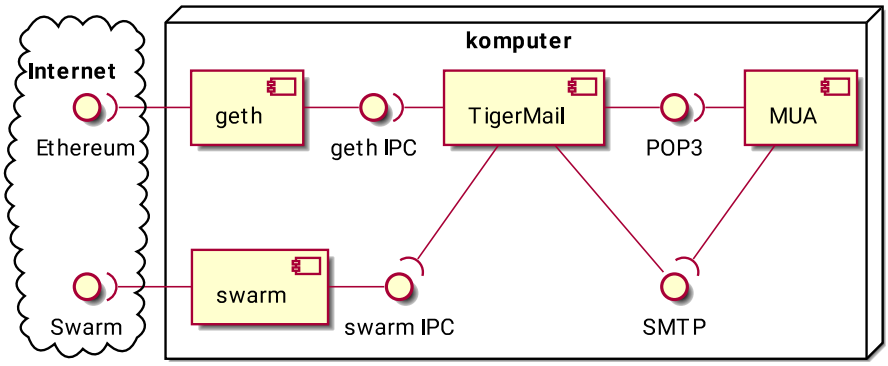


Rysunek 4.6: Sekwencja operacji przy przesłaniu wiadomości

Procesy i komunikacja w ramach jednego komputera

TigerMail będzie wykorzystywał rozproszone sieci *Ethereum* i *Swarm*. Do komunikacji z tymi sieciami użyte będą aplikacje klienckie *geth* i *Swarm* z projektu *Go Ethereum* (`https://geth.ethereum.org/`). *TigerMail* będzie się z nimi komunikował przez udostępniane przez nie interfejsy przez *Unix Domain Socket*.

Jako interfejsu użytkownika będzie można użyć dowolnego *MUA*. *TigerMail* będzie udostępniał dla nich interfejsy *POP3* i *SMTP* (syrunek 4.7).



Rysunek 4.7: Procesy TigerMail

Jak zaproponowana architektura spełnia postawione wymagania

W tym rozdziale znajdują się wymagania z rozdziału 4.1 i opis, jak przedstawiona powyżej architektura je spełnia.

Obsługuje asynchroniczną wymianę wiadomości. – Nadawca zapisuje wiadomość w *Swarm* i *Swarm Hash* w *Ethereum*. Odbiorca może pobrać te dane w dowolnym momencie, bez interakcji z nadawcą.

Korzysta z sieci rozproszonej. – Sieci *Ethereum* i *Swarm* są rozproszone.

Identyfikuje użytkowników przez klucz kryptograficzny. – Identyfikatorem użytkownika jest jego adres *Ethereum*, czyli klucz kryptograficzny.

Ujawnia innym użytkownikom minimum o tym, kto, komu, kiedy i co wysłał. – Jedyne, co jest ujawnione, to sam fakt wysłania wiadomości.

Szyfruje wiadomości algorytmem zapewniającym *forward secrecy*. – Użyty jest protokół *Signal*.

4.4 Moduły i komunikacja między modułami

Jak pokazałem w rozdziale 4.3, *TigerMail* będzie komunikował się z innymi użytkownikami za pośrednictwem klientów sieci *Ethereum* i *Swarm* nazywającymi się odpowiednio *geth* i *swarm*.

Do komunikacji z *geth* wykorzystane będą biblioteki dostarczone przez *aleth*. Biblioteka *libethereum* pozwala tworzyć transakcje, które są wysyłane do sieci przez *RPC*.

Do komunikacji ze *swarm* *TigerMail* będzie wykorzystywał *HTTP API* dostępne lokalnie przez *TCP*. Żeby uniknąć zagrożeń opisanych w rozdziale 4.5 *TigerMail*, przed nawiązaniem połączenia, wykorzysta *JSON-RPC* dostępne przez *Unix Domain Socket* do sprawdzenia numeru portu, pod którym *Swarm* oczekuje na połączenia.

Ponieważ do tworzenia transakcji w *Ethereum* i aktualizowania *Swarm Feed* potrzebne jest hasło do klucza *Ethereum*, *TigerMail* będzie pobierać je od użytkownika przy uruchomieniu i przechowywać w pamięci operacyjnej.

Użytkownik będzie komunikował się z *TigerMail*, wykorzystując wybrany *MUA* przez protokoły *POP3* i *IMAP*.

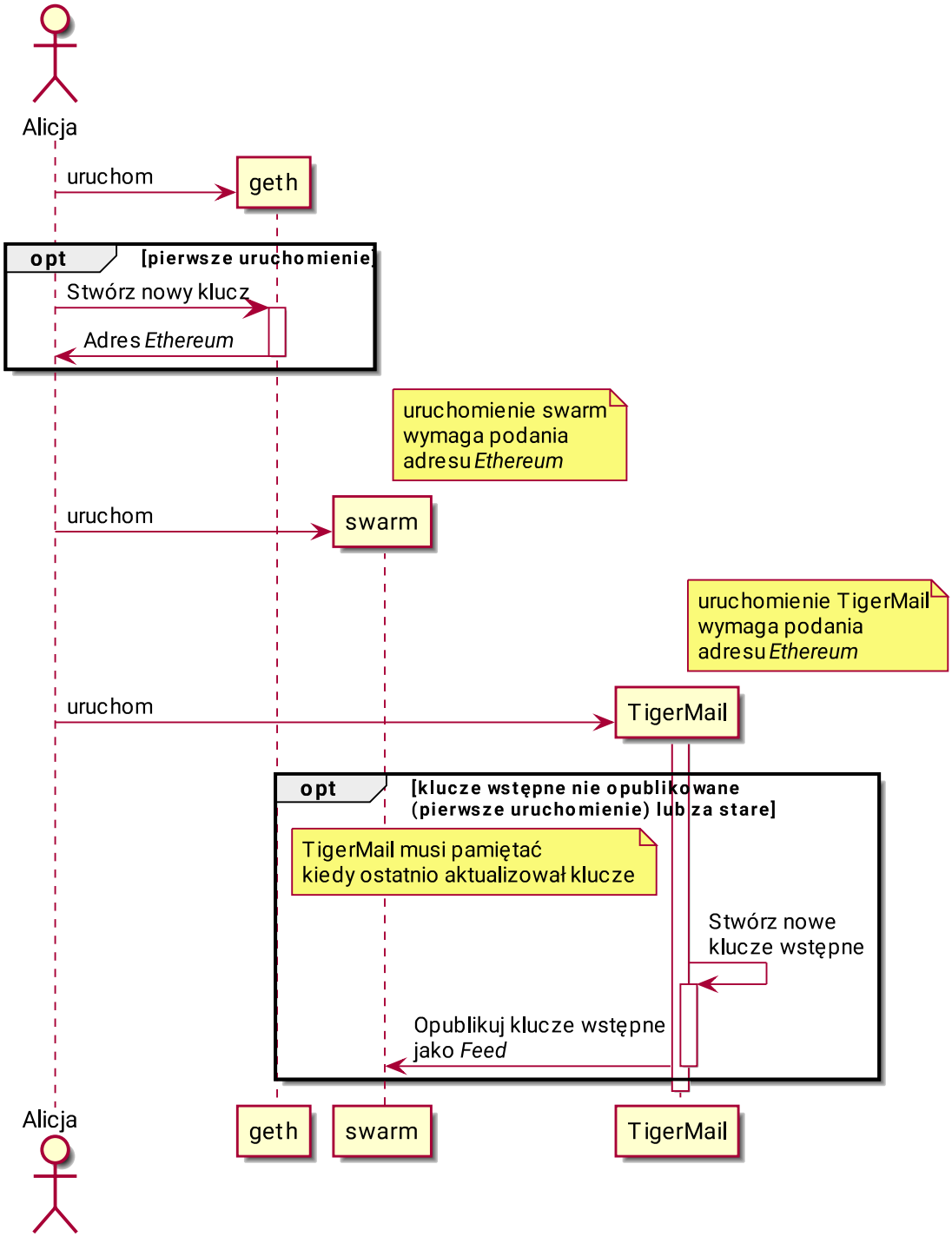
Inicjalizacja

Procesy *geth* i *swarm* nie komunikują się ze sobą, ale korzystają bezpośrednio z pliku z kluczem *Ethereum*. Dlatego przy pierwszym uruchomieniu trzeba będzie używać *geth* do stworzenia klucza. Procesy *swarm* i *TigerMail* będą wymagały podania adresu *Ethereum* przy uruchomieniu.

Aplikacja *TigerMail* przy uruchomieniu sprawdzi w swojej bazie, kiedy zostały opublikowane klucze jednorazowe. Wygeneruje i opublikuje nowe, jeżeli jest taka potrzeba. Klucze będą publikowane w *Swarm* jako *Swarm Feed* o określonym temacie, dzięki czemu nie trzeba będzie publikować dodatkowych identyfikatorów w *Ethereum*.

Za uruchamianie procesów odpowiedzialny będzie użytkownik. Byłoby możliwe, żeby *TigerMail* uruchamiał *geth* i *swarm* jako swoje podprocesy i nadzorował ich działanie, jednak implementacja tego nie jest priorytetowa.

Diagram sekwencji inicjalizacji *TigerMail* jest przedstawiony na rysunku 4.8.



Rysunek 4.8: Inicjalizacja TigerMail

Wysłanie wiadomości

Wysłanie wiadomości przez *TigerMail* będzie zaczynać się od wysłania jej przez użytkownika za pomocą *MUA* do aplikacji *TigerMail*. Na wstępie aplikacja sprawdzi, czy adres nadawcy podany przez protokół *SMTP* i w wiadomości zgadzają się z adresem podanym przy uruchomieniu.

Jeżeli jest to pierwsza wiadomość wysyłana do danego odbiorcy, *TigerMail* pobierze klucze wstępne ze *Swarm Feed* (zestaw zawiera jeden klucz podpisany i wiele niepodpisanych) i losowo wybierze jeden z niepodpisanych do stworzenia sesji. W komunikatorze *Signal*, gdzie klucze wstępne są przechowywane na serwerze, to serwer jest odpowiedzialny za wydawanie kolejnych kluczy kolejnym chętnym. W tym przypadku nie ma serwera, który mógłby to robić, a używanie kontraktu na *Ethereum* byłoby zbyt kosztowne i niepotrzebnie zdradzałoby informacje o tym, kto z kim zaczyna rozmawiać. Protokół *Signal* dopuszcza możliwość wielokrotnego wykorzystania tego samego klucza niepodpisanego przez kilku użytkowników lub w ogóle pominięcie go. W tym kontekście wybranie losowego klucza niepodpisanego jest rozwiązaniem wystarczająco dobrym.

Dla rozmów już rozpoczętych *TigerMail* będzie lokalnie przechowywał dane o sesji i używane w niej klucze.

TigerMail zaszyfruje wiadomość dla odbiorcy zgodnie z protokołem *Signal* i opublikuje ją w *Swarm*. W przypadku pierwszej wiadomości w sesji wygeneruje losowy klucz tymczasowy. Klucz prywatny będzie użyty razem z podpisanym kluczem publicznym odbiorcy do stworzenia klucza szyfrującego dla *Swarm Hash* przy użyciu funkcji *ECDH*, a klucz publiczny będzie dołączony do danych opublikowanych w kontrakcie. Dzięki temu odbiorca będzie mógł odszyfrować *Swarm Hash*, nie mając rozpoczętej sesji. W przeciwnym przypadku kluczem szyfrującym będzie ten sam, którym zaszyfrowana została wiadomość, a do opublikowanych danych będzie dołączony klucz publiczny identyfikujący *ratchet chain*, wymagany przez bibliotekę *Signal*. Następnie aplikacja zaszyfruje *Swarm Hash*, wykorzystując algorytm *AES-256*.

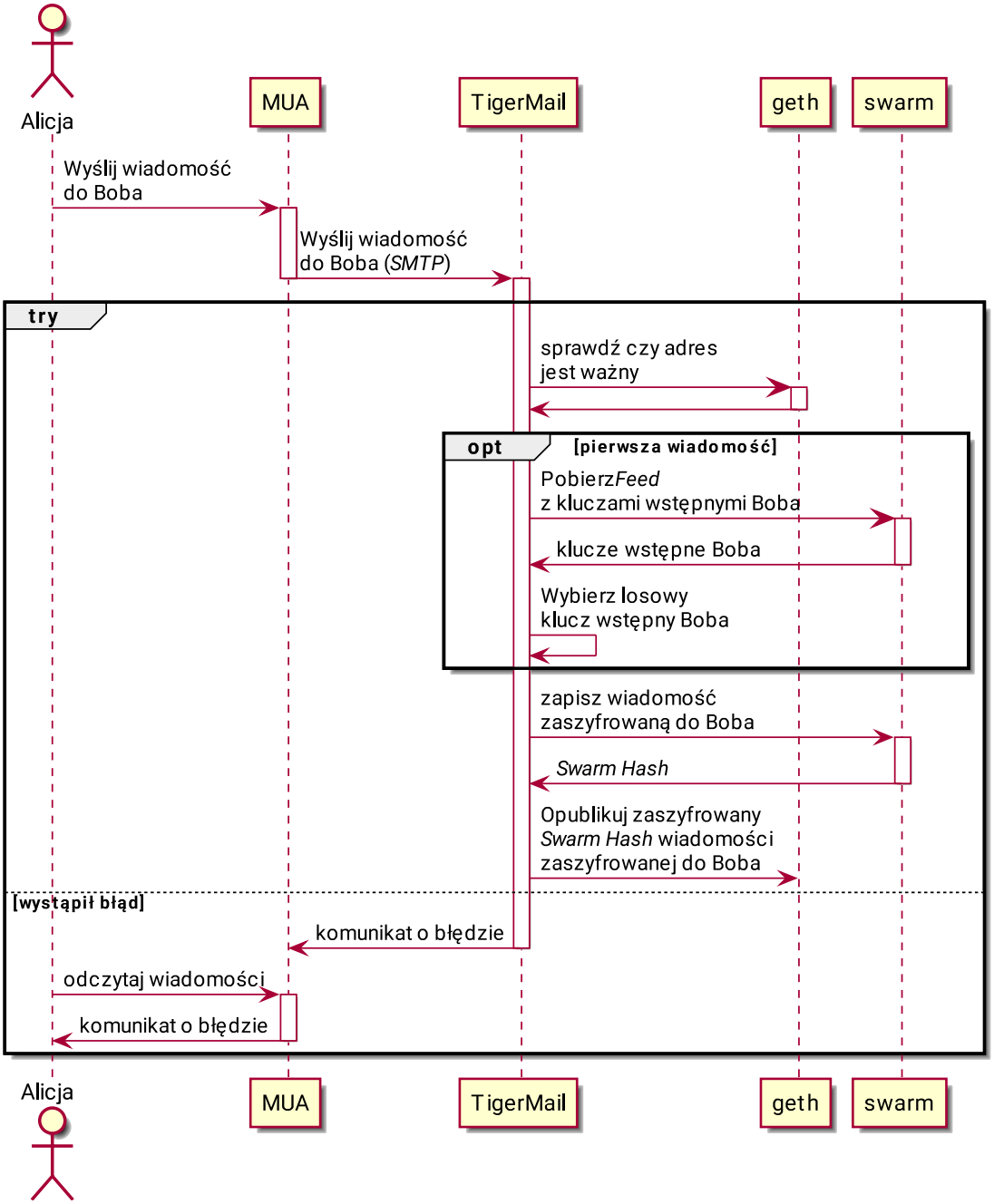
Kod *HMAC-SHA256* będzie wykorzystany do zabezpieczenia opublikowanych danych. Dzięki temu odbiorca będzie miał jednoznaczną odpowiedź na pytanie, czy klucz szyfrujący wygenerowany po jego stronie był prawidłowy. Jest to istotne, ponieważ sam *Swarm Hash* jest skrótem kryptograficznym, więc trudno jest rozpoznać poprawnie odszyfrowane dane.

Podsumowując, danymi publikowanymi w kontrakcie będą zaszyfrowany *Swarm Hash*, klucz publiczny albo tymczasowy dla pierwszej wiadomości albo identyfikujący *ratchet chain* i kod *HMAC*.

Algorytmy *AES-256* i *HMAC-SHA256* będą wykorzystane, ponieważ są już używane przez *Signal*).

W przypadku jakiegokolwiek błędu komunikat zostanie przekazany użytkownikowi w postaci wiadomości pocztowej.

Diagram sekwencji wysyłania wiadomości w *TigerMail* jest przedstawiony na rysunku 4.9.



Rysunek 4.9: Wysłanie wiadomości

Odebranie wiadomości

Odbieranie wiadomości będzie wymagać od *TigerMail* aktywnego odpytywania *geth* w poszukiwaniu nowych zdarzeń w kontrakcie (na przykład co 15 minut). W tym celu będzie musiał pamiętać ostatni sprawdzony numer bloku *Ethereum*.

TigerMail będzie próbował odszyfrować każdy nowy *Swarm Hash* opublikowanym kluczem podpisanym lub kluczami istniejących sesji. Użycie kodu *HMAC* pozwoli stwierdzić, czy użyty klucz był właściwy i odszyfrowywanie przebiegło poprawnie. Dla odszyfrowanych *Swarm Hash* *TigerMail* pobierze dane ze *Swarm* i odszyfruje zgodnie z protokołem *Signal*.

Żeby uniknąć możliwości dokonania *email spoofing*, *TigerMail* będzie porównywał adres nadawcy podany w wiadomości z adresem, z którego zostało wysłane zdarzenie w kontrakcie *Ethereum*. Jeżeli adresy nie będą się zgadzać, wiadomość zostanie odrzucona.

Po weryfikacji adresu nadawcy *TigerMail* sprawdzi, czy adres został unieważniony. Jeżeli tak, to wiadomość zostanie odrzucona. Na tym etapie istotne jest, żeby sprawdzenie brało pod uwagę chronologię zdarzeń: jeżeli klucz został unieważniony po wysłaniu wiadomości, wiadomość powinna zostać przyjęta.

