AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA TELEKOMUNIKACJI

Praca dyplomowa inżynierska

Opracowanie biblioteki programistycznej do bezpiecznego uwierzytelniania urządzeń AVR.

Development of libraries for authentication of AVR devices.

Autor: *Kacper Żuk*
Kierunek studiów: *Teleinformatyka*
Opiekun pracy: *dr inż. Jarosław Bułat*

Kraków, 2016

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Spis treści

Wprowadzenie

1 Charakterystyka platformy sprzętowej

Tabela 1.1. Wybrane modele AVR wraz z ich parametrami

Nazwa	**SRAM**[1]	**Wymagane napięcie**	**Taktowanie procesora**	**Liczba linii I/O**
ATtiny4 [1]	32 B	1.8 - 5.5 V	do 12 MHz	4
ATmega32u4 [2]	2,5 KB	2.7 - 5.5 V	do 16 MHz	26
ATxmega384C3 [3]	32 KB	1.6 - 3.6 V	do 32 MHz	50

Mikropocesory Atmel AVR są w większości 8-bitowe i na takich skupia się praca. Rodzina AVR jest szeroka, kilka wybranych modeli przedstawiono w tabeli 1.1. W pracy wykorzystany został model ATmega32u4 z 2,5 kilobajta SRAM [2].

SRAM jest głównym ograniczeniem w implementacji uwierzytelniania, ponieważ 32 bajty nie są wystaczające do przeprowadzania operacji kryptograficznych, przy których sam klucz zajmuje 16 lub 32 bajty. Należy też pamiętać, że obsługa bezpiecznego połączenia nie może zajmować całości pamięci. Część pamięci należy przeznaczyć na obsługę peryferiów oraz właściwą logikę programu.

Istotnym elementem jest też wielkość domyślnych buforów. *Arduino* w modułach *Serial* oraz *SoftwareSerial* domyślnie używa 16- lub 64-bajtowego (w zależności od ilości dostępnej pamięci) buforu na przychodzące dane[2]. Przy wiadomościach dłuższych niż 32 bajty oznacza to, że zbyt długie przetwarzanie jednej wiadomości spowoduje błędne odebranie następnej, jeżeli zostanie ona za szybko wysłana.

Maksymalne taktowanie mikroprocesora zależy od konkretnego modelu (od 12 do 32 MHz) oraz napięcia zasilania. Wykorzystywany w pracy Atmega32u4 zasilany był napięciem 5 V, co przekłada się na taktowanie 16 MHz.

[1] ang. Static Random Access Memory

[2] https://github.com/arduino/Arduino/blob/master/hardware/arduino/avr/cores/arduino/HardwareSerial.h

2 Metody uwierzytelniania

W zależności od potrzeb i ograniczeń stosuje się różne metody uwierzytelniania podmiotów w komunikacji. Wyróżnić należy uwierzytelnianie przy pomocy kryptografii asymetrycznej, w której używana jest para matematycznie związanych ze sobą kluczy, oraz uwierzytelnianie przy pomocy kryptografii symetrycznej, w której używany jest jeden, współdzielony, tajny klucz.

Klucze w przypadku kryptografii asymetrycznej muszą posiadać konkretne właściwości. W przypadku algorytmu RSA bezpieczeństwo polega na trudności w faktoryzowaniu dużych liczb, co wymaga stosowania kluczy co najmniej 2048 bitowych [4]. Klucze w przypadku kryptografii symetrycznej nie muszą mieć konkretnych właściwości poza ich nieprzewidywalnością.

Ważną różnicą jest też wydajność. Kryptografia asymetryczna jest dużo bardziej złożona obliczeniowo od symetrycznej [5]. Jest to szczególnie istotne na ograniczonych sprzętowo systemach wbudowanych. Przewagą kryptografii asymetrycznej jest jednak brak konieczności ustalenia wspólnego klucza przed rozpoczęciem komunikacji, jak ma to miejsce w przypadku kryptografii symetrycznej.

Zalecanym rozwiązaniem jest najpierw ustalenie wspólnego, tajnego klucza przy użyciu kryptografii asymetrycznej, a następnie użycie tego klucza do kryptografii symetrycznej [5].