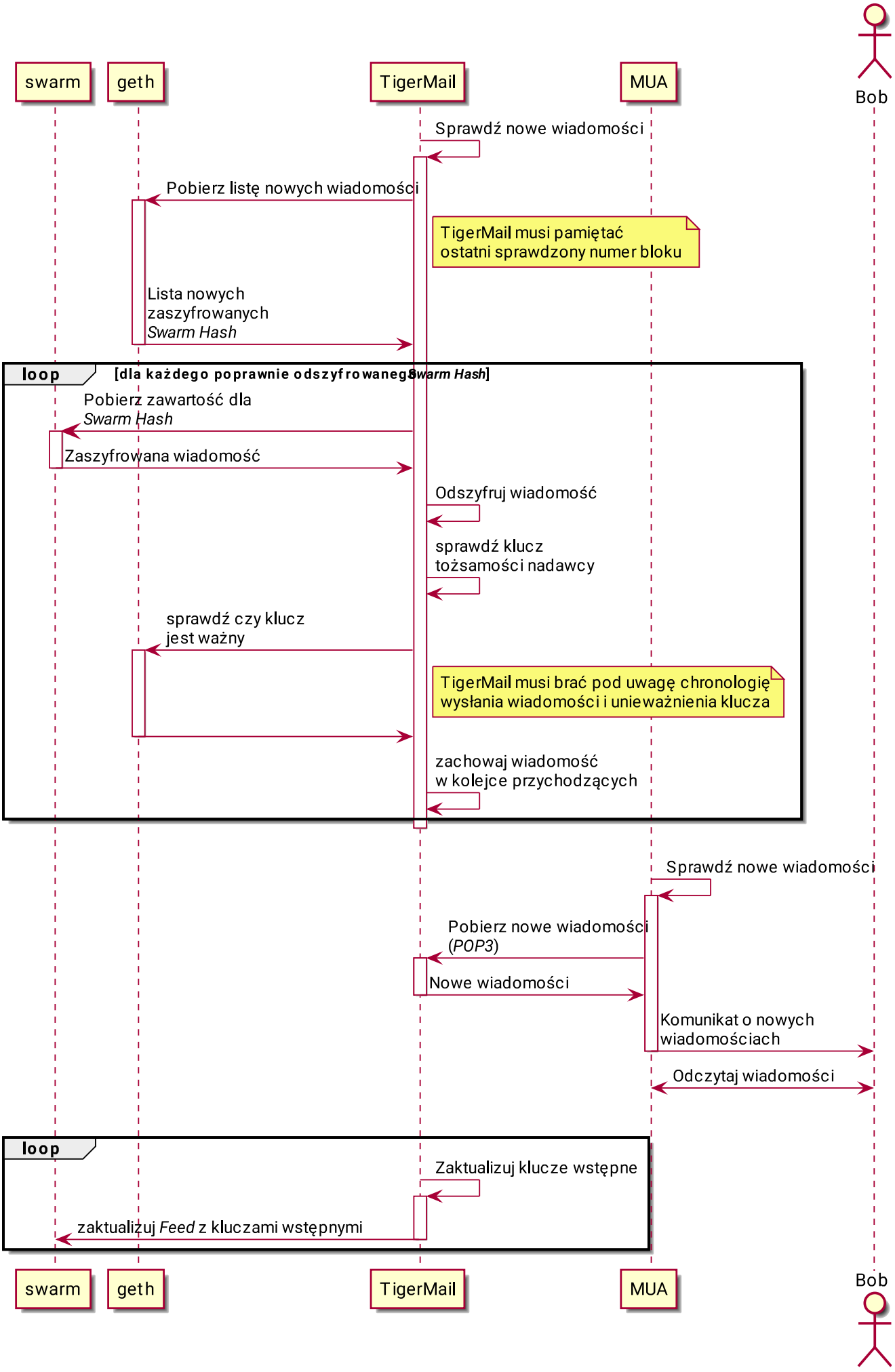
Po weryfikacji wiadomości trafi ona do kolejki, z której *MUA* będzie mógł ją pobrać przez protokół *POP3*.

Diagram sekwencji odbierania wiadomości w *TigerMail* jest przedstawiony na rysunku 4.10.

Aktualizacja kluczy jednorazowych

W określonych odstępach czasu (na przykład co tydzień lub miesiąc) *TigerMail* będzie generował i publikował w *Swarm Feed* nowy zestaw kluczy wstępnych. *TigerMail* musi przechowywać informację o dacie ostatniej aktualizacji.

Diagram sekwencji akutalizacji kluczy jednorazowych w *TigerMail* jest przedstawiony na rysunku 4.10.



Rysunek 4.10: Odebranie wiadomości

Unieważnienie klucza

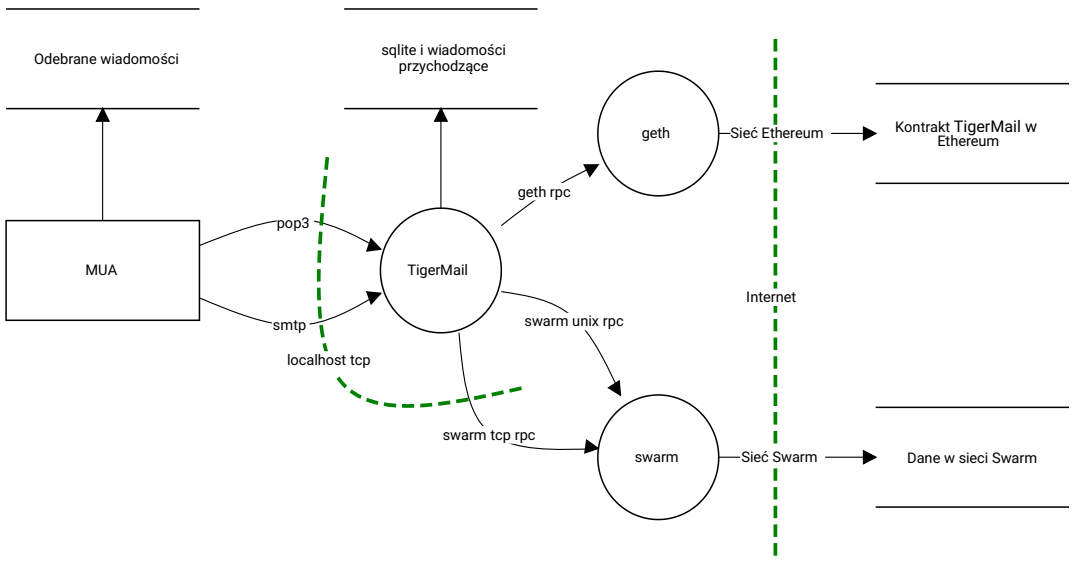
Unieważnienie klucza będzie polegało na opublikowaniu w kontrakcie *Ethereum* zdarzenia zawierającego unieważniany adres *Ethereum*. Tylko właściciel klucza będzie mógł unieważnić swój adres.

Unieważniony adres będzie nadal normalnie działającym kluczem *Ethereum*, ale aplikacja *TigerMail* nie pozwoli wysłać wiadomości na taki adres i odrzuci wiadomości przychodzące z niego.

4.5 Model zagrożeń

Do modelowania zagrożeń użyłem diagramu przepływu danych (rys. 4.11) i metody *STRIDE* [40]. Metoda *STRIDE* polega na identyfikowaniu następujących rodzajów zagrożeń (nazwa pochodzi od pierwszych liter):

Spoofing – Podszywanie się pod innego użytkownika.
Tampering – Modyfikowanie danych.
Repudiation – Zaprzeczalność.
Information disclosure – Ujawnienie informacji.
Denial of Service – Odmowa usługi.
Elevation of privileges – Eskalacja uprawnień.



Rysunek 4.11: Diagram przepływu danych.

Wytypowałem następujące zagrożenia i metody obrony:

- **Rodzaj** Podszywanie.
  Moduł Lokalne połączenie *SMTP* i *POP3* przez protokół *TCP*.
  Opis Inny proces, uruchomiony z konta innego użytkownika, może otworzyć gniazdo (*socket*) zanim *TigerMail* to zrobi. Program *MUA* nie ma możliwości stwierdzić, czy jest podłączony do właściwego serwera.
  Przeciwdziałanie Można wymusić w programie *MUA* wykonywanie połączeń przez *Unix Domain Socket* przy użyciu narzędzia *ip2unix*.

- **Rodzaj** Podszywanie.
  Moduł Lokalne połączenie do *swarm* przez protokół *TCP*.
  Opis Inny proces, uruchomiony z konta innego użytkownika, może otworzyć gniazdo (*socket*) zanim *swarm* to zrobi. Program *TigerMail* nie ma możliwości stwierdzić, czy jest podłączony do właściwego serwera.
  Przeciwdziałanie Można dostać numer portu *TCP* przez drugi interfejs *JSON-RPC*, który *swarm* udostępnia przez *Unix Domain Socket*.

- **Rodzaj** Podszywanie.
  Moduł Sieć *Ethereum*.
  Opis Atakujący chce wysłać wiadomość, podszywając się pod innego użytkownika.
  Przeciwdziałanie Wszystkie transakcje są podpisane kluczem nadawcy.

- **Rodzaj** Ujawnienie lub modyfikacja danych.
  Moduł Dane przechowywane na dysku przez *TigerMail*.
  Opis Aplikacja musi przechowywać swój stan, klucze kryptograficzne sesji *Signal* i odebrane wiadomości (do momentu aż zostaną wysłane do *MUA*).
  Przeciwdziałanie Ponieważ programy *MUA* zazwyczaj nie zabezpieczają przechowywanych na dysku odebranych wiadomości, jakiekolwiek bardziej zaawansowane zabezpieczenie danych programu *TigerMail* nie zwiększy bezpieczeństwa całego systemu. Wystarczającym zabezpieczeniem przed innymi użytkownikami są odpowiednie uprawnienia dostępu do katalogu. Do zabezpieczenia danych przed osobą mającą fizyczny dostęp do dysku należy stosować szyfrowanie na poziomie systemu plików.

- **Rodzaj** Modyfikowanie danych.
  Moduł Kontrakt *TigerMail* w *Ethereum*.
  Opis Atakujący chce zmienić wysłaną wiadomość.
  Przeciwdziałanie Zdarzenie jest podpisane przez nadawcę i, raz zapisane w *blockchain*, nie może być usunięte ani zmodyfikowane.

- **Rodzaj** Modyfikowanie danych.
  Moduł Dane w sieci *Swarm*.
  Opis Atakujący chce zmienić treść wiadomości.
  Przeciwdziałanie W sieci *Swarm* dane są identyfikowane i weryfikowane przez sumę kontrolną.

- **Rodzaj** Zaprzeczalność.
  Moduł Kontrakt *TigerMail* w *Ethereum*.
  Opis Atakujący chce zaprzeczyć temu, że właściciel klucza go unieważnił.
  Przeciwdziałanie Zdarzenie raz zapisane w *blockchain* nie może być usunięte ani zmodyfikowane.

- **Rodzaj** Ujawnienie informacji.
  Moduł Kontrakt *TigerMail* w *Ethereum*.
  Opis Atakujący chce odczytać treść lub zidentyfikować odbiorcę wiadomości.
  Przeciwdziałanie Zdarzenie zapisane w kontrakcie nie zawiera identyfikatora odbiorcy. *Swarm Hash* prowadzący do treści wiadomości jest zaszyfrowany jednorazowym kluczem z protokołu *Signal* i zabezpieczony kodem *HMAC*.

- **Rodzaj** Ujawnienie informacji.
  Moduł Scieć *Swarm*.
  Opis Atakujący chce odkryć odbiorcę wiadomości przez korelację w czasie pojawienia się zdarzenia w *Ethereum* i pobierania danych ze *Swarm*.
  Przeciwdziałanie *TigerMail* pobiera dane wiadomości ze *Swarm* w dużych odstępach czasu. Dzięki temu istnieje mała szansa, że wiadomość zostanie pobrana zaraz po jej wysłaniu.

- **Rodzaj** Ujawnienie informacji.
  Moduł Dane w sieci *Swarm*.
  Opis Atakujący chce odczytać treść wiadomości.
  Przeciwdziałanie Wiadomość jest zaszyfrowana protokołem *Signal*.

- **Rodzaj** Odmowa usługi.
  Moduł Sieci *Ethereum* i *Swarm*.
  Opis Atakujący chce zablokować możliwość wymiany wiadomości.
  Przeciwdziałanie To są sieci rozproszone. Z wyjątkiem sytuacji, gdy atakujący kontroluje router bezpośrednio przy ofierze, to nie ma możliwości zablokowania komunikacji.

4.6 Analiza bezpieczeństwa

W tym rozdziale znajduje się analiza bezpieczeństwa aplikacji *TigerMail*, w szczególności omówienie wykorzystywanych kluczy, obiegu danych, mechanizmów zapewniających poufność i integralność komunikacji.

Zarządzanie kluczami

W *TigerMail* występują dwa rodzaje kluczy asymetrycznych: długotrwałe i krótkotrwałe. Długotrwałe to klucz *Ethereum* i klucz tożsamości *Signal*, które są wykorzystywane przez cały czas korzystania z adresu *Ethereum* do wysyłania i odbierania wiadomości. Klucze krótkotrwałe to klucze wstępne publikowane w *Swarm Feed* i klucze przesyłane razem z wiadomością jako część *Diffie-Hellman ratchet* protokołu *Signal*.

Wszystkie klucze prywatne są przechowywane na urządzeniach rozmówców i nie są przesyłane przez sieć.

W *TigerMail* wykorzystywane są też klucze symetryczne w ramach *Diffie-Hellman ratchet*. Te klucze również nie są przesyłane przez sieć. Algorytm *Diffie-Hellman ratchet* jest odpowiedzialny za wyliczanie tych kluczy u rozmówców na podstawie przesyłanych publicznych kluczy asymetrycznych.

Tworzenie kluczy *Ethereum* nie jest częścią *TigerMail*. Dla protokołu *Signal* za tworzenie nowych kluczy odpowiedzialna jest biblioteka *libsignal-protocol-c*, która wymaga podania jedynie generatora liczb pseudolosowych. W mojej implementacji użyte zostały funkcje z biblioteki *OpenSSL* `RAND_load_file` do pobrania losowych danych z pliku `/dev/urandom` i `RAND_bytes` do generowania danych.

Klucze wstępne są generowane w stałych odstępach czasu. Klucze prywatne są przechowywane przez dwie iteracje, po czym trwale usuwane. Klucze publiczne opublikowane w *Swarm Feed* mogą być dostępne bardzo długo, ponieważ *Swarm* nie daje żadnych gwarancji w kwestii usuwania danych. W praktyce są zazwyczaj usuwane po kilku tygodniach.

Klucze używane do odszyfrowywania wiadomości są zarządzane przez bibliotekę *Signal*. Pozwala ona na odbieranie wiadomości w zmienionej kolejności (na przykład odebranie trzeciej, zanim dotrze druga) dzięki przechowywaniu starych kluczy (pomimo odebrania trzeciej wiadomości, klucz do drugiej jest wciąż dostępny). Zapisywane jest 5 ostatnich kluczy tworzących łańcuch odbierający w *Diffie-Hellman ratchet* oraz do 2000 kluczy w ramach jednego łańcucha *Symmetric-key ratchet* [15].

Obieg danych

Jak przedstawiłem w rozdziale 4.3 i na rysunku 4.6, przesyłane wiadomości są szyfrowane *end-to-end*. Dane przechowywane w *Swarm*, czyli zaszyfrowane wiadomości i publiczne klucze wstępne, w nieokreślonym, ale skończonym czasie powinny być usunięte (patrz rozdział 4.2.2). Zaszyfrowane odnośniki zapisane w kontrakcie *Ethereum* zostają dostępne na zawsze.

Na komputerze nadawcy *MUA* może przechowywać kopię wysłanej wiadomości. Analogicznie na komputerze odbiorcy *MUA* może przechowywać odebrane wiadomości. Aplikacja *TigerMail* przechowuje odszyfrowane przychodzące wiadomości w kolejce, aż zostaną odebrane przez *MUA*. Jak napisałem w rozdziale 4.5, te dane powinny być chronione szyfrowaniem całego dysku.

Poufność

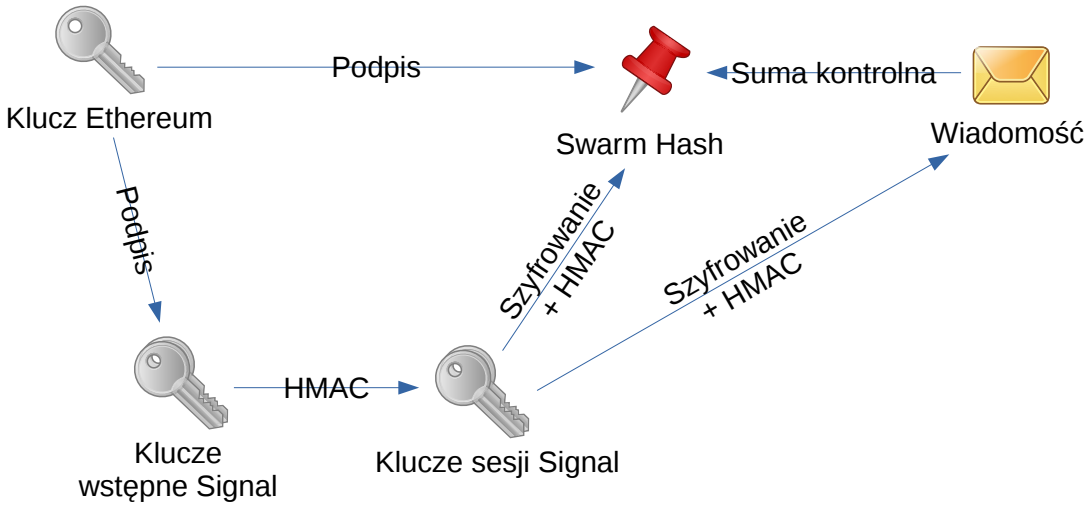
Poufność danych jest zagwarantowana przez protokół *Signal*, dzięki któremu wszystkie wiadomości są zaszyfrowane z zachowaniem *forward secrecy*. Dodatkowo dostęp do samej zaszyfrowanej wiadomości umieszczonej w *Swarm* jest ograniczony przez szyfrowanie *Swarm Hash* zapisywanego w *Ethereum*. Dane zapisywane w *Swarm* są dzielone na małe fragmenty i przechowywane przez różnych użytkowników, więc nikt w sieci nie posiada całej wiadomości.

Zachowanie *forward secrecy* jest niedoskonałe, ponieważ *Swarm* nie daje gwarancji usunięcia danych. Nie jest to jednak krytycznym problemem, ponieważ klucze prywatne pozwalające na odszyfrowanie *Swarm Hash* z kontraktu w *Ethereum* są trwale usuwane.

Integralność

Integralność danych jest zapewniona przez następujące elementy (rysunek 4.12):

- Dowolne dane publikowane w *Swarm Feed*, w tym klucze wstępne *TigerMail*, są podpisywane kluczem *Ethereum*.
- Transakcje publikowane w sieci *Ethereum*, w tym zdarzenia publikowane w kontrakcie, są podpisywane kluczami *Ethereum*. *TigerMail* nie sprawdza tych podpisów, ponieważ jest to weryfikowane przez klienta sieci *Ethereum*.
- *Swarm Hash* opublikowane w kontrakcie *Ethereum* są zaszyfrowane i zabezpieczone kodem *HMAC*.



Rysunek 4.12: Integralność danych

- Klucze publiczne wykorzystywane przez *Diffie-Hellman ratchet*, dołączane do przesyłanej wiadomości są niezaszyfrowane, ale zabezpieczone kodem *HMAC*.
- Dane umieszczone w *Swarm* są identyfikowane i weryfikowane przez sumę kontrolną.

4.7 Implementacja aplikacji

W tym rozdziale znajduje się opis stworzonej przeze mnie implementacji aplikacji *TigerMail*.

4.7.1 Wybór języka programowania i wykorzystanych bibliotek

W celu implementacji *TigerMail* wybrałem język *C++*, ponieważ znam go najlepiej. Rozważałem jeszcze użycie języka *Rust*[1], ale musiałem go odrzucić ze względu na brak wielu potrzebnych bibliotek.

Biblioteka realizująca protokół *Signal*, stworzona przez twórców komunikatora *Signal* jest dostępna między innymi w języku C (*libsignal-protocol-c*), więc można ją wykorzystać w programie napisanym w *C++*. Nie zawiera ona funkcji kryptograficznych, serializacji i przechowywania danych, żeby nie narzucać wyboru zależnych bibliotek i oczekuje od użytkownika samodzielnej realizacji tych funkcji. Do kryptografii wybrałem biblioteki *LibreSSL* (*fork* popularnego *OpenSSL*)

[1] https://www.rust-lang.org/

i *CryptoPP* (*LibreSSL* nie byłoby potrzebne, gdyby *CryptoPP* zawierało implementację algorytmu *Rijndael* dla 256-bitowych kluczy). Do serializacji danych wybrałem bibliotekę *protobuf*. Do przechowywania danych wybrałem bibliotekę *sqlite* z dodatkiem *sqlite_ modern_ cpp*.

Do obsługi sieci, zarówno w roli serwera, jak i klienta, wykorzystałem bibliotekę *Boost Asio*. Dodatkowo pozwala ona wywoływać funkcje w interwałach czasu.

Nie znalazłem gotowych bibliotek w *C++* obsługujących komunikację z programami *geth* i *swarm*, więc musiałem zaimplementować ją samodzielnie. W tym celu, oprócz *Boost Asio*, wykorzystałem biblioteki *jsoncpp* i *Boost Beast* (obsługuje protokół *HTTP*). Rozważałem wykorzystanie biblioteki *libjson-rpc-cpp*[2], jednak wykorzystuje ona własną implementację komunikacji sieciowej i nie udostępnia asynchronicznego *API*, które by pasowało do *Boost Asio*.

Do komunikacji z *geth* wykorzystałem bibliotekę *libethereum*, rozwijaną jako część programu *aleth* (węzeł sieci *Ethereum*, podobny do *geth*, zaimplementowany w *C++*). Dostarcza ona funkcje do tworzenia i podpisywania transakcji, które można wysłać do sieci *Ethereum*.

Do aktualizowania danych w *Swarm Feed* wykorzystywany będzie *swarm* jako polecenie uruchamiane w podprocesie, ponieważ żeby zrobić to przez *HTTP API*, trzeba stworzyć odpowiedni podpis i nie ma biblioteki w *C++*, która to implementuje.

4.7.2 Architektura modułów sieciowych

Aplikacja *TigerMail* wykorzystuje pętlę zdarzeń z biblioteki *Boost Asio*.

Elementy pasywne

Serwery *POP3* i *SMTP*, oczekują na połączenia przychodzące i reagują na przysłane dane. *POP3* i *SMTP* są protokołami tekstowymi i wysyłają dane tylko w odpowiedzi na odebrane komendy. Ze względu na to podobieństwo, wydzieliłem kilka niezależnych elementów:

Serwer obsługujący przychodzące połączenia Implementacja znajduje się w katalogu `pop3_smtp_common/server/` i używa przestrzeni nazw `TigerMail::POP3_SMTP_common::server`. Przy tworzeniu wymaga przekazania obiektu implementującego interfejs `SessionHandlerInterface` (listing 4.2).

[2] https://github.com/cinemast/libjson-rpc-cpp

Listing 4.2: Interfejs `SessionHandlerInterface`

```
class SessionHandlerInterface
{
public:
    virtual std::string handle_hello() = 0;
    /// return: response, should disconnect after responding
    virtual std::tuple<std::string, bool>
        handle_input(std::string_view input) = 0;
    virtual std::string handle_stop() = 0;
};
```

Metoda `handle_hello` zwraca tekst wysyłany do klienta po nawiązaniu połączenia. Metoda `handle_input` przyjmuje odebrany przez serwer tekst. Zwraca tekst do wysłania i wartość logiczną oznaczającą czy serwer ma zamknąć połączenie. Metoda `handle_stop` zwraca tekst do wysłania przed rozłączeniem, gdy serwer jest wyłączany.

Przykładowy serwer, który zwraca wysłany do niego tekst, znajduje się w pliku `pop3_smtp_common/server/simple_echo_server.cpp`.

Parser Implementacja znajduje się w plikach `pop3_smtp_common/parser.*` i używa przestrzeni nazw `TigerMail::POP3_SMTP_common`. Przyjmuje tekst wejściowy w dowolnych fragmentach i dzieli na linie rozdzielone znakami `\r\n`. Odrzuca linie, które przekraczają zadany limit długości.

Wymaga przekazania obiektu implementującego interfejs `CommandProcessorClientInterface` (listing 4.3).

Listing 4.3: Interfejs `CommandProcessorClientInterface`

```
class CommandProcessorClientInterface
{
public:
    virtual void command_too_long() = 0;
    virtual void parse_command(std::string_view command) = 0;
};
```

Dla każdej linii wywołuje metodę `parse_command`. Jeżeli linia przekroczyła limit długości, wywołuje metodę `command_too_long`.

Elementy aktywne

Funkcje wywoływane w interwałach czasu, odpowiedzialne za publikowanie kluczy wstępnych lub odpytywanie klientów *Ethereum* i *Swarm* o przychodzące wiadomości, uruchamiane są przez funkcję `asio::spawn`, która uruchamia je jako współprogramy (ang. *coroutine*). Używanie funkcji sieciowych ze współprogramów wymaga przekazania do nich obiektu typu `boost::asio::yield_context`.

Żeby ułatwić używanie kodu w testach jednostkowych i dla uniezależnienia interfejsów klas od biblioteki sieciowej, obiekt `yield` jest przekazywany jako `std::any`. Dzięki temu w testach mogę zaimplementować sztuczne aplikacje klienckie *geth* i *swarm*.

4.7.3 Klient geth

Implementacja klienta *geth* znajduje się w katalogu `ethereum` i wykorzystuje przestrzeń nazw `TigerMail::Ethereum`.

Klasa `RPCClient` zawiera implementacje niezbędnych wywołań metod *JSON-RPC*, opisanych na stronie https://github.com/ethereum/wiki/wiki/JSON-RPC.

Plik `tigermail_contract.sol` zawiera kod kontraktu *TigerMail* dla *Ethereum*. W podkatalogu `contract_deployment` znajudją się skrypty pozwalające na kompilację kontraktu i umieszczenie go na *blockchain*.

Klasa `TigerMailContract` wykorzystuje `RPCClient` do komunikacji z *geth* i implementuje interakcje z kontraktem *TigerMail*. Dane do transakcji kodowane są zgodnie z opisem na stronie https://solidity.readthedocs.io/en/develop/abi-spec.html.

4.7.4 Klient swarm

Implementacja klienta *swarm* znajduje się w katalogu `swarm` i wykorzystuje przestrzeń nazw `TigerMail::Swarm`.

W plikach `ipc_client.*` znajdują się funkcje do odczytania numeru portu usługi *HTTP* z połączenia przez *Unix Domain Socket*.

Klasa `TCPClient` zawiera implementacje niezbędnych wywołań metod *JSON-RPC*, opisanych na stronie https://swarm-guide.readthedocs.io/en/latest/dapp_developer/index.html#id17. Jak napisałem w rozdziale 4.7.1, z powodu braku biblioteki dostarczającej odpowiednich funkcji kryptograficznych, aktualizacja *Swarm Feed* odbywa się przez wywołanie polecenia `swarm feed update` w podprocesie.

4.7.5 Serwer POP3

Implementacja serwera *POP3* znajduje się w katalogu `pop3` i wykorzystuje przestrzeń nazw `TigerMail::POP3`.

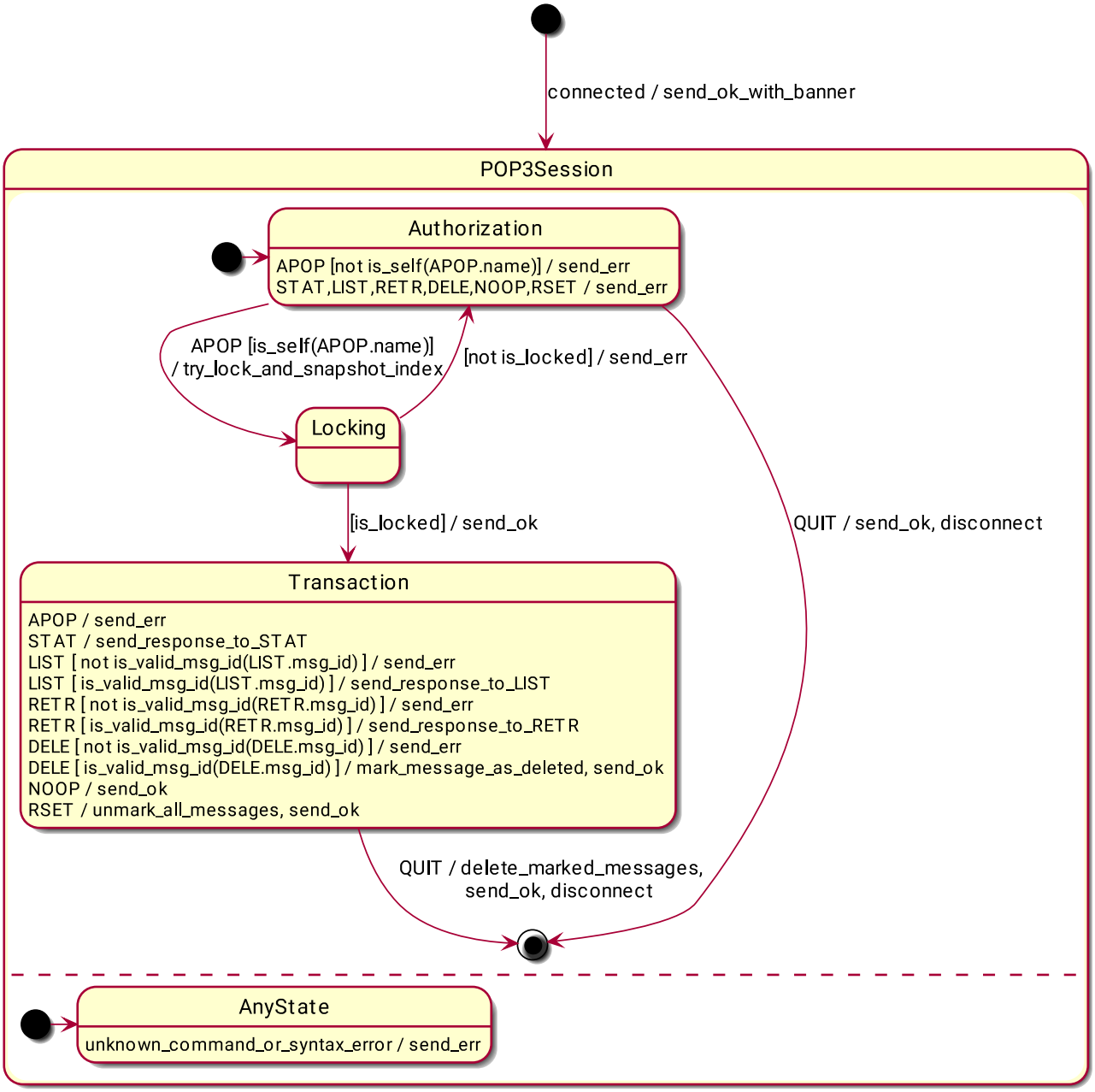
Na podstawie dokumentu *RFC* 1939 [64] stworzyłem maszynę stanów (rysunek 4.13), wykorzystując bibliotekę *SML*. Stany, akcje, warunki i tablica przejść są zaimplementowane w plikach `state_machine.*`. Zdarzenia są w nagłówku `events.hpp`.

Serwer *POP3* jest zaimplementowany jako klasa `SessionHandler`, która implementuje interfejs `SessionHandlerInterface` opisany w rozdziale 4.7.2. Przychodzące polecenia są przekazywane do parsera zaimplementowanego w plikach `parser.*`. Polecenia są dalej przekazywane do maszyny stanów jako zdarzenia. Odpowiedzi wysyłane do klienta są formatowane przez funkcje z plików `format_response.*`, umieszczane w buforze w `SessionHandler` i wysyłane do klienta po zakończeniu przetwarzania zdarzenia przez maszynę stanów.

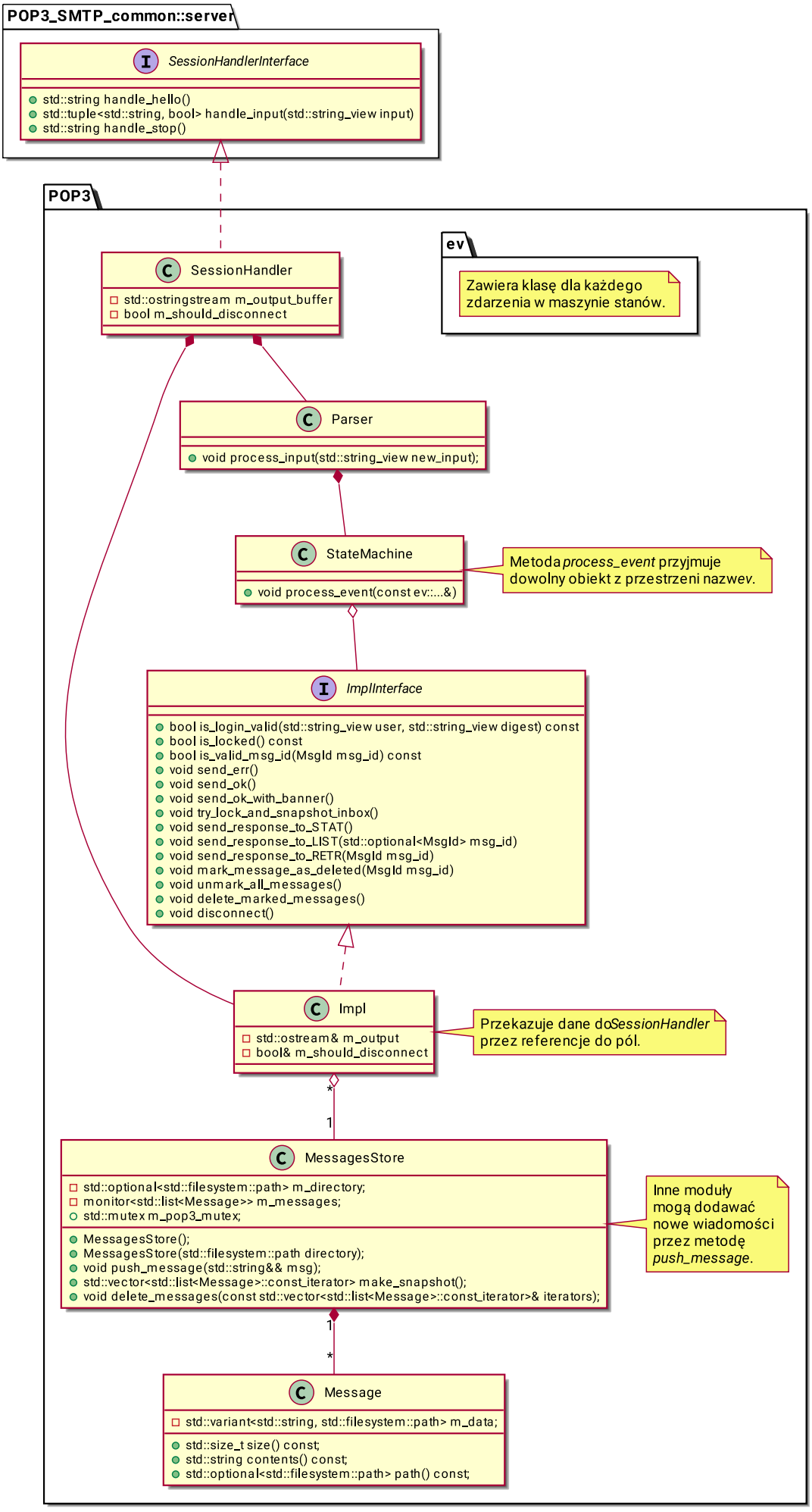
Maszyna stanów wywołuje działania przez interfejs `ImplInterface`. To pozwala testować parser niezależnie od logiki serwera. Klasa `Impl`, implementująca ten interfejs i znajdująca się w plikach `session_handler.*`, operuje na wiadomościach przechowywanych w klasie `MessagesStore`.

Klasa `MessagesStore`, znajdująca się w plikach `messages_store.*`, przechowuje wiadomości, które mają być odebrane przez użytkownika. Jeżeli przy tworzeniu obiektu poda się ścieżkę do katalogu, to wiadomości są przechowywane w plikach i mogą być wczytane ponownie, przy następnym uruchomieniu programu. W przeciwnym przypadku wiadomości przechowywane są w pamięci (ta opcja jest używana tylko w testach).

Diagram klas znajduje się na rysunku 4.14.