2.1. Kryptografia asymetryczna

Przy wyborze algorytmu używanego do ustalania klucza dla potrzeb pracy istotne były:

– jakość implementacji algorytmów dostępnych na mikroprocesory AVR,

– złożoność obliczeniowa,

– długość klucza wymagana do zapewnienia niezbędnego poziomu bezpieczeństwa.

Biblioteka *AVR-Crypto-Lib* dostarcza implementację algorytmów RSA oraz DSA[1]. Biblioteka *Emsign* dostarcza implementację RSA, lecz tylko z 64 bitowym kluczem[2], co nie jest wystarczające dla zapewnienia bezpieczeństwa. Komercyjna biblioteka *LightCrypt-AVR8-ECC* oraz biblioteka *micro-ecc* dostarczają implementację kryptografii opartej o krzywe eliptyczne[3]. Brak jest na rynku implementacji

[1] *https://trac.cryptolib.org/avr-crypto-lib/browser*
[2] *http://www.emsign.nl/*
[3] *http://industrial.crypto.cmmsigma.eu/lightcrypt_avr8/lc_avr8_ecc.pl.html*

innych algorytmów klucza publicznego. Dostępność implementacji ogranicza wybór algorytmu do RSA, DSA oraz krzywych eliptycznych.

Następnym kryterium jest złożoność obliczeniowa. W analizie przeprowadzonej przez pracowników *Sun Microsystems Laboratories* wykazano, że na mikroprocesorach AVR algorytmy oparte o krzywe eliptyczne są o rząd wielkości szybsze od algorytmu RSA [6].

Krzywe eliptyczne wymagają najkrótszych kluczy. Rekomendacje NIST [4] (National Institute of Standards and Technology) podają, że 256-bitowy klucz ECC *(ang. Elliptic Curve Cryptography)* zapewnia bezpieczeństwo porównywalne do 3072-bitowego klucza RSA lub DSA.

W związku z przewagą krzywych eliptycznych przy zadanych założeniach do ustalenia wspólnego klucza wybrano algorytm ECDH *(ang. Elliptic Curve Diffie-Hellman)*. Wadą tego rozwiązania jest niezmienność klucza ustalanego tą metodą. Powoduje to brak utajnienia przekazywania *(ang. forward secrecy)*.

2.2. Kryptografia symetryczna

Przy wyborze algorytmu dla potrzeb pracy istotne były:

– jakość implementacji dostępnych na mikroprocesory AVR,

– możliwość szyfrowania i uwierzytelniania danych.

Powszechnie dostępne są jedynie implementacje samych blokowych algorytmów szyfrowania takich jak AES oraz DES lub funkcji skrótu takich jak SHA-256. By uzyskać uwierzytelnianie wiadomości o zmiennej długości trzeba algorytmy blokowe zastosować w odpowiedni sposób. Przykładem jest tryb CBC-MAC *(ang. Cipher Block Chaining - Message Authentication Code)*. Pozwala on na wygenerowanie kodu uwierzytelniającego daną wiadomość, poprzez zaszyfrowanie jej w trybie CBC i użycie ostatniego bloku szyfrogramu jako kodu.

Tryb CBC-MAC – przy nieprawidłowej implementacji – może wprowadzić podatności:

– użycie zmiennego wektora inicjalizacyjnego i przesyłanie go wraz z uwierzytelnianą wiadomością pozwala na dowolną modyfikację pierwszego bloku (16 bajtów) wiadomości bez zmiany kodu uwierzytelniającego,

– użycie tego samego klucza do szyfrowania w trybie CBC oraz uwierzytelniania w trybie CBC-MAC pozwala na obliczenie użytego klucza bez jego wcześniejszej znajomości,

– atakujący znający dwie wiadomości m oraz m' oraz ich kody uwierzytelniające może policzyć klucz uwierzytelniający wiadomości będącej specyficznym połączeniem wiadomości m oraz m'.