Rysunek 4.13: Maszyna stanów *POP3*



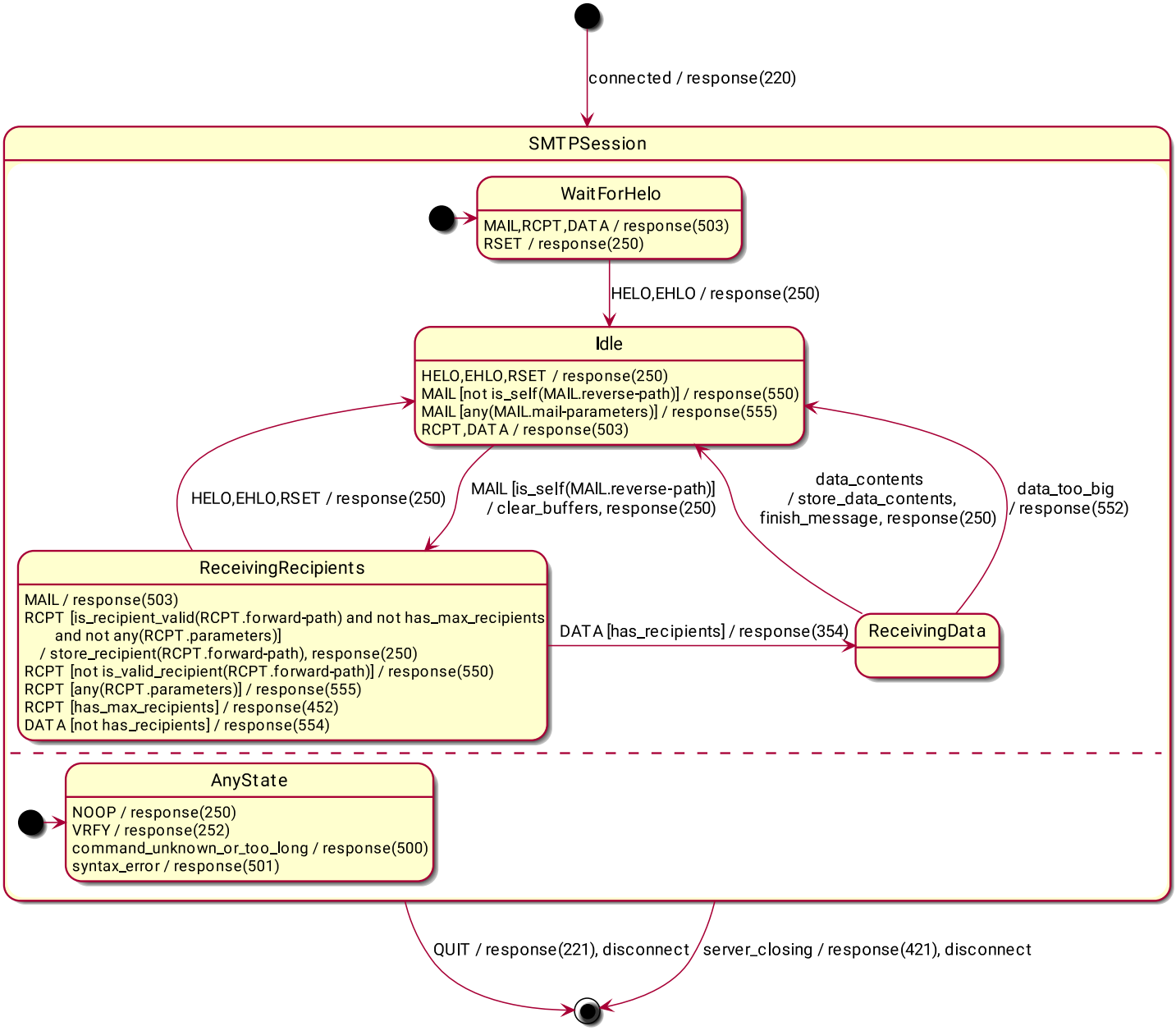
Rysunek 4.14: Diagram klas *POP3*

4.7.6 Serwer SMTP

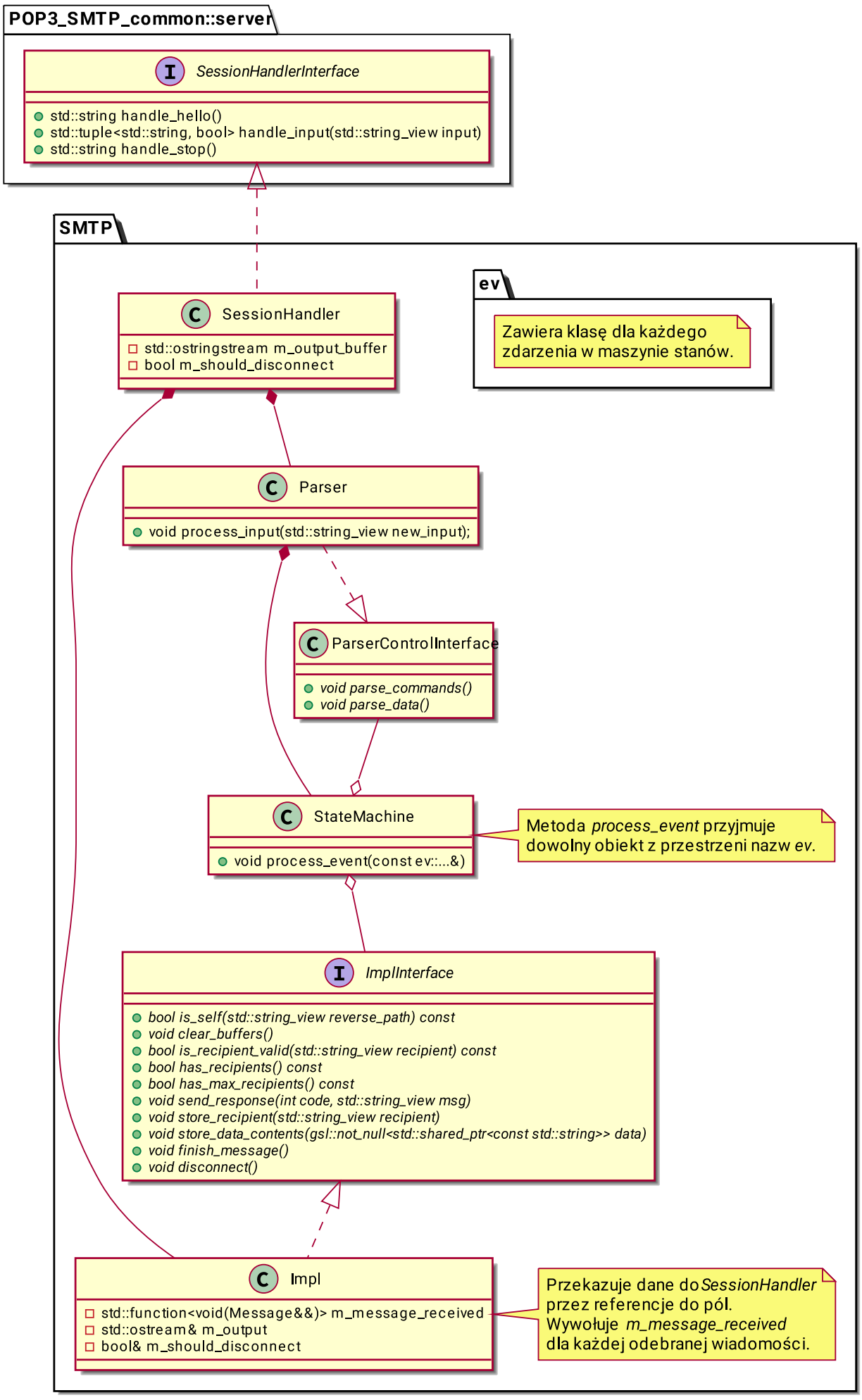
Implementacja serwera *SMTP* znajduje się w katalogu `smtp` i wykorzystuje przestrzeń nazw `TigerMail::SMTP`.

Architektura serwera *SMTP* jest bardzo podobna do serwera *POP3* opisanego powyżej. Maszynę stanów stworzyłem na podstawie dokumentu [39] (rysunek 4.15), wykorzystując bibliotekę *SML*. Serwer *SMTP* przyjmuje wiadomości przychodzące od użytkownika, więc jedynym punktem komunikacji z zewnętrznymi modułami jest przekazywany do konstruktora klasy `SessionHandler` obiekt typu `std::function<void(Message&&)>`. Dodatkowo `Parser` implementuje interfejs `ParserControlInterface`, który jest używany przez maszynę stanów do przełączania go pomiędzy parsowaniem komend i przyjmowaniem treści wiadomości.

Diagram klas znajduje się na rysunku 4.16.



Rysunek 4.15: Maszyna stanów *SMTP*



Rysunek 4.16: Diagram klas *SMTP*

4.7.7 Wykorzystanie biblioteki Signal

Jak napisałem w rozdziale 4.7.1, biblioteka *Signal* jest zaimplementowana w języku C i wymaga od użytkownika zaimplementowania kryptografii, serializacji i przechowywania danych. Implementacje tych komponentów oraz warstwa ułatwiająca korzystanie z tej biblioteki w języku *C++* znajdują się w katalogu `signal` i wykorzystują przestrzeń nazw `TigerMail::Signal`.

Warstwa ułatwiająca korzystanie z biblioteki Signal w języku *C++*

Biblioteka `libsignal-protocol-c`, wykorzystuje typowe dla języka *C* mechanizmy, takie jak ręczne zarządzanie czasem życia obiektów i komunikowanie o błędach przez wartość zwracaną z funkcji. Język *C++* dostarcza ułatwienia w postaci destruktorów i wyjątków.

Większość struktur z biblioteki `libsignal-protocol-c` posiada wbudowany mechanizm zliczania referencji. Do modyfikowania licznika używa się makr `SIGNAL_REF` i `SIGNAL_UNREF`. Klasa szablonowa `Ref`, zaimplementowana w pliku `common.hpp`, automatyzuje obsługę tego mechanizmu dla dowolnego typu.

Klasa `SignalBufferRef`, zaimplementowana w plikach `signal_buffer.*`, opakowuje typ `signal_buffer`, dodając automatyczne niszczenie bufora w destruktorze, oraz dodaje metody typowe dla kontenerów z biblioteki standardowej *C++*.

Klasa `SignalError`, zaimplementowana w pliku `common.hpp`, dziedziczy po `std::exception` i jest używana do przekazywania błędów przez wyjątki.

W podkatalogu `signalpp` znajdują się wymagane funkcje opakowujące funkcje z biblioteki `libsignal-protocol-c`. Opakowanie polega na zamienieniu zwracania kodu błędu na rzucenie wyjątku i zwracanie wartości z funkcji, zamiast przyjmowania wskaźnika w argumentach. Nazwy plików i funkcji dla przejrzystości są takie same jak w bibliotece `libsignal-protocol-c`. Przykładowe opakowanie funkcji znajduje się w listingu 4.4.

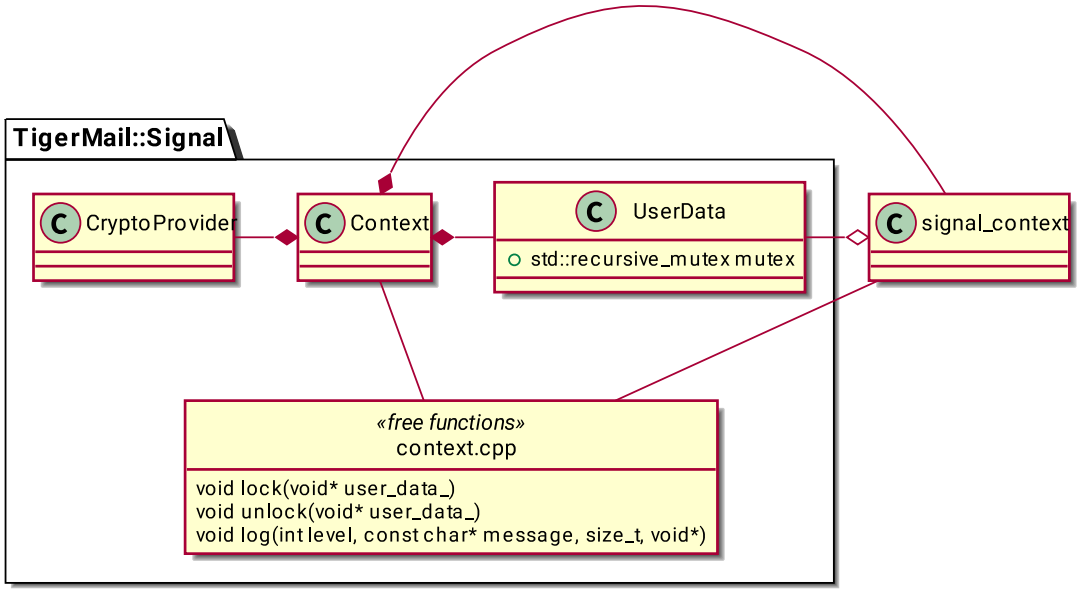
Listing 4.4: Przykładowe opakowanie funkcji z biblioteki `libsignal-protocol-c`

```
Ref<ratchet_chain_key> ratchet_chain_key_create(
    hkdf_context* kdf, BufferView key, uint32_t index, signal_context* context)
{
    auto chain_key = Ref<ratchet_chain_key>{};
    if (const auto result = ::ratchet_chain_key_create(
                    chain_key.get2(), kdf, key.data(), key.size(), index, context);
                    result < 0) {
        throw SignalError{"failed creating ratchet chain", result};
    }
    return chain_key;
}
```

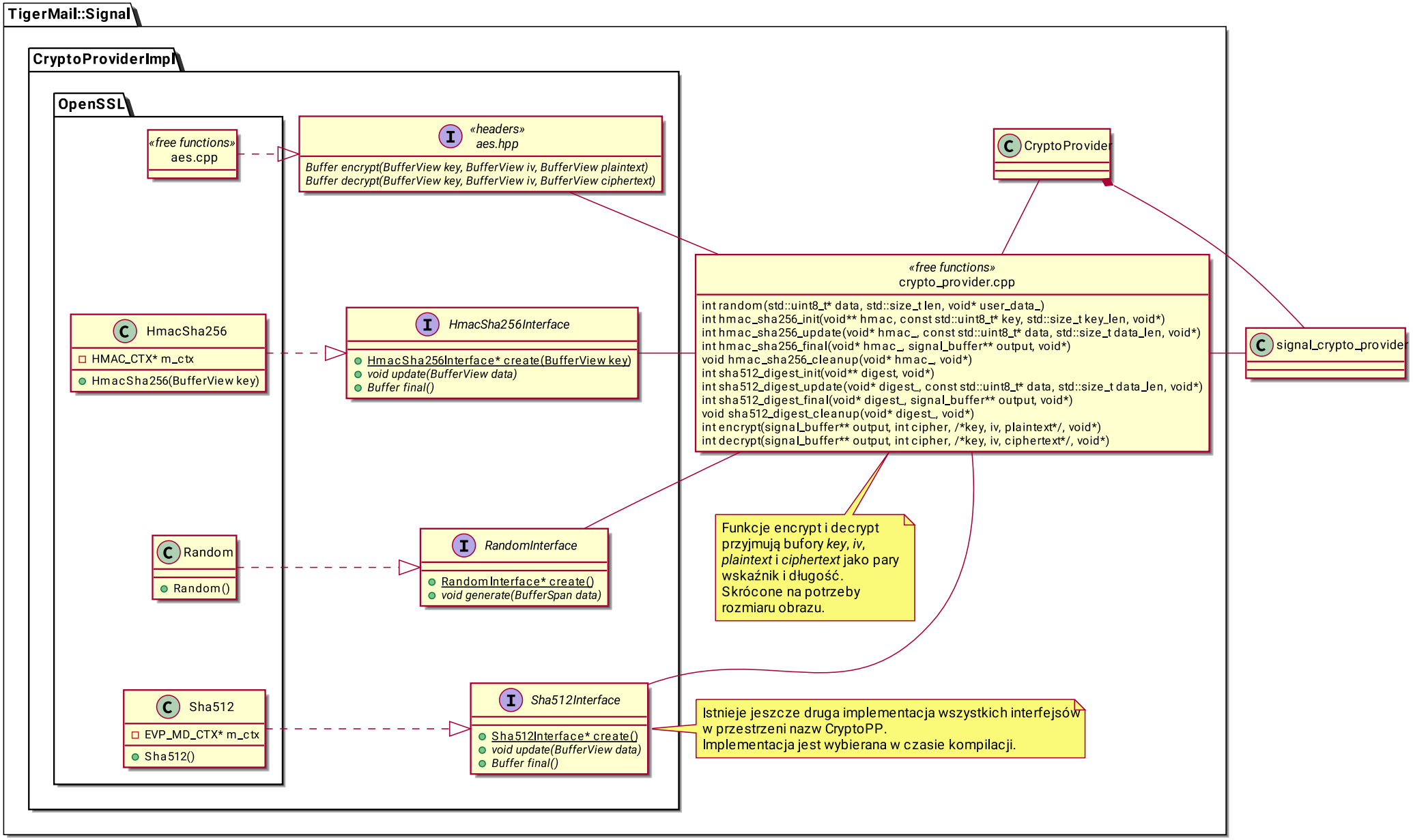
Kontekst

Biblioteka `libsignal-protocol-c` przyjmuje funkcje zaimplementowane przez użytkownika przez struktury nazwane `signal_context` i `signal_protocol_store_context`. Żeby z nich skorzystać, należy stworzyć obiekty tych struktur, dodać do nich wskaźniki do funkcji implementujących różne zadania i przekazywać wskaźniki do tych obiektów do funkcji z biblioteki.

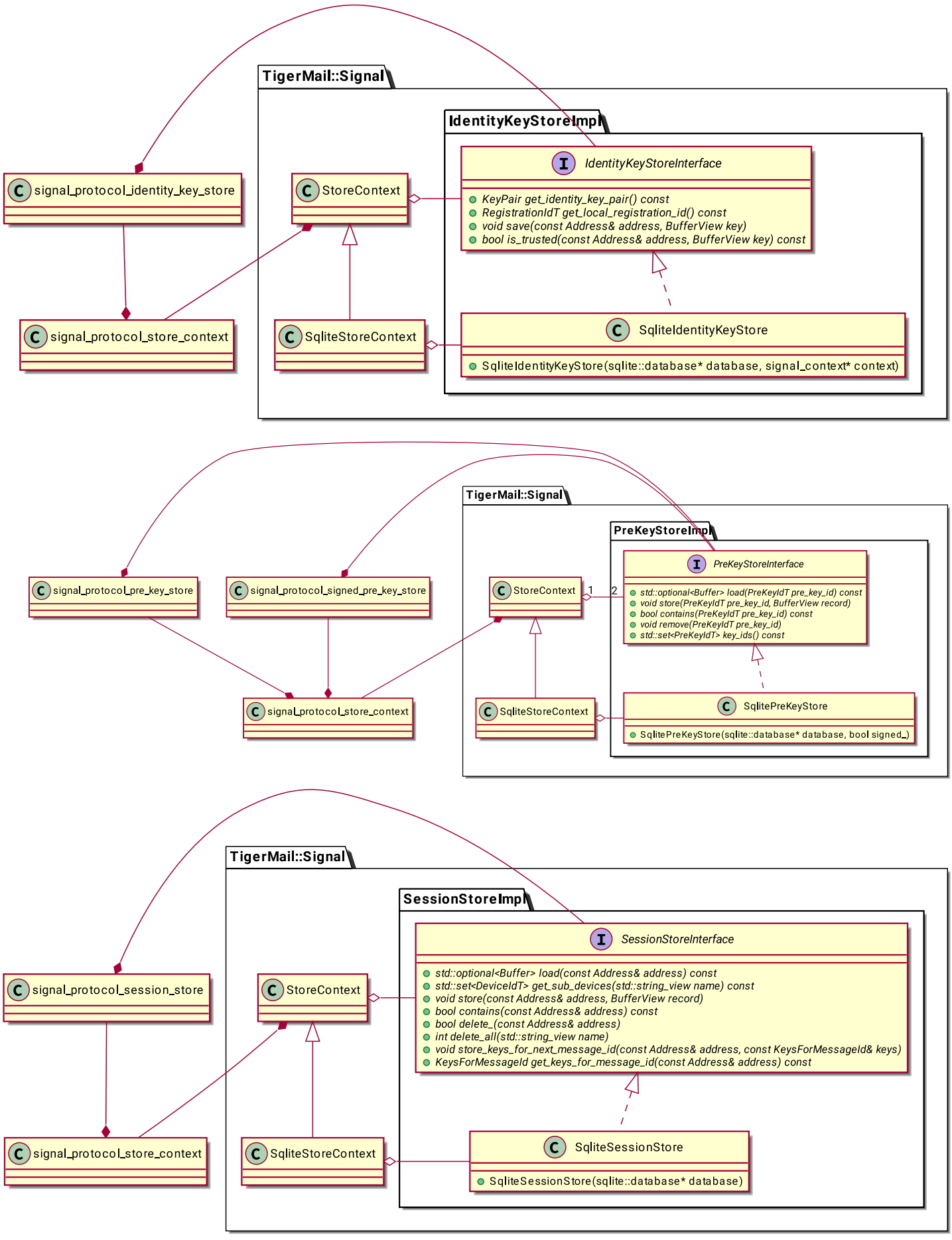
Struktura `signal_context` (rysunki 4.17 i 4.18) daje dostęp bibliotece `libsignal-protocol-c` do globalnego mutexu, funkcji do logowania komunikatów i funkcji kryptograficznych.