Wszystkim tym podatnościom da się zapobiec poprzez użycie niezmiennego wektora inicjalizacyjnego oraz zaszyfrowanie ostatniego bloku innym kluczem (tryb ECBC-MAC, *ang. Encrypt-last-block CBC-MAC*).

Alternatywą jest także zastosowanie HMAC *(ang. keyed-hash message authentication code)*. Kodem uwierzytelniającym jest wtedy wynik funkcji skrótu policzony z połączenia współdzielonego klucza oraz uwierzytelnianej wiadomości [7].

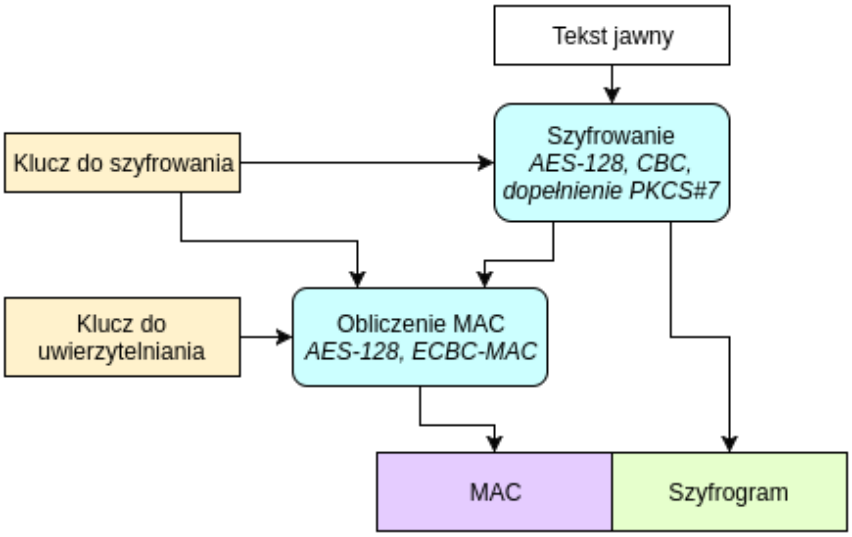
W pracy do uwierzytelniania wybrano AES w trybie ECBC-MAC. Zaletą tego rozwiązania jest możliwość użycia tej samej implementacji trybu CBC zarówno do szyfrowania jak i jako element trybu ECBC-MAC.

W implementacji szyfrowania w trybie CBC należało rozwiązać problemy wymienione poniżej.

1. Użycie przewidywalnych wektorów inicjalizacyjnych pozwala atakującemu na zgadywanie treści wiadomości, a następnie – poprzez odpowiednie spreparowanie nowej wiadomości – weryfikację, czy wiadomość się zgadza. Wektory inicjalizacyjne muszą być nieprzewidywalne.

2. CBC operuje na blokach danych, a więc wiadomości o długości niebędącej wielokrotnością długości bloku trzeba dopełniać. Oznacza to że do szyfrowanej wiadomości trzeba dołączać jej długość lub użyć dopełnienia, które jest jednoznaczne.

Szyfrowanie z uwierzytelnianiem jest połączone wedle zasady *Encrypt-then-MAC*. Oznacza to że wiadomość najpierw jest szyfrowana, a następnie uwierzytelniany jest szyfrogram, a nie bezpośrednio wiadomość. Jest to rozwiązanie zapewniające najwyższe bezpieczeństwo, zapobiegające między innymi atakom typu *padding oracle* [8]. Całość procesu została przedstawiona na rysunku 2.1.



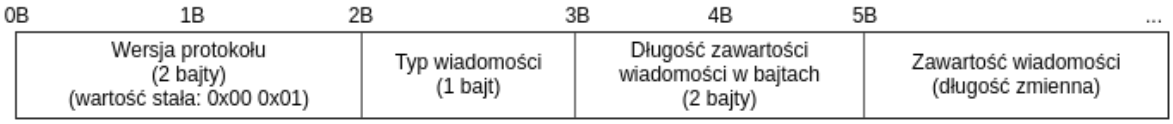
Rys. 2.1. Proces szyfrowania i uwierzytelniania danych

3 Implementacja protokołu komunikacji

Protokół komunikacji między dwoma węzłami zaprojektowano i zaimplementowano z następującymi założeniami:

- pełna funkcjonalność przy jak najmniejszych wymaganiach sprzętowych, w szczególności przy dostępnej małej ilości pamięci operacyjnej,
- częściowa niezależność od warstwy sieciowej,
- zapewnienie uwierzytelniania i szyfrowania wiadomości.

3.1. Podstawowe struktury protokołu



Rys. 3.1. Budowa wiadomości w protokole komunikacji.

Podstawową jednostką protokołu są wiadomości zbudowane według schematu zaprezentowanego na Rys. 3.1.

Zdefiniowane typy wiadomości są przedstawione w tabeli 3.1.

Budowę przykładowych wiadomości przedstawiono na rysunkach A.1, A.2 oraz A.3 w dodatku A.

Odbiorca powinien zweryfikować:

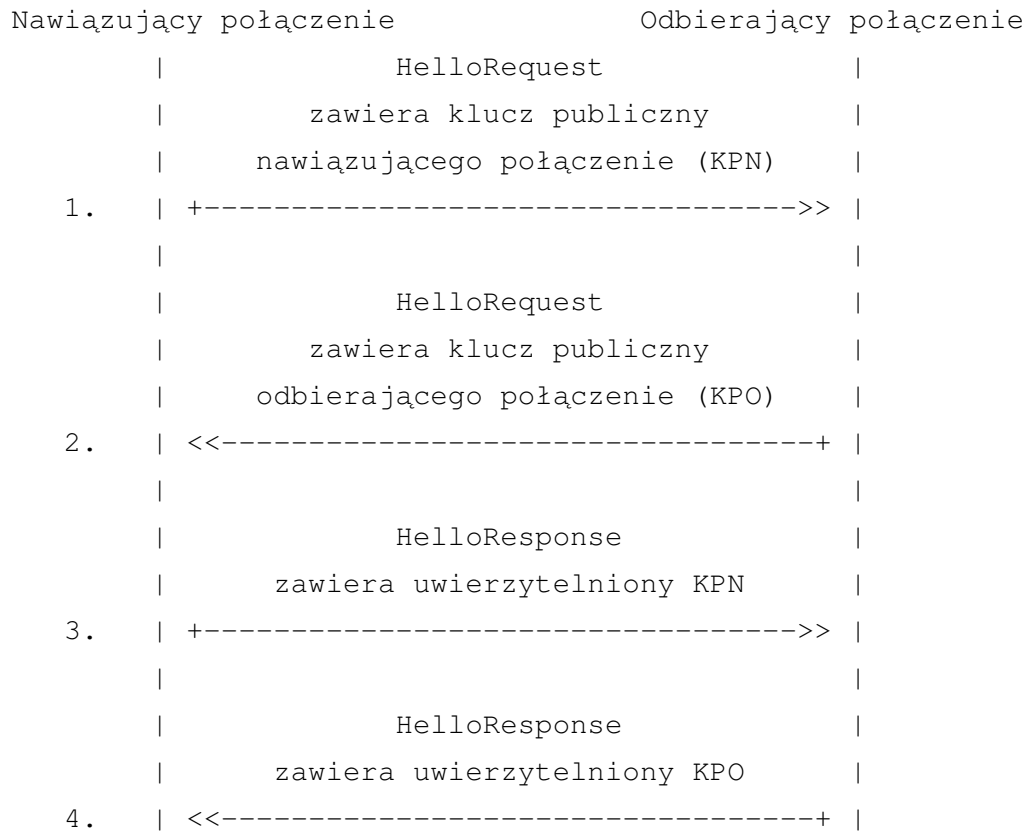
- zgodność wersji protokołu – wymagane bajty 0x00 oraz 0x01,
- prawidłowość bajtu określającego typ – wymagana wartość 0x00, 0x01 lub 0x02,
- zgodność zadeklarowanej długości zawartości z typem,
- w przypadku typów HelloResponse oraz EncryptedData – prawidłowość kodu uwierzytelniającego.

W przypadku niezgodności któregokolwiek elementu wiadomość powinna zostać zignorowana.