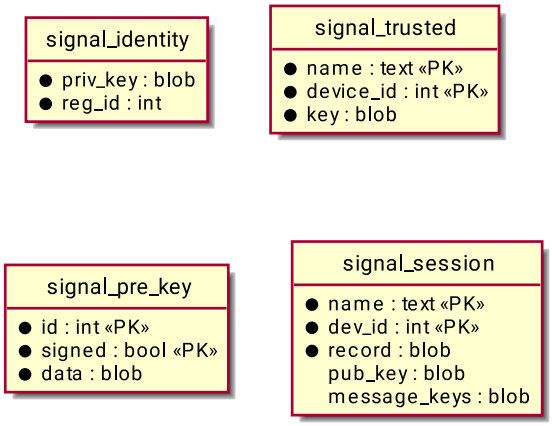
Rysunek 4.17: Diagram klas kontekstu *Signal*



Rysunek 4.18: Diagram klas kontekstu *Signal*



Rysunek 4.19: Diagram klas kontekstu *Signal*



Rysunek 4.20: Diagram encji bazy danych *Signal*

Struktura `signal_protocol_store_context` (rysunek 4.19) daje dostęp do przechowywania danych takich jak wygenerowane klucze, zaufani rozmówcy i istniejące sesje. Struktura bazy danych (rysunek 4.20) jest bardzo prosta, ponieważ tabele nie zawierają żadnych kluczy obcych. Pola `pub_key` i `message_keys` w tabeli `signal_session` są dodane na potrzeby aplikacji *TigerMail*.

Serializacja kluczy wstępnych

Biblioteka `libsignal-protocol-c` wymaga od użytkownika implementacji mechanizmu dystrybucji kluczy wstępnych. *API* tej biblioteki dostarcza jedynie metody do stworzenia lokalnego obiektu klasy `session_pre_key_bundle` i do stworzenia z jego pomocą sesji.

Jak napisałem w rozdziale 4.7.1, do serializacji kluczy wybrałem bibliotekę *protobuf*. Pozwala ona na wygenerowanie kodu do serializacji struktury, opisanej w specjalnym języku.

W listingu 4.5 znajduje się kod wykorzystywany przez bibliotekę *protobuf*, a w listingu 4.6 znajduje się powiązana z tym struktura w *C++*.

Listing 4.5: Struktura `KeyBundle` używana w *protobuf*

```
syntax = "proto3";
option optimize_for = LITE_RUNTIME;

package TigerMail.Signal.KeyBundleImpl;

message KeyBundle {
    message PreKey {
        uint32 id = 1;
        bytes pub = 2;
    }

    uint32 registration_id = 1;
    uint32 device_id = 2;
    bytes identity_key_pub = 3;
    PreKey signed_pre_key = 4;
    bytes signed_pre_key_signature = 5;
    repeated PreKey pre_keys = 6;
}
```

Listing 4.6: Struktura `KeyBundle` używana w *C++*

```


struct KeyBundle {
    struct PreKey {
        PreKeyIdT id;
        Ref<ec_public_key> key;
    };

    RegistrationIdT registration_id;
    std::uint32_t device_id;
    Ref<ec_public_key> identity_key;
    PreKey signed_pre_key;
    Buffer signed_pre_key_signature;
    std::vector<PreKey> pre_keys;
};
```

W podkatalogu `key_bundle_impl` znajdują się powyższy kod i wygenerowany na jego podstawie kod w języku *C++*. W plikach `key_bundle.*` znajdują się funkcje, które opakowują serializację struktury `KeyBundle`.

4.7.8 Główna logika aplikacji TigerMail

Główna logika aplikacji znajduje się w podkatalogu `tigermail` i wykorzystuje przestrzeń nazw `TigerMail`.

Najważniejszym elementem aplikacji są klasa `Client`, znajdująca się w plikach `client.*` i funkcje w pliku `main.cpp`.

Klasa `Client`

Klasa `Client` jest odpowiedzialna za kryprografię i komunikację z klientami *geth* i *swarm*. Na tym poziomie treścią wiadomości może być dowolny tekst, w szczególności nie musi to być wiadomość w formacie *IMF*. Dzięki temu możliwe jest ponowne wykorzystanie tego kodu, do przesyłania wiadomości w innym standardzie. Udostępnia trzy metody, wymienione w listingu 4.7.

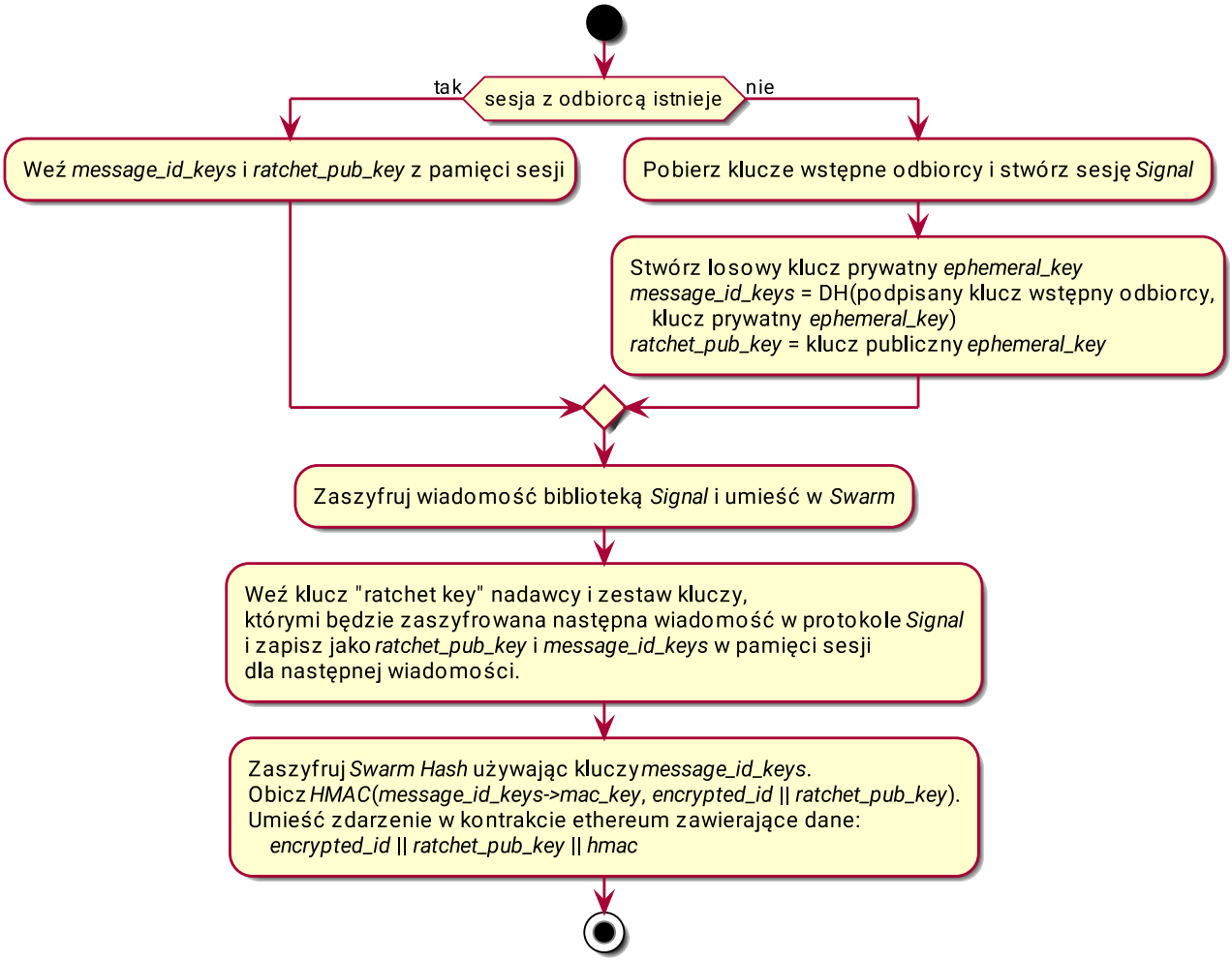
Listing 4.7: Metody klasy `Client`

```
class Client {
    [...]
    void rotate_keys(std::any yield);
    void send_message(const Ethereum::Address& to, BufferView content, std::any
        yield);
    std::vector<Message> receive_messages(std::any yield);
}
```

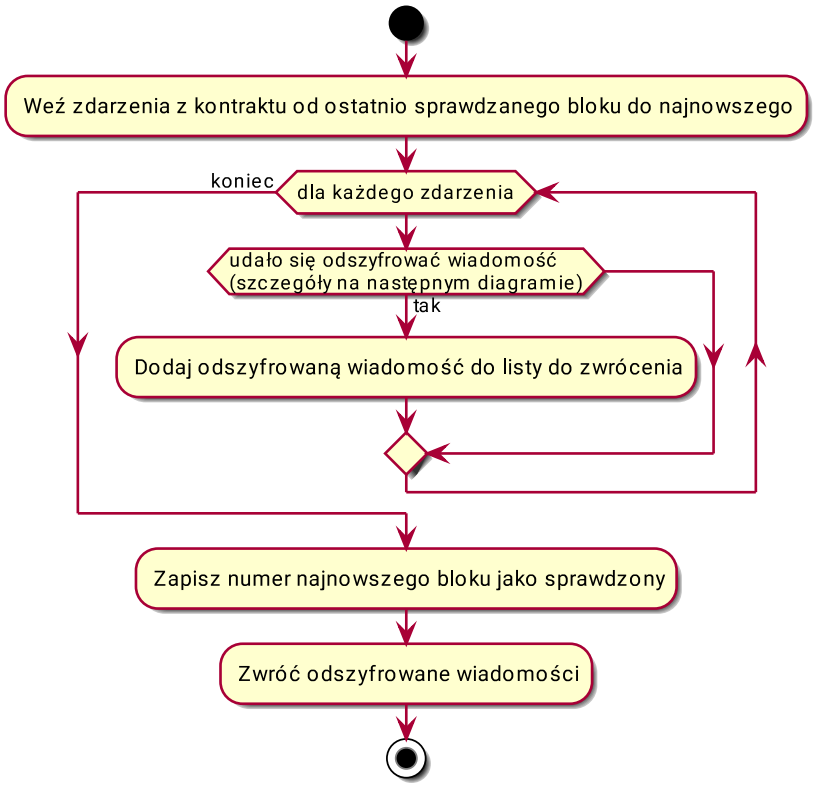
Metoda `rotate_keys` składa się z trzech kroków: generowanie nowych kluczy, publikowanie kluczy i usuwanie z pamięci starych kluczy. Trzeci krok usuwa klucze tak, żeby pamiętane były dwie generacje (obecna i poprzednia). Dzięki temu program może odbierać wiadomości, które ktoś mógłby wysłać, zanim nowe klucze zostaną umieszczone w sieci *Swarm*.

Metoda `send_message` wysyła wiadomość zgodnie z opisem w rozdziale 4.4. Diagram aktywności znajduje się na rysunku 4.21. Ponieważ biblioteka `libsignal-protocol-c` po zaszyfrowaniu wiadomości usuwa klucz, którym została zaszyfrowana, trzeba ten klucz zdobyć przed zaszyfrowaniem wiadomości, żeby móc zaszyfrować nim *Swarm Hash*. Dlatego po zaszyfrowaniu wiadomości klucze są zapisywane w danych sesji, żeby były gotowe do użycia przy wysyłaniu następnej wiadomości.

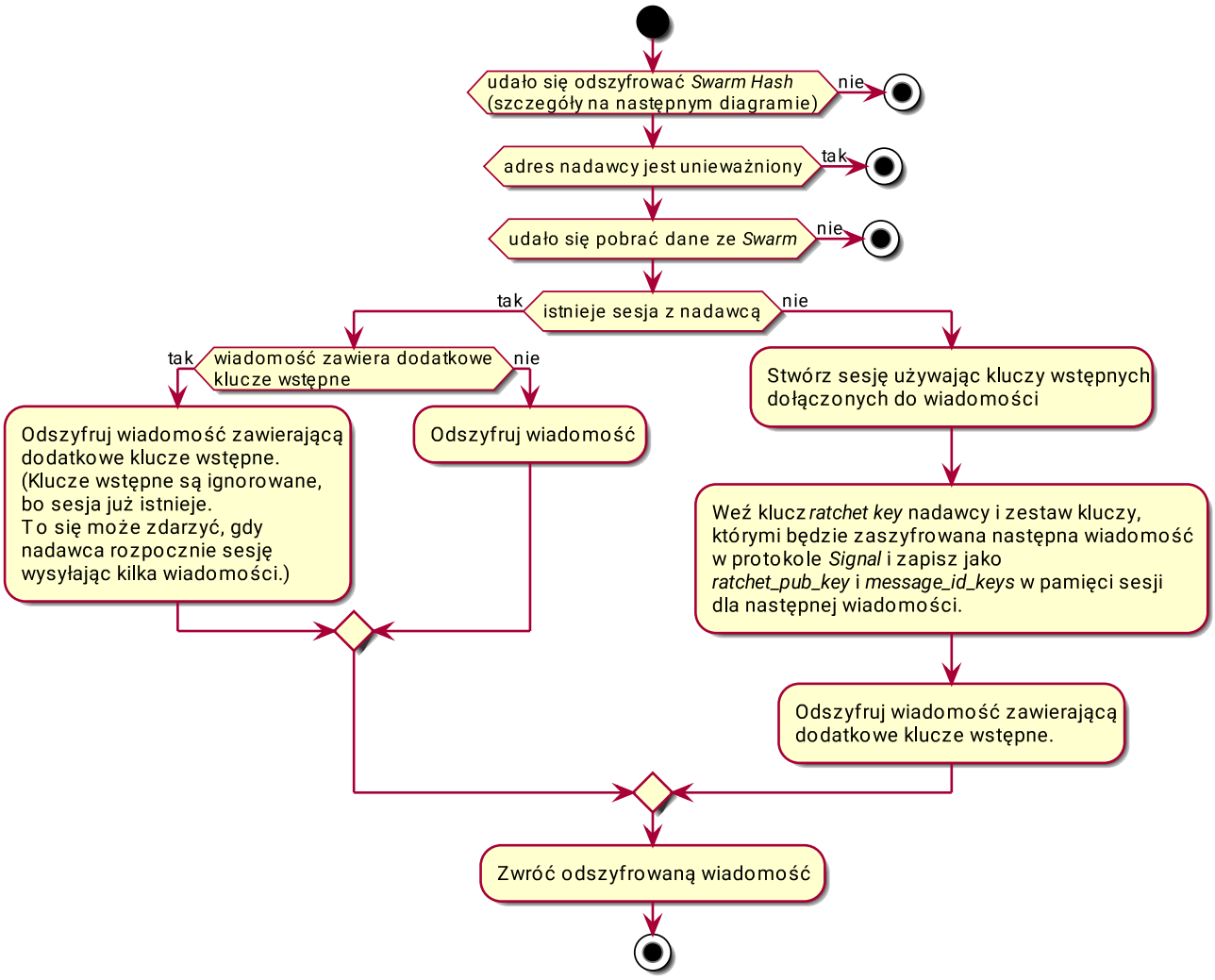
Metoda `receive_messages` odbiera wiadomość zgodnie z opisem w rozdziale 4.4. Diagram aktywności znajduje się na rysunkach 4.22, 4.23 i 4.24. Duża złożoność operacji odbierania wiadomości wynika ze szczegółowego sprawdzania poprawności otrzymanych danych oraz z obecności mechanizmu pomijania zagubionych wiadomości. Dodatkowo pierwsza wiadomość rozpoczynająca sesję wymaga specjalnej obsługi.



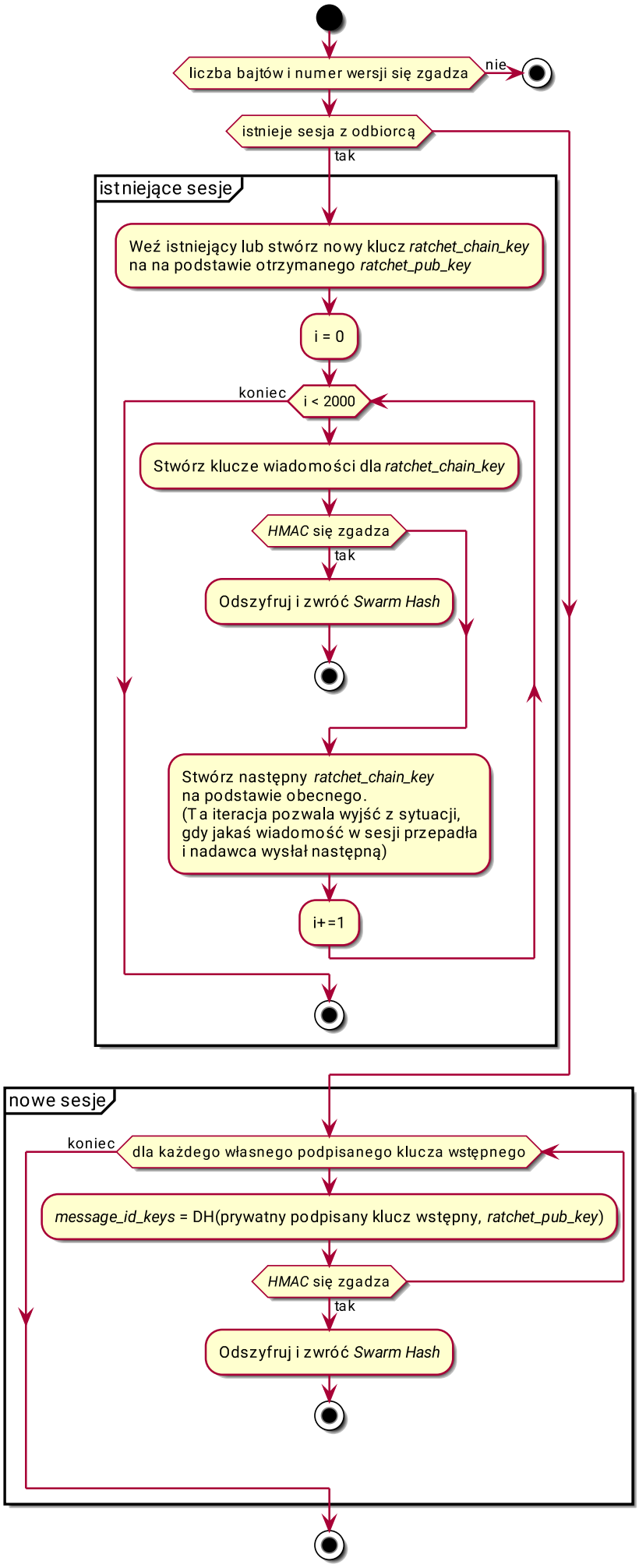
Rysunek 4.21: Diagram aktywności metody `send_message`.



Rysunek 4.22: Diagram aktywności metody `receive_messages`.



Rysunek 4.23: Diagram aktywności odszyfrowania jednej wiadomości.



Rysunek 4.24: Diagram aktywności odszyfrowania *Swarm Hash*.