Tabela 3.1. Typy wiadomości wraz z ich charakterystyką

Nazwa typu wiadomości	Wartość pola typ	Długość zawartości	Czy zawartość jest zaszyfrowana	Czy zawartość jest uwierzytelniona
HelloRequest	0x00	64 bajty	nie	nie
HelloResponse	0x01	96 bajtów	tak	tak
EncryptedData	0x02	zmienna, minimum 32 bajty	tak	tak

3.2. Nawiązywanie połączenia



Rys. 3.2. Kolejność wymiany wiadomości w procesie nawiązywania połączenia

Kolejność przesyłania wiadomości w celu nawiązania połączenia przedstawiona jest na rysunku 3.2. HelloRequest zawiera klucz publiczny węzła, który go wysyła. Węzeł, który odbiera HelloRequest, używa swojego klucza publicznego oraz klucza publicznego z odebranej wiadomości do ustalenia sekretnego klucza. Nawiązującym połączenie może być dowolny węzeł.

Po ustaleniu wspólnego klucza węzły mogą wysłać HelloResponse, który zawiera zaszyfrowany i uwierzytelniony klucz publiczny węzła go wysyłającego. Jeżeli węzeł odbierający wiadomość skutecznie potwierdzi, że jest ona prawidłowo uwierzytelniona, a zdeszyfrowany klucz publiczny pokrywa się z kluczem przesłanym wcześniej w HelloRequest, połączenie uznawane jest za nawiązane. Jeżeli przed

odebraniem HelloResponse odebrany był więcej niż jeden HelloRequest, brana pod uwagę jest wiadomość odebrana jako ostatnia. Po nawiązaniu połączenia wymieniane mogą być tylko wiadomości typu EncryptedData.

Istotne jest, że protokół nie zapewnia autentyczności danego klucza publicznego. Powinno to zostać zweryfikowane niezależnie, na przykład poprzez wyświetlenie skrótu klucza użytkownikowi i poproszenie go o potwierdzenie, że na obu urządzeniach uczestniczących w komunikacji jest wyświetlony taki sam klucz.

Protokół zakłada też, że przesyłanie danych jest niezawodne, połączeniowe oraz zachowana jest ich kolejność. Nie są więc zaimplementowane retransmisje ani wykrywanie, czy drugi węzeł rzeczywiście nasłuchuje na przychodzące dane.

3.3. Generowanie współdzielonego klucza

Każdy z węzłów po odebraniu HelloRequest używa odebranego klucza publicznego oraz swojego klucza publicznego do ustalenia wspólnego sekretu przy użyciu algorytmu ECDH oraz proponowanej przez NIST krzywej eliptycznej P-256 [9] (w RFC 5480 nazwaną krzywą secp256r1 [10]).

Z sekretu będącego wynikiem algorytmu ECDH liczony jest skrót przy użyciu algorytmu SHA-256. Następnie jest on dzielony na dwie części po 128-bitów. Pierwsza część staje się współdzielonym kluczem używanym do szyfrowania, druga część staje się współdzielonym kluczem używanym do uwierzytelniania.

Implementacja algorytmu ECDH z krzywą eliptyczną P-256 pochodzi z biblioteki *micro-ecc*. Jest to jedyna darmowa biblioteka implementująca ECDH.

3.4. Szyfrowanie i deszyfrowanie danych

Szyfrowanie zawartości wiadomości odbywa się za pomocą szyfru blokowego AES ze 128-bitowym kluczem używanym w trybie CBC. Wektor inicjalizacyjny jest losowy i dołączany do zawartości przesyłanej wiadomości przed szyfrogramem. Tekst jawny jest dopełniany do pełnego bloku według algorytmu zdefiniowanego w PKCS#7 [11].

Stworzona biblioteka nie posiada własnego źródła liczb losowych, musi zostać ono dostarczone w ramach integracji. Przykładowa metoda generowania liczb losowych na platformie Arduino została opisana w Dodatku C.

Właściwe kroki potrzebne do zaszyfrowania zawartości wiadomości wypisano poniżej.