Plik `main.cpp`

W pliku `main.cpp` znajdują się elementy spajające główne moduły ze sobą oraz funkcje uruchamiane w interwałach czasu.

Funkcje `first_rotate_keys` i `rotate_keys` odpowiedzialne są za wywoływanie metody `rotate_keys` klasy `Client` opisanej powyżej. W domyślnej konfiguracji interwałem jest 14 dni, a czas ostatniego wywołania jest zapisywany w bazie danych i sprawdzany przy uruchomieniu aplikacji.

Funkcja `smtp_message_received` obsługuje wiadomości odebrane z serwera *SMTP* i przekazuje je do metody `send_message` klasy `Client`.

Funkcja `fetch_new_messages` wywołuje domyślnie co 15 minut metodę `receive_messages` klasy `Client`. Odebrane wiadomości są dodawane do obiektu klasy `MessagesStore` dla serwera *POP3*.

W powyższych funkcjach wszelkie błędy są zapisywane do logu oraz dodawane jako wiadomości do `MessagesStore` z adresem nadawcy *noreply@tigermail.eth*.

4.8 Testy

W tym rozdziale znajduje się opis przygotowanych i przeprowadzonych przeze mnie testów.

Testy jednostkowe

Testy jednostkowe pokrywają znaczną część kodu. Według narzędzia *lcov*: 83,4% linii, 85,9% funkcji i 46,5% rozgałęzień.

Szczególnie warte uwagi są te, które znajdują się w pliku `tigermail/test/client/client.cpp`. Wykorzystując sztuczne aplikacje klienckie sieci *Ethereum* i *Swarm*, zaimplementowane jako obiekty wewnątrz programu testowego, testowane są następujące scenariusze:

`simple_message_exchange_with_known_answer` – Prosta wymiana wiadomości pomiędzy Alicją i Bobem.

`multiple_starting_messages_without_reply` – Alicja rozpoczyna konwersację, wysyłając kilka wiadomości po kolei. To jest istotne, ponieważ protokół *Signal* dołącza do tych wiadomości dodatkowe klucze i trzeba poprawnie obsłużyć sytuację, gdy jest tych wiadomości więcej niż jedna.

`invalidation` – Alicja wysyła pierwszą wiadomość do Boba, unieważnia swój klucz i wysyła drugą wiadomość. Bob odczytuje pierwszą, ale drugą już odrzuca.

`lost_first_message_id` **i** `lost_first_message_content` – te testy symulują utratę pierwszej wiadomości na skutek błędu w *Ethereum* lub *Swarm*. Obecnie ten scenariusz nie jest poprawnie obsługiwany i testy są domyślnie wyłączone.

`lost_message_id_in_conversation`**, i** `lost_message_content_in_conversation` – Te testy, podobnie jak powyższe, symulują utratę wiadomości, ale w sesji, gdzie wiadomości już wcześniej zostały wymienione. Aplikacja poprawnie generuje klucze do odszyfrowania zdarzeń z *Ethereum*, mimo utraconych wiadomości.

`failed_to_get_transaction_by_hash` – W czasie testowania aplikacji z klientem *geth* czasami się zdarzało, że po odebraniu powiadomień o nowych zdarzeniach, w odpowiedzi na zapytanie o konkretną transakcję, *geth* zwracał *null*. Ten test symuluje taką sytuację i sprawdza, czy wiadomości są odbierane przy ponownej próbie.

Oprócz tego każdy moduł (`common`, `ethereum`, `json_rpc`, `pop3_smtp_common`, `pop3`, `signal`, `smtp`, `swarm`) zawiera swój podkatalog `test` z testami jednostkowymi.

Testy bezpieczeństwa – fuzzing

Fuzzing jest metodą pozwalającą na wykrywanie błędów w przetwarzaniu danych wejściowych. Wykorzystałem do tego narzędzia *American fuzzy lop*[3] i *libFuzzer*[4].

Do ich użycia potrzebne jest napisanie funkcji, która przyjmuje ciąg bajtów i przetwarza jak dane wejściowe. Program musi wykonywać się zawsze tak samo dla tych samych danych wejściowych, więc generator liczb losowych jest zastąpiony deterministycznym generatorem pseudolosowym zainicjowanym ziarnem o stałej wartości. Sprawdzanie sum kontrolnych jest wyłączone, żeby badany program mógł częściej wykonywać dalsze instrukcje.

Narzędzia uruchamiają taką funkcję wielokrotnie, badając przebieg wykonania programu i szukają nowych danych, które powodują nowe przebiegi. W ten sposób mogą być znajdowane błędy dostępu do pamięci lub przekroczenie czasu wykonania.

Testy znajdują się w katalogu `fuzz`. Każdy z modułów odpowiedzialnych za komunikację sieciową ma swoje testy w osobnym podkatalogu (`ethereum_jsonrpc`, `ethereum_transport_protocol`,

[3] https://lcamtuf.coredump.cx/afl/
[4] https://llvm.org/docs/LibFuzzer.html

`pop3`, `smtp`, `swarm_ipc_client`, `swarm_tcp_client`). W podkatalogu `tigermail` znajduje się test dla głównej logiki aplikacji. W pliku `README.md` znajduje się instrukcja uruchamiania testów.

Testy bezpieczeństwa – Known Answer Test (KAT)

Testy typu *KAT* pozwalają sprawdzić, czy aplikacja poprawnie wykonuje operacje kryptograficzne przez porównanie wyników z wyliczonymi wcześniej wartościami. Do tych testów, tak samo jak dla *fuzzingu*, generator liczb losowych jest zastąpiony deterministycznym. Te testy zaimplementowane są jako część testów jednostkowych.

Testowane w ten sposób są interfejsy do zewnętrznych bibliotek kryptograficznych (każdy test jednostkowy zawiera w nazwie `known_answer`):

- algorytmy *AES*, *HMAC-SHA256* i *SHA-512* w katalogu `signal/test/crypto_provider_impl/`,
- generator pseudolosowy w pliku `signal/test/crypto_provider_impl/random.cpp`,
- algorytm *keccak* w pliku `ethereum/test/crypto.cpp`,
- podpisywanie transakcji dla sieci *Ethereum* w pliku `ethereum/test/tigermail_contract.cpp`,

Test głównej logiki aplikacji znajduje się w pliku `tigermail/test/client/client.cpp` w teście `simple_message_exchange_with_known_answer`.

Test integracyjny: Przesłanie wiadomości przez sieci Ethereum i Swarm

Środowisko testowe

Dwa komputery („Alice” i „Bob”) podłączone do sieci przez różnych dostawców Internetu. Na każdym z nich:

- Klient *geth*, wersja 1.9.7, uruchomiony z opcjami `--goerli --syncmode light`.
- Klient *swarm*, wersja 0.5.4, uruchomiony z opcją `--lightnode`.
- Aplikacja *TigerMail*, uruchomiona z adresem kontraktu `0xD6B41d0888CAbDEe3E733B210553E2E361988383`.
- Klient pocztowy *Thunderbird*, skonfigurowany zgodnie z opisem w załączniku A.

Scenariusz

- *Bob* uruchomiony w celu wygenerowania i opublikowania kluczy wstępnych.
- *Bob* wyłączony.
- *Alice* uruchomiony.
- *Alice* wysyła wiadomość do *Bob*.
- *Alice* wyłączony.
- *Bob* uruchomiony.
- *Bob* odbiera wiadomość od *Alice*.
- *Bob* wysyła odpowiedź do *Alice*.
- *Bob* wyłączony.
- *Alice* uruchomiony.
- *Alice* odbiera odpowiedź od *Bob*.

Wyniki

Test przeszedł poprawnie. Szczegółowe logi i zrzut przesłanych danych znajdują się w załączonych plikach, w katalogu `testy/1`.

W czasie implementacji aplikacji, przy wcześniejszych próbach wykonania tego testu, napotkałem następujące problemy:

- Czasami transakcje wysłane do *geth* nie są publikowane w sieci. Niestety z poziomu aplikacji *TigerMail* nie da się zweryfikować, czy taka sytuacji występuje, ale można to sprawdzić na przykład przez serwisy internetowe pozwalające na przeglądanie danych w sieci *Ethereum* (na przykład `https://etherscan.io/`), próbując otworzyć dane o wysłanej transakcji. Żeby to naprawić, trzeba uruchomić ponownie klienta *geth*. Jest to problemem tylko przy przesłaniu pierwszej wiadomości w konwersacji (szczegóły opisałem powyżej, przy testach jednostkowych).
- Raz zdarzyło się, że publikacja kluczy wstępnych zakończyła się błędem. Przy ponownym uruchomieniu aplikacji, operacja się powiodła i aplikacja działała dalej poprawnie.

Wnioski

Udało się zrealizować system pocztowy zgodny z założeniami wymienionymi w rozdziale 4.1.

Test wydajności: Odporność aplikacji na masowe przysyłanie wiadomości

Jednym z problemów wykorzystania protokołu *Signal* jest to, że dla każdej rozmowy musi być zapamiętany stan sesji. Jeżeli Mallory stworzyłby dużo adresów i z każdego wysłał wiadomość na adres Alicji, to mogłoby się zdarzyć, że przechowywanie dużej ilości sesji przez Alicję spowolniłoby działanie jej aplikacji.

W tym teście sprawdzę, jak aplikacja zachowa się po odebraniu 100'000 wiadomości z unikalnych adresów.

Środowisko testowe

Ten test sprawdza działanie biblioteki *Signal*, więc zaimplementowałem go jako jeden z testów jednostkowych (znajduje się w pliku `tigermail/test/client/client_stress_test.cpp`), który jest domyślnie wyłączony ze względu na czasochłonność. Zamiast klientów sieci *Ethereum* i *Swarm* wykorzystałem implementacje biblioteczne.

- Procesor: *Intel i7-6500U*[5] – 2,50GHz, Turbo 3,10GHz, *Cache* 4MiB.
- RAM: *GoodRam GR2133S464L15/16G* – DDR4-2133 (PC4-17000), CL15, 16GB.
- Dysk: *Intel SSD 545s Series*[6] – odczyt 550MB/s, zapis 440MB/s.
- System plików: BTRFS

Scenariusz

- Stwórz klienta Alicji i opublikuj klucze wstępne.
- 100'000 razy:
  - Stwórz nową tożsamość.
  - Wyślij wiadomość do Alicji.
- Alicja odbiera wiadomości.

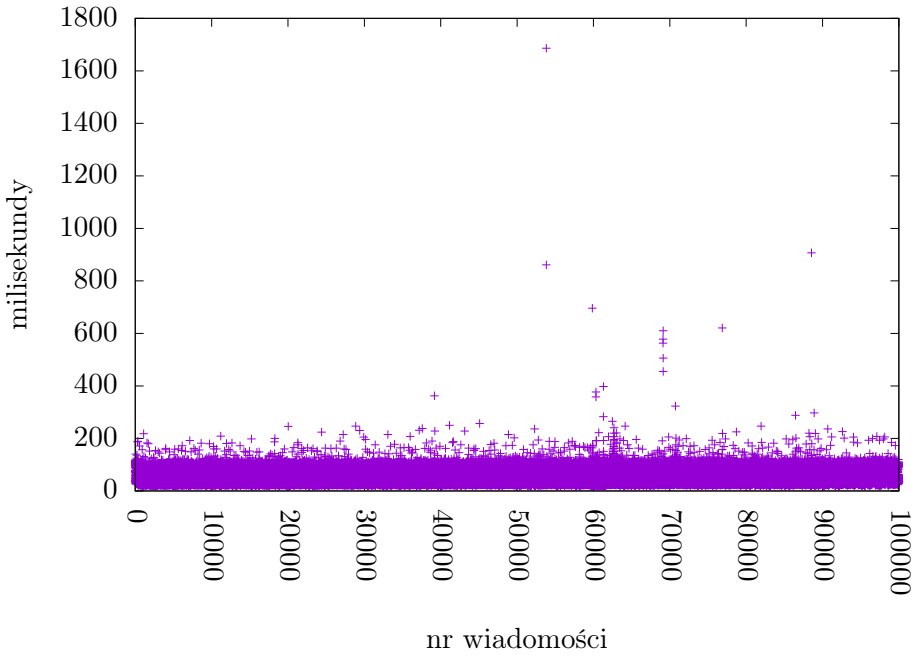
[5] https://ark.intel.com/content/www/us/en/ark/products/88194/intel-core-i7-6500u-processor-4m-cache-up-to-3-10-ghz.html

[6] https://ark.intel.com/content/www/us/en/ark/products/125018/intel-ssd-545s-series-128gb-2-5in-sata-6gb-s-3d2-tlc.html

Wyniki

W przeprowadzonym teście czas przetwarzania jednej przychodzącej wiadomości wyniósł średnio 61 milisekund. Po zakończeniu testu baza danych zajmowała 75MB miejsca na dysku. Rysunek 4.25 (stworzony na podstawie danych zawartych w pliku `testy/2/millis.txt`) przedstawia czasy przetwarzania kolejnych wiadomości.

W czasie trwania testu procesor był obciążony w około 50% na jednym rdzeniu. Głównym czynnikiem ograniczającym wydajność był dostęp do dysku, na którym jest zapisany plik z bazą danych *sqlite*. Na wykresie widać, że poza pojedynczymi przypadkami, spowodowanymi prawdopodobnie przez działalność innych programów, czas przetwarzania wiadomości nie zmienia się.



Rysunek 4.25: Czas przetwarzania wiadomości przychodzących.

Wnioski

Przy przepustowości sieci *Ethereum* 15 transakcji na sekundę [30] i zakładając, że nikt inny nie korzysta z sieci, wysłanie takiej ilości wiadomości zajęłoby godzinę i 55 minut.

Jedna transakcja publikująca zaszyfrowany *Swarm Hash* w sieci *Ethereum* potrzebuje 32798 *Gasu*. Przyjmując niewygórowaną cenę 5 *Gwei* za 1 *Gas*, opublikowanie tyle *Swarm Hash* kosztowałoby $16,399$ *Etheru*. Przy obecnym kursie wynoszącym około 184 USD za 1 *Ether*, kosztowałoby to 3017,42 dolarów.

W powyższych wyliczeniach pomijam takie kwestie jak wzrost ceny *Gasu* przy wzmożonym zapotrzebowaniu (cena czasami wzrasta do kilkudziesięciu *Gwei*) czy ewentualne opłaty za umieszczanie danych w *Swarm*.

Podsumowując, taki atak jest nieskuteczny i nieopłacalny.

Rozdział 5

Podsumowanie

Głównym moim celem było stworzenie bezpiecznego systemu asynchronicznej wymiany wiadomości, nazwanego przeze mnie *TigerMail*, z wykorzystaniem sieci rozproszonej i zapewniającego *forward secrecy*.

W tym celu zacząłem od przedstawienia architektury i protokołów wykorzystywanych w poczcie elektronicznej. Omówiłem zagrożenia i stworzone rozszerzenia mające im przeciwdziałać.

Opisałem również alternatywne systemy asynchronicznej wymiany wiadomości, ze szczególnym uwzględnieniem zastosowanych mechanizmów szyfrowania i ochrony prywatności.

Stworzyłem specyfikację wymagań, koncepcję i architekturę systemu *TigerMail*, w języku naturalnym i *UML*. Zaimplementowałem aplikację w języku *C++* dla systemu *GNU/Linux*. Przeprowadziłem testy opracowanego rozwiązania, w tym testy: funkcjonalne, bezpieczeństwa i wydajnościowe.

Wnioski

Najtrudniejszym problemem, który rozwiązałem w mojej pracy, było połączenie komunikacji asynchronicznej z siecią rozproszoną. Bardzo podobne rozwiązanie oferuje aplikacja *LemonMail* (rozdział 3.6), jednak wykorzystany w niej system plików *IPFS* powoduje, że do pełnej asynchroniczności potrzebna jest pomoc dodatkowego węzła w sieci, który przechowuje wysłane wiadomości.

Dodatkowym atutem mojego rozwiązania jest wykorzystanie biblioteki *Signal* do zapewnienia poufności przez szyfrowanie z *forward secrecy*. Z porównanych przeze mnie narzędzi jedynie *Freemail* (rozdział 3.4) zawiera takie zabezpieczenie, ale w niepełnej formie: żeby pobrać poprzednią wiadomość z sieci, trzeba wyliczyć jej adres przez odwrócenie funkcji skrótu *SHA-256*, ale wiadomości są szyfrowane tym samym kluczem.

Testując moją aplikację, pokazałem, że nadaje się do użycia w Internecie z wykorzystaniem publicznych sieci *Ethereum* i *Swarm*. Może z powodzeniem konkurować z opisanymi przeze mnie alternatywami.

Zdaję sobie sprawę z tego, że moja aplikacja nie zastąpi obecnie istniejącej poczty elektronicznej. Założenia, jakie przyjąłem, nie są tym, czego zwykły użytkownik oczekuje. To znaczy, jest on raczej zainteresowany tym, żeby jego wiadomości były przechowywane na serwerze, dostępne przez interfejs *www*, synchronizowane z telefonem i zabezpieczone jednym hasłem. Uważam, że dobrym kierunkiem rozwoju i realną szansą na poprawienie bezpieczeństwa powszechnie używanej poczty jest projekt *DarkMail* (rozdział 3.1).

Dalsze prace

Aplikacja *TigerMail* może być dalej rozszerzona o następujące funkcje:

- Dostęp z wielu urządzeń do jednego konta. Protokół *Signal* obsługuje taką możliwość. Używanie tego samego klucza *Ethereum* na wielu urządzeniach byłoby niebezpieczne, ponieważ zwiększa to szansę na wykradzenie klucza. Lepszym rozwiązaniem byłoby, gdyby każde urządzenie miało własny klucz, powiązany z głównym kontem w kontrakcie na *Ethereum*.
- Obsługa nazw zarejestrowanych przez *Ethereum Name Service* (ENS)[1] jako adresów pocztowych.
- Publiczne i zamknięte listy dyskusyjne. Prawdopodobnie w tym przypadku do przechowywania wiadomości lepszy byłby *IPFS*.
- Zapewnianie dostępności danych w *Swarm*.