1. Dopełnienie tekstu jawnego do pełnego bloku:
   - jeżeli długość tekstu jawnego jest wielokrotnością długości bloku, do tekstu jawnego doklejone musi być 16 bajtów o wartości 16,

– w przeciwnym wypadku, gdy wymagane jest dopełnienie N bajtów, do tekstu jawnego doklejone musi być N bajtów o wartości N.

2. Zaszyfrowanie dopełnionego tekstu jawnego w trybie CBC z losowym wektorem inicjalizacyjnym.
3. Doklejenie wektora inicjalizacyjnego przed szyfrogramem.

Przykłady dopełniania danych o różnych długościach przedstawiono w tabeli 3.2.

Tabela 3.2. Dopełnanie danych do pełnego bloku. Dopełnienie zaznaczone zostało kolorem niebieskiem.

Dane „Witaj swiecie” (13 bajtów) zostają dopełnione 3 bajtami o wartości 3 (0x03):

```
0x57 0x69 0x74 0x61
0x6a 0x20 0x73 0x77
0x69 0x65 0x63 0x69
0x65 0x03 0x03 0x03
```

Dane „Witaj swiecie !!” (16 bajtów) zostają dopełnione 16 bajtami o wartości 16 (0x10):

```
0x57 0x69 0x74 0x61
0x6a 0x20 0x73 0x77
0x69 0x65 0x63 0x69
0x65 0x20 0x21 0x21
0x10 0x10 0x10 0x10
0x10 0x10 0x10 0x10
0x10 0x10 0x10 0x10
0x10 0x10 0x10 0x10
```

Kod implementujący szyfrowanie danych przedstawiono w tabeli B.1 w dodatku B.
Właściwe kroki potrzebne do odszyfrowania zawartości wypisano poniżej.

1. Oddzielenie wektora inicjalizacyjnego od szyfrogramu.
2. Zdeszyfrowanie szyfrogramu w trybie CBC przy wykorzystaniu oddzielonego wektora inicjalizacyjnego.
3. Pobranie wartości ostatniego bajtu zdeszyfrowanego ciągu:
   – wartość ta nazywana jest dalej N.
4. Zweryfikowanie poprawności dopełnienia:
   – ostatnie N bajtów musi mieć wartość N,
   – jeżeli dopełnienie jest nieprawidłowe, cała wiadomość jest ignorowana.
5. Usunięcie ostatnich N bajtów.

Kod implementujący odszyfrowanie danych przedstawiono w tabeli B.2 w dodatku B.

Implementacja algorytmu AES pochodzi z biblioteki *AVR-Crypto-Lib*. Jest to najlepiej udokumentowana, darmowa biblioteka implementująca algorytm AES. Implementacja trybu CBC oraz algorytmu dopełniania zostały zrealizowane w ramach pracy.

3.5. Uwierzytelnienie wiadomości

Uwierzytelnienie wiadomości odbywa się poprzez dołączenie do zawartości MAC. MAC dla danej wiadomości tworzony jest za pomocą szyfru blokowego AES ze 128-bitowym kluczem używanym w trybie ECBC-MAC. Wektor inicjalizacyjny wypełniony jest zerami i nie jest przesyłany. Uwierzytelniany jest kompletny szyfrogram wraz z wektorem inicjalizacyjnym użytym do szyfrowania, a nie tekst jawny. Długość szyfrogramu wraz z wektorem inicjalizacyjnym zawsze będzie wielokrotnością długości bloku, a więc nie jest stosowane dopełnianie.

Tryb ECBC-MAC to tryb CBC-MAC, którego wynik jest dodatkowo szyfrowany innym kluczem niż ten użyty do CBC-MAC. W tej pracy do CBC-MAC użyty jest klucz służący do uwierzytelniania, a wynik CBC-MAC jest szyfrowany używając klucza służącego do szyfrowania.

Właściwe kroki potrzebne do obliczenia kodu uwierzytelniającego opisano poniżej.

1. Obliczenie ostatniego bloku będącego wynikiem zaszyfrowania szyfrogramu wraz z wektorem inicjalizacyjnym w trybie CBC z wektorem inicjalizacyjnym wypełnionym zerami przy użyciu klucza przeznaczonego do uwierzytelniania.