[1] https://ens.domains/

Słownik terminów

ARC *Authenticated Received Chain* – Metoda przeciwdziałania *email spoofing*. Opis w rozdziale 2.4. Strony: 13, 27, 28

blockchain – Struktura danych, w której dane są grupowane w blokach, a każdy blok zawiera sumę kontrolną poprzedniego bloku, lub oprogramowanie implementujące tę strukturę danych wraz z algorytmem dochodzenia do konsensusu w sieci na temat globalnego stanu. Strony: 3, 4, 6, 10–13, 51, 52, 56, 64, 75, 76, 82, 109

CMS *Cryptographic Message Syntax* – Format zapisu wiadomości kryptograficznych, używany m.in. przez *S/MIME*. Strony: 30, 40

CRL *Certificate Revocation Lists* – Lista unieważnionych certyfikatów *X.509*, podpisana przez urząd certyfikacji. Strony: 30–32

DIME *Dark Internet Mail Environment* – System pocztowy zaprojektowany jako modyfikacja protokołów *SMTP* i *IMAP*, dodający obowiązkowe szyfrowanie i metody zarządzania kluczami. Opis w rozdziale 3.1. Strony: 11–13, 42, 44, 53

DKIM *DomainKeys Identified Mail* – Metoda przeciwdziałania *email spoofing*. Opis w rozdziale 2.4. Strony: 3, 4, 13, 26–28

DMARC *Domain-based Message Authentication, Reporting, and Conformance* – Metoda przeciwdziałania *email spoofing*. Opis w rozdziale 2.4. Strony: 3, 4, 13, 27–29

DNS *Domain Name System* – Hierarchiczna, zdecentralizowana baza danych zawierająca rekordy typu klucz-wartość. Najpopularniejsze wykorzystanie to mapowanie nazwy domeny na adres *IP*. Strony: 24–27, 43, 108, 110, 111

DNSSEC *Domain Name System Security Extensions* – Rozszerzenie protokołu *DNS* pozwalające klientowi na weryfikację autentyczności i integralności otrzymanych danych. Strony: 25, 43, 44

email spoofing – Wysyłanie wiadomości poczty elektronicznej z fałszywym adresem nadawcy. Strony: 9, 12, 20, 25, 53, 72, 108, 111

Ether – Waluta używana w sieci *Ethereum* służąca do płacenia za wykonane transakcje. Strony: 57, 105, 109

Ethereum – *Blockchain* pozwalający na programowanie własnych kontraktów i przechowywanie dowolnych danych. Opis w rozdziale 4.2.1. Strony: 3, 4, 6, 11–14, 51, 52, 56–59, 64–68, 70, 72, 74–78, 80–82, 99–103, 105, 107, 109, 111, 119

forward secrecy – Metoda szyfrowania serii wiadomości, która w przypadku kompromitacji klucza rozmówcy lub ujawnienia jednej wiadomości, zachowuje poufność poprzednich. Strony: 3, 4, 12, 13, 39, 49, 53, 56, 59–61, 67, 78, 106, 107, 110

Gas – Jednostka miary kosztu operacji w maszynie wirtualnej *Ethereum*. Opis w rozdziale 4.2.1. Strony: 57, 105

Gwei – 10^{-9} *Ether*. Strona: 105

HTTP *Hypertext Transfer Protocol* – Protokół sieciowy wykorzystywany głównie do interackji ze stronami *WWW*. Strony: 23, 32, 67, 80, 109

HTTPS *HTTP Secure* – Protokół *HTTP* z obowiązkowym zabezpieczeniem przez *TLS*. Strony: 24, 25

IMAP *Internet Message Access Protocol* – Protokół pozwalający na synchronizację zawartości skrzynki pocztowej między serwerem i wieloma klientami. Opis w rozdziale 2.1. Strony: 11, 13, 16, 18, 22–24, 44, 67, 108, 110, 111

IMF *Internet Message Format* – Format zapisu wiadomości poczty elektronicznej, pozwalający dodawać nagłówki do treści. Opis w rozdziale 2.2. Strony: 15, 17–20, 25, 26, 95

IPFS *InterPlanetary File System* – Rozproszony system plików. Opis w rozdziale 3.6. Strony: 11, 51–53, 59, 106, 107

Kademlia DHT (*Distributed Hash Table*) – Algorytm pozwalający przechowywać dane typu klucz-wartość w sieci rozproszonej. Strony: 11, 50, 58

komunikacja asynchroniczna – Metoda komunikacji niewymagająca jednoczesnej interakcji obu stron, na przykład wysyłanie listów. [37, 83] Strony: 10, 12, 13, 15, 42, 52, 53, 55, 67, 106

komunikacja synchroniczna – Metoda komunikacji wymagająca jednoczesnej interakcji obu stron, na przykład rozmowa telefoniczna. Strona: 10

MUA *Mail User Agent* – Program pocztowy służący do odbierania i wysyłania poczty, np. *Thunderbird*. Strony: 16, 17, 19, 23, 30, 38, 66, 67, 70, 72, 75, 78

OpenPGP – Protokół zarządzania kluczami asymetrycznymi, oparty o sieć zaufania. Opis w rozdziale 2.5. Strony: 3, 4, 9, 11, 13, 33, 34, 36, 39, 46, 56

POP3 *Post Office Protocol* – Protokół sieciowy pozwalający pobierać oczekujące wiadomości z serwera pocztowego. Opis w rozdziale 2.1. Strony: 11, 13, 16, 18, 21, 22, 24, 66, 67, 72, 75, 80, 83–86, 99, 110, 111, 119

rekord *MX* – wpis w bazie danych *DNS* zawierająca adres *IP* serwera *MTA* (opis w rozdziale 2.1) dla danej domeny. Strony: 24, 26

rekord *TXT* – wpis w bazie danych *DNS*, który może zawierać dowolny tekst. Jedna domena może zawierać wiele rekordów tego typu. Strony: 25–28, 43

rozproszony system plików – Sieć rozproszona pozwalająca na przechowywanie i współdzielenie danych. Strony: 3, 4, 6, 11–13, 46, 51, 58, 109, 111

RPC *Remote Procedure Call* – Nazwa dla protokołów umożliwiających wywoływane procedur i zwracanie wyników. Strony: 57, 67

S/MIME *Secure/Multipurpose Internet Mail Extensions* – Protokół zabezpieczania wiadomości pocztowych, wykorzystujący certyfikaty podobne do *TLS*. Opis w rozdziale 2.5. Strony: 3, 4, 9, 13, 30–32, 39, 108

SASL *Simple Authentication and Security Layer* – Protoków uwierzytelniania użytkowników, zaprojektowany jako element innych protokołów (na przykład *POP3* lub *IMAP*), pozwalający wykorzystywać wiele różnych metod. Opis w rozdziale 2.1. Strony: 13, 18, 20–22

sieć rozproszona – Sieć, w której użytkownicy łączą się za pośrednictwem innych użytkowników. Strony: 3, 4, 10–13, 45, 46, 50, 53, 55, 56, 64, 66, 67, 76, 106, 109–111

sieć zdecentralizowana – Sieć, w której użytkownicy mają konta na wybranych serwerach. Serwery należące do różnych organizacji przekazują informacje między sobą w imieniu swoich użytkowników. Strony: 10, 45, 55, 108

Signal – Komunikator internetowy (`signal.org`) lub protokół używany przez ten komunikator, zapewniający szyfrowanie z *forward secrecy*. Opis w rozdziale 4.2.3. Strony: 3, 4, 6, 12–14, 59, 64, 67, 70, 72, 75–79, 89–93, 99, 103, 107

SMTP *Simple Mail Transfer Protocol* – Protokół używany do wysyłania poczty elektronicznej. Strony: 3, 4, 11, 13, 16, 18–20, 23–25, 44, 66, 70, 75, 80, 86–88, 99, 108, 111

SMTP DANE *SMTP Security via Opportunistic* DNS*-Based Authentication of Named Entities* – Protokół chroniący przed pominięciem użycia *STARTTLS* w protokole *SMTP*. Opis w rozdziale 2.3. Strony: 13, 24, 25

SMTP STS *SMTP Strict Transport Security* – Protokół chroniący przed pominięciem użycia *STARTTLS* w protokole *SMTP*. Opis w rozdziale 2.3. Strony: 13, 25

SPF *Sender Policy Framework* – Metoda przeciwdziałania *email spoofing*. Opis w rozdziale 2.4. Strony: 3, 4, 13, 26–29

STARTTLS *Opportunistic TLS* – Rozszerzenie używane w protokołach, między innymi *SMTP*, *POP3* i *IMAP*, pozwalające na włączenie *TLS* w pierwotnie niezabezpieczonym połączeniu. Opis w rozdziale 2.3. Strony: 3, 4, 13, 20, 22–25, 111

Swarm – Rozproszony system plików, który razem z *Ethereum* ma służyć do budowania rozproszonych aplikacji. Opis w rozdziale 4.2.2. Strony: 3, 4, 6, 12, 13, 58, 59, 65–68, 70, 72, 76–79, 81, 95, 99, 100, 103, 105, 107, 111

Swarm Feed – Mechanizm w sieci *Swarm* pozwalający publikować dane zmienne, identyfikowane przez nadawcę i temat. Strony: 59, 64, 67, 68, 70, 72, 77, 78, 80, 82

Swarm Hash – Skrót kryptograficzny *Keccak 256* danych umieszczonych w *Swarm*, pozwalający pobrać te dane z sieci. Strony: 58, 64, 65, 67, 70, 72, 76, 78, 95, 98, 105

TLS *Transport Layer Security* – Protokół sieciowy służący jako warstwa zapewniająca szyfrowanie połączenia i uwierzytelnianie serwera. Opis w rozdziale 2.3. Strony: 9, 11, 13, 23–25, 30, 41, 43, 44, 46, 109–111

Unix Domain Socket – Mechanizm komunikacji międzyprocesowej. Używa się go tak samo, jak połączenia sieciowego, z tą różnicą, że gniazdo serwera identyfikowane jest przez plik. Strony: 15, 66, 67, 75, 82

Bibliografia

[1] *15 reasons not to start using PGP*. URL: https://secushare.org/PGP (strony 10, 39).

[2] Spencer Ackerman. "Lavabit email service abruptly shut down citing government interference". W: *The Guardian* (sierp. 2013). URL: https://www.theguardian.com/technology/2013/aug/08/lavabit-email-shut-down-edward-snowden (strona 42).

[3] *Against DNSSEC*. URL: https://sockpuppet.org/blog/2015/01/15/against-dnssec/ (strona 25).

[4] Kurt Andersen i in. *Authenticated Received Chain (ARC) Protocol*. Internet-Draft draft-ietf-dmarc-arc-protocol-23. Work in Progress. Internet Engineering Task Force, list. 2018. 39 s. URL: https://datatracker.ietf.org/doc/html/draft-ietf-dmarc-arc-protocol-23 (strona 27).

[5] *Autocrypt*. URL: https://autocrypt.org/ (strona 38).

[6] Daniel J. Bernstein i in. *Ed25519: high-speed high-security signatures*. URL: https://ed25519.cr.yp.to/ (strona 43).

[7] Abhay Bhushan i in. *Standardizing Network Mail Headers*. RFC 561. Wrz. 1973. DOI: 10.17487/RFC0561. URL: https://rfc-editor.org/rfc/rfc561.txt (strona 9).

[8] Bitcoin wikipedia contributors. *Base58Check encoding*. 2017. URL: https://en.bitcoin.it/w/index.php?title=Base58Check_encoding&oldid=64289 (strona 45).

[9] Bitcoin wikipedia contributors. *Proof of work*. 2016. URL: https://en.bitcoin.it/w/index.php?title=Proof_of_work&oldid=60949 (strona 45).

[10] Bitcoin wikipedia contributors. *Secp256k1*. 2018. URL: https://en.bitcoin.it/w/index.php?title=Secp256k1&oldid=65600 (strony 43, 45).

[11] Bitmessage wikipedia contributors. *Decentralized Mailing List*. 2018. URL: https://bitmessage.org/w/index.php?title=Protocol_specification&oldid=47591 (strona 45).

[12] Bitmessage wikipedia contributors. *Protocol specification*. 2018. URL: https://bitmessage.org/w/index.php?title=Protocol_specification&oldid=47591 (strona 45).

[13] Sharon Boeyen i in. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. Maj 2008. DOI: 10.17487/RFC5280. URL: https://rfc-editor.org/rfc/rfc5280.txt (strony 30, 32).

[14] Jeremy Clark i in. "Securing Email". W: *CoRR* abs/1804.07706 (2018). arXiv: 1804.07706. URL: http://arxiv.org/abs/1804.07706 (strony 8, 11, 17).

[15] Katriel Cohn-Gordon i in. *A Formal Security Analysis of the Signal Messaging Protocol*. Cryptology ePrint Archive, Report 2016/1013. 2016. URL: https://eprint.iacr.org/2016/1013 (strony 59, 77).

[16] Lutz Donnerhacke, Jon Callas i David Shaw. *OpenPGP Message Format*. RFC 4880. List. 2007. DOI: 10.17487/RFC4880. URL: https://rfc-editor.org/rfc/rfc4880.txt (strony 33, 35).

[17] Viktor Dukhovni i Wes Hardaker. *SMTP Security via Opportunistic DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS)*. RFC 7672. Paź. 2015. DOI: 10.17487/RFC7672. URL: https://rfc-editor.org/rfc/rfc7672.txt (strona 24).

[18] Zakir Durumeric i in. "Neither Snow Nor Rain Nor MITM...: An Empirical Analysis of Email Delivery Security". W: *Proceedings of the 2015 Internet Measurement Conference*. IMC '15. Tokyo, Japan: ACM, 2015, s. 27–39. ISBN: 978-1-4503-3848-6. DOI: 10.1145/2815675.2815695. URL: http://doi.acm.org/10.1145/2815675.2815695 (strona 9).

[19] *Ethereum Project*. URL: https://www.ethereum.org/ (strona 56).

[20] *Evil 32: Check Your GPG Fingerprints*. URL: evil32.com (strona 35).

[21] Ian D. Foster i in. "Security by Any Other Name: On the Effectiveness of Provider Based Email Security". W: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, s. 450–464. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813607. URL: http://doi.acm.org/10.1145/2810103.2813607 (strona 9).

[22] *Freemail*. Maj 2011. URL: https://github.com/freenet/plugin-Freemail/ (strony 46, 49).

[23] *Freenet documentation*. URL: https://freenetproject.org/pages/documentation.html (strona 46).

[24] *g10 Code – Smartcard*. URL: https://www.g10code.com/p-card.html (strona 39).

[25] Randall Gellens i Chris Newman. *POP3 Extension Mechanism*. RFC 2449. List. 1998. DOI: 10.17487/RFC2449. URL: https://rfc-editor.org/rfc/rfc2449.txt (strona 22).

[26] Roy Greenslade. "How Edward Snowden led journalist and film-maker to reveal NSA secrets". W: *The Guardian* (sierp. 2013). URL: https://www.theguardian.com/world/2013/aug/19/edward-snowden-nsa-secrets-glenn-greenwald-laura-poitra (strona 10).

[27] Michael Groves. *Sakai-Kasahara Key Encryption (SAKKE)*. RFC 6508. Lut. 2012. DOI: 10.17487/RFC6508. URL: https://rfc-editor.org/rfc/rfc6508.txt (strona 41).

[28] Paul E. Hoffman. *SMTP Service Extension for Secure SMTP over Transport Layer Security*. RFC 3207. Lut. 2002. DOI: 10.17487/RFC3207. URL: https://rfc-editor.org/rfc/rfc3207.txt (strona 24).

[29] Russ Housley. *Cryptographic Message Syntax (CMS)*. RFC 5652. Wrz. 2009. DOI: 10.17487/RFC5652. URL: https://rfc-editor.org/rfc/rfc5652.txt (strony 30, 40).

[30] *How many transactions per second will we have after the successful implementation of Constantinople version?* URL: https://www.reddit.com/r/ethereum/comments/9edwkk/how_many_transactions_per_second_will_we_have/ (strona 105).

[31] *I'm giving up on PGP*. URL: https://blog.filippo.io/giving-up-on-long-term-pgp/ (strony 10, 39).

[32] *I2P-Bote Technical Documentation*. URL: https://github.com/i2p/i2p.i2p-bote/blob/master/doc/techdoc.txt (strona 50).

[33] *Installing an S/MIME certificate*. URL: http://kb.mozillazine.org/Installing_an_SMIME_certificate (strona 32).

[34] *INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1*. RFC 3501. Mar. 2003. DOI: 10.17487/RFC3501. URL: https://rfc-editor.org/rfc/rfc3501.txt (strona 22).

[35] Andrey Jivsov. *Elliptic Curve Cryptography (ECC) in OpenPGP*. RFC 6637. Czer. 2012. DOI: 10.17487/RFC6637. URL: https://rfc-editor.org/rfc/rfc6637.txt (strona 33).

[36] *keybase.io*. URL: https://keybase.io/ (strona 37).

[37] Mehdi Khosrow-Pour. *Emerging Trends And Challenges in Information Technology Management*. Igi Global, 2006. ISBN: 1599040190 (strona 109).

[38] Scott Kitterman. *Sender Policy Framework (SPF) for Authorizing Use of Domains in Email, Version 1*. RFC 7208. Kw. 2014. DOI: 10.17487/RFC7208. URL: https://rfc-editor.org/rfc/rfc7208.txt (strona 26).

[39] Dr. John C. Klensin. *Simple Mail Transfer Protocol*. RFC 5321. Paź. 2008. DOI: 10.17487/RFC5321. URL: https://rfc-editor.org/rfc/rfc5321.txt (strony 19, 86).

[40] Loren Kohnfelder i Praerit Garg. *The threats to our products*. Kw. 1999. URL: https://adam.shostack.org/microsoft/The-Threats-To-Our-Products.docx (strona 74).

[41] Dr. Hugo Krawczyk i Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. Maj 2010. DOI: 10.17487/RFC5869. URL: https://rfc-editor.org/rfc/rfc5869.txt (strona 60).

[42] Murray Kucherawy. *Message Header Field for Indicating Message Authentication Status*. RFC 8601. Maj 2019. DOI: 10.17487/RFC8601. URL: https://rfc-editor.org/rfc/rfc8601.txt (strona 28).

[43] Murray Kucherawy, Dave Crocker i Tony Hansen. *DomainKeys Identified Mail (DKIM) Signatures*. RFC 6376. Wrz. 2011. DOI: 10.17487/RFC6376. URL: https://rfc-editor.org/rfc/rfc6376.txt (strona 26).

[44] Murray Kucherawy i Elizabeth Zwicky. *Domain-based Message Authentication, Reporting, and Conformance (DMARC)*. RFC 7489. Mar. 2015. DOI: 10.17487/RFC7489. URL: https://rfc-editor.org/rfc/rfc7489.txt (strona 27).

[45] Adam Langley, Mike Hamburg i Sean Turner. *Elliptic Curves for Security*. RFC 7748. Sty. 2016. DOI: 10.17487/RFC7748. URL: https://rfc-editor.org/rfc/rfc7748.txt (strona 59).

[46] Lavabit. *Security Through Asymmetric Encryption*. 2011. URL: https://web.archive.org/web/20110713191909/http://lavabit.com/secure.html (strona 42).

[47] *Lemon Email - On the road towards total decentralization of email*. URL: https://lemon.email/ (strona 51).

[48] Ladar Levison. "Dark Internet Mail Environment Architecture and Specifications". W: (2018). URL: https://darkmail.info/downloads/dark-internet-mail-environment-june-2018.pdf (strony 42, 44).

[49] Daniel Margolis i in. *SMTP MTA Strict Transport Security (MTA-STS)*. RFC 8461. Wrz. 2018. DOI: 10.17487/RFC8461. URL: https://rfc-editor.org/rfc/rfc8461.txt (strona 25).

[50] Moxie Marlinspike. *The Double Ratchet Algorithm, Revision 1*. List. 2016. URL: https://signal.org/docs/specifications/doubleratchet/ (strony 61–63).

[51] Moxie Marlinspike. *The Sesame Algorithm: Session Management for Asynchronous Message Encryption*. Kw. 2017. URL: https://signal.org/docs/specifications/sesame/ (strona 63).

[52] Moxie Marlinspike. *The X3DH Key Agreement Protocol, Revision 1*. List. 2016. URL: https://signal.org/docs/specifications/x3dh/ (strona 60).

[53] Keith Moore i Chris Newman. *Cleartext Considered Obsolete: Use of Transport Layer Security (TLS) for Email Submission and Access*. RFC 8314. Sty. 2018. DOI: 10.17487/RFC8314. URL: https://rfc-editor.org/rfc/rfc8314.txt (strona 24).

[54] K. Murchison i M. Crispin. *The LOGIN SASL Mechanism*. Internet-Draft draft-murchison-sasl-login-00. Work in Progress. Internet Engineering Task Force, sierp. 2003. 7 s. URL: https://datatracker.ietf.org/doc/html/draft-murchison-sasl-login-00 (strona 18).

[55] Chris Newman. *Using TLS with IMAP, POP3 and ACAP*. RFC 2595. Czer. 1999. DOI: 10.17487/RFC2595. URL: https://rfc-editor.org/rfc/rfc2595.txt (strona 24).

[56] *Optimizing a distributed spam filter for FreeNet*. Wrz. 2015. URL: https://github.com/freenet/plugin-WebOfTrust/blob/next/developer-documentation/core-developers-manual/OadSFfF-version1.2-non-print-edition.pdf (strona 48).

[57] Trevor Perrin. *The XEdDSA and VXEdDSA Signature Schemes*. Paź. 2016. URL: https://signal.org/docs/specifications/xeddsa/ (strona 59).

[58] Karen Renaud, Melanie Volkamer i Arne Renkema-Padmos. "Why Doesn't Jane Protect Her Privacy?" W: (2014). URL: https://www.petsymposium.org/2014/papers/Renkema.pdf (strona 55).

[59] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Sierp. 2018. DOI: 10.17487/RFC8446. URL: https://rfc-editor.org/rfc/rfc8446.txt (strona 23).

[60] Certicom Research. "SEC 1: Elliptic Curve Cryptography – Version 2.0". W: (maj 2009). URL: http://www.secg.org/sec1-v2.pdf (strona 59).

[61] Pete Resnick. *Internet Message Format*. RFC 5322. Paź. 2008. DOI: 10.17487/RFC5322. URL: https://rfc-editor.org/rfc/rfc5322.txt (strona 17).

[62] *Retroshare*. URL: http://retroshare.net/ (strona 46).

[63] Thomas Roessler i Michael Elkins. *MIME Security with OpenPGP*. RFC 3156. Sierp. 2001. DOI: 10.17487/RFC3156. URL: https://rfc-editor.org/rfc/rfc3156.txt (strona 33).

[64] Dr. Marshall T. Rose i John G. Myers. *Post Office Protocol - Version 3*. RFC 1939. Maj 1996. DOI: 10.17487/RFC1939. URL: https://rfc-editor.org/rfc/rfc1939.txt (strony 21, 83).

[65] Jim Schaad, Blake C. Ramsdell i Sean Turner. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Certificate Handling*. RFC 8550. Kw. 2019. DOI: 10.17487/RFC8550. URL: https://rfc-editor.org/rfc/rfc8550.txt (strona 30).

[66] Jim Schaad, Blake C. Ramsdell i Sean Turner. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification*. RFC 8551. Kw. 2019. DOI: 10.17487/RFC8551. URL: https://rfc-editor.org/rfc/rfc8551.txt (strona 30).

[67] Mark J. Schertler, Guido Appenzeller i Mark J. Schertler. *Identity-Based Encryption Architecture and Supporting Data Structures*. RFC 5408. Sty. 2009. DOI: 10.17487/RFC5408. URL: https://rfc-editor.org/rfc/rfc5408.txt (strona 40).

[68] Mark J. Schertler i Xavier Boyen. *Identity-Based Cryptography Standard (IBCS) #1: Supersingular Curve Implementations of the BF and BB1 Cryptosystems*. RFC 5091. Grud. 2007. DOI: 10.17487/RFC5091. URL: https://rfc-editor.org/rfc/rfc5091.txt (strona 41).

[69] Mark J. Schertler i Mark J. Schertler. *Using the Boneh-Franklin and Boneh-Boyen Identity-Based Encryption Algorithms with the Cryptographic Message Syntax (CMS)*. RFC 5409. Sty. 2009. DOI: 10.17487/RFC5409. URL: https://rfc-editor.org/rfc/rfc5409.txt (strona 40).

[70] David Shaw. *The Camellia Cipher in OpenPGP*. RFC 5581. Czer. 2009. DOI: 10.17487/RFC5581. URL: https://rfc-editor.org/rfc/rfc5581.txt (strona 33).

[71] Rob Siemborski i Alexey Melnikov. *SMTP Service Extension for Authentication*. RFC 4954. Lip. 2007. DOI: 10.17487/RFC4954. URL: https://rfc-editor.org/rfc/rfc4954.txt (strony 18, 20).

[72] Rob Siemborski i Abhijit Menon-Sen. *The Post Office Protocol (POP3) Simple Authentication and Security Layer (SASL) Authentication Mechanism*. RFC 5034. Lip. 2007. DOI: 10.17487/RFC5034. URL: https://rfc-editor.org/rfc/rfc5034.txt (strona 21).

[73] *Simple Authentication and Security Layer (SASL) Mechanisms*. URL: https://ietf.org/assignments/sasl-mechanisms/sasl-mechanisms.xml (strona 18).

[74] Mark C. Smith. *Definition of the inetOrgPerson LDAP Object Class*. RFC 2798. Kw. 2000. DOI: 10.17487/RFC2798. URL: https://rfc-editor.org/rfc/rfc2798.txt (strona 32).

[75] *Solidity Language Documentation*. URL: https://solidity.readthedocs.io/ (strona 56).

[76] *SquirrelMail – Webmail for Nuts!* URL: https://squirrelmail.org/ (strona 23).

[77] Petr Švenda. "Basic comparison of Modes for Authenticated-Encryption". W: (paź. 2004). URL: https://www.fi.muni.cz/~xsvenda/docs/AEcomparison2004.pdf (strona 47).

[78] *Swarm documentation: Architecture*. URL: https://swarm-guide.readthedocs.io/en/latest/architecture.html (strona 58).

[79] *Technical specifications for the IPFS protocol stack*. URL: https://github.com/ipfs/specs (strony 51, 59).

[80] *The Brewing Problem Of PGP Short-ID Collision Attacks – Phoronix*. URL: https://www.phoronix.com/scan.php?page=news_item&px=Short-PGP-Collision-Attacks (strona 35).

[81] *The GNU Privacy Guard*. URL: https://gnupg.org/ (strony 34, 38).

[82] *The Invisible Internet Project*. URL: https://geti2p.net/ (strona 50).

[83] Blake Thorne. *Asynchronous Communication Is The Future Of Work*. Sierp. 2018. URL: http://blog.idonethis.com/asynchronous-communication/ (strona 109).

[84] Guido Urdaneta, Guillaume Pierre i Maarten van Steen. "A Survey of DHT Security Techniques". W: *ACM Computing Surveys* 43.2 (sty. 2011). URL: http://www.globule.org/publi/SDST_acmcs2009.html (strona 45).

[85] *Voltage SecureMail Data Sheet*. URL: https://www.microfocus.com/media/data-sheet/voltage_securemail_ds.pdf (strona 40).

[86] Jonathan Warren. "Bitmessage: A Peer – to – Peer Message Authentication and Delivery System". W: (2012). URL: https://bitmessage.org/bitmessage.pdf (strona 45).

[87] *Webmail – The Horde Project*. URL: https://www.horde.org/apps/webmail (strona 23).

[88] *What's the matter with PGP?* URL: https://blog.cryptographyengineering.com/2014/08/13/whats-matter-with-pgp/ (strony 10, 39).

[89] Kurt Zeilenga. *The PLAIN Simple Authentication and Security Layer (SASL) Mechanism*. RFC 4616. Sierp. 2006. DOI: 10.17487/RFC4616. URL: https://rfc-editor.org/rfc/rfc4616.txt (strona 18).

[90] Kurt Zeilenga i Alexey Melnikov. *Simple Authentication and Security Layer (SASL)*. RFC 4422. Czer. 2006. DOI: 10.17487/RFC4422. URL: https://rfc-editor.org/rfc/rfc4422.txt (strona 18).

Dodatek A

Konfiguracja klienta Thunderbird do użycia z aplikacją TigerMail

Stwórz plik `tigermail.yaml`, podmieniając ścieżki w liniach `socketPath:` na właściwe.

Listing A.1: Plik `tigermail.yaml`

```
- direction: outgoing
  addrress: 127.0.0.1
  port: 9110
  socketPath: /path/to/your/pop3.socket

- direction: outgoing
  address: 127.0.0.1
  port: 9587
  socketPath: /path/to/your/smtp.socket
```

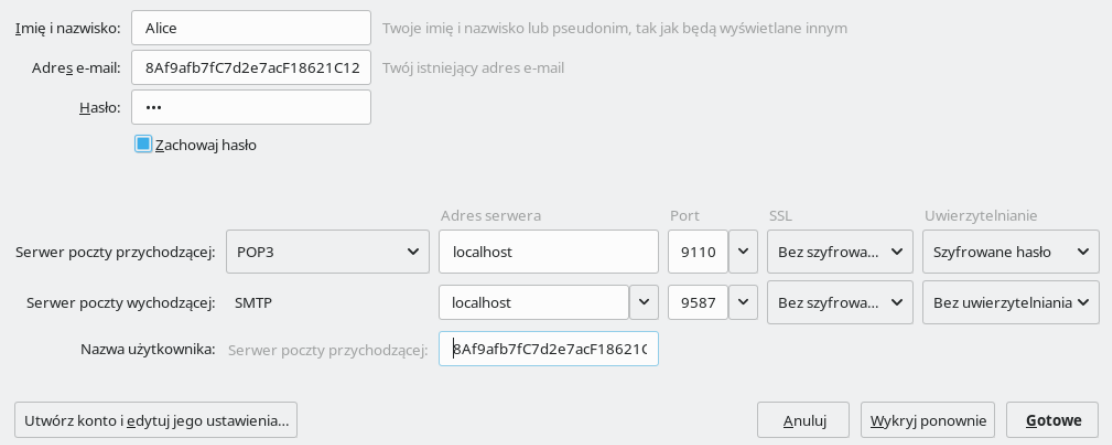
Uruchom program `thunderbird` używając polecenia:

```
ip2unix -f tigermail.yaml thunderbird
```

Utwórz konfigurację nowego konta (rysunek A.1):

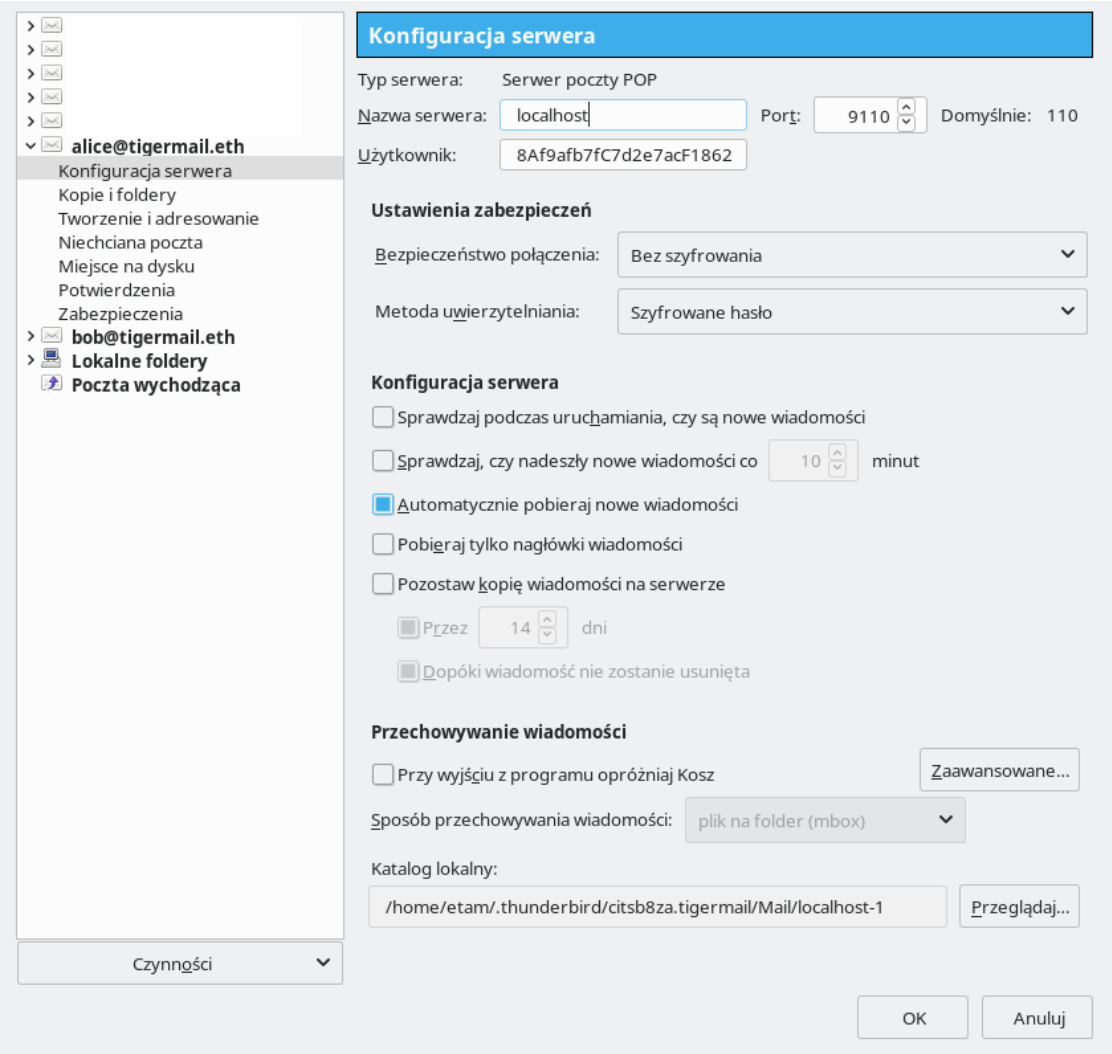
- Jako adres *email* podaj adres *Ethereum*, bez „0x” na początku i z dodanym `@tigermail.eth` na końcu,
- Hasło może być dowolne, ponieważ jest ignorowane przez aplikację *TigerMail*,
- Serwer poczty przychodzącej:
  - Protokół: *POP3*,
  - Adres serwera: `localhost`,
  - Port: 9110,

  - *SSL*: Bez szyfrowania,
  - Uwierzytelnianie: Szyfrowane hasło,
  - Nazwa użytkownika: taka sama jak adres *email*,
- Serwer poczty wychodzącej:
  - Adres serwera: `localhost`,
  - Port: 9587,
  - *SSL*: Bez szyfrowania,
  - Uwierzytelnianie: Bez uwierzytelniania,
- Utwórz konto i edytuj jego ustawienia,



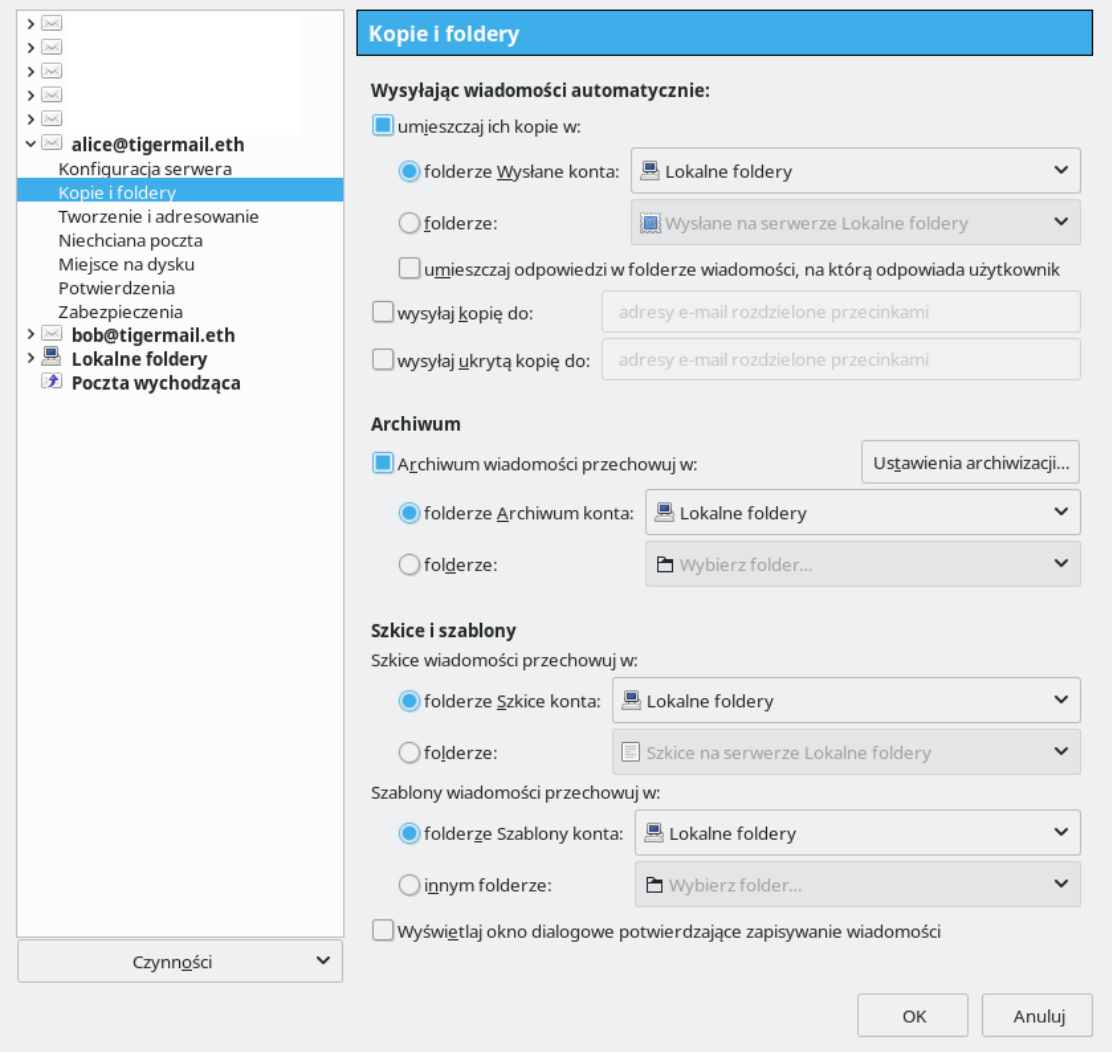
Rysunek A.1: Konfiguracja nowego konta w *Thunderbird*.

Konfiguracja serwera (rysunek A.2): Wyłącz zostawianie kopii na serwerze.



Rysunek A.2: Konfiguracja serwera w *Thunderbird*.

Kopie i foldery (rysunek A.3): Wszelkie wiadomości przechowuj w lokalnych folderach.



Rysunek A.3: Konfiguracja kopii i folderów w *Thunderbird*.