2. Zaszyfrowanie bloku przy wykorzystaniu AES i klucza przeznaczonego do szyfrowania.

Węzeł wysyłający dokleja kod uwierzytelniający przed szyfrogramem. Węzeł odbierający oddziela otrzymany kod od szyfrogramu, oblicza kod uwierzytelniający dla danego szyfrogramu i porównuje, czy zgadza się on z kodem otrzymanym. Jeżeli kod obliczony różni się od kodu otrzymanego, cała wiadomość jest ignorowana.

Kod implementujący obliczanie MAC przedstawiono w tabeli B.3 w dodatku B.

Implementacja trybu ECBC-MAC została zrealizowana w ramach pracy.

4 Test opracowanej biblioteki

Walidacji poddane zostały trzy aspekty:

– poprawność zaprojektowania interfejsu programistycznego stworzonej biblioteki,

– możliwość poprawnego nawiązania połączenia i przesłania danych między dwoma węzłami,

– poprawność implementacji algorytmów kryptograficznych.

4.1. Poprawność interfejsu programistycznego

Poprawny interfejs programistyczny biblioteki musi umożliwiać stworzenie pełnego rozwiązania zapewniającego bezpieczną komunikację. Zostało to zweryfikowane poprzez stworzenie przykładowego oprogramowania wykorzystującego bibliotekę. Oprogramowanie to powstało na platformę Arduino oraz wykorzystuje moduł Bluetooth XM-15B do komunikacji.

Po uruchomieniu urządzenia inicjalizowana jest stworzona biblioteka implementujące bezpieczną komunikację, a moduł Bluetooth zostaje skonfigurowany w trybie *slave* z nazwą „seconn" i oczekuje na połączenie. Przy inicjalizacji biblioteki przekazywane są następujące funkcje zaimplementowane w przykładowym oprogramowaniu:

– funkcja obsługująca przekazywanie danych z biblioteki do modułu Bluetooth celem przesłania do drugiego węzła,

– funkcja obsługująca i przekazująca połączeniem szeregowym dane przychodzące z biblioteki, które zostały przez bibliotekę poprawnie uwierzytelnione oraz zdeszyfrowane,

– funkcja obsługująca powiadomienia o zmianie stanu połączenia przychodzące z biblioteki i przekazująca połączeniem szeregowym uwierzytelniony klucz publiczny drugiego węzła po nawiązaniu bezpiecznego połączenia,

– funkcja generująca liczby losowe stworzona w oparciu o implementację zaproponowaną w bibliotece *micro-ecc* (opis w dodateku C).

Tabela 4.1. Przykładowe dane przesłane przez połączenie szeregowe. Stan nr 4 oznacza, że odebrany został prawidłowo uwierzytelniony pakiet HelloResponse zawierający klucz publiczny drugiego węzła.

```
S!
Our pubkey is: 0x6D35D8BE2F0C67210C143E649F250FC4E
B014F25C305AC7C2FA6B02F0B4A4E63EA0BB52367AAF96E63B
BD968C186830ADE2B2A24769CB32E1E1A690F51079C7E
State:4
Pubkey of other side is: 0x22743237010F6830994886B
BFB781184C10D25E1D6819D075F40CF0724FEC049FF4804F82
58C14049E373595BC0987061B93493E16C8C59E8C7C2A64FF5
247B0
D:>Some message...<
```

Dodatkowo zaimplementowane zostało przekazywanie połączeniem szeregowym klucza publicznego urządzenia celem weryfikacji z drugim węzłem oraz przekazywanie danych przychodzących z modułu Bluetooth do biblioteki. Przepływ danych między przykładowym oprogramowaniem, biblioteką, drugim węzłęm oraz połączeniem szeregowym przedstawiono w Dodatku D.

Należy zwrócić uwagę, że oprogramowanie korzystające z biblioteki nie implementuje żadnej logiki związanej z protokołem komunikacji. Jest ona w całości zaimplementowana w bibliotece, a oprogramowanie jedynie zajmuje się przekazywaniem danych między fizycznym połączeniem a biblioteką.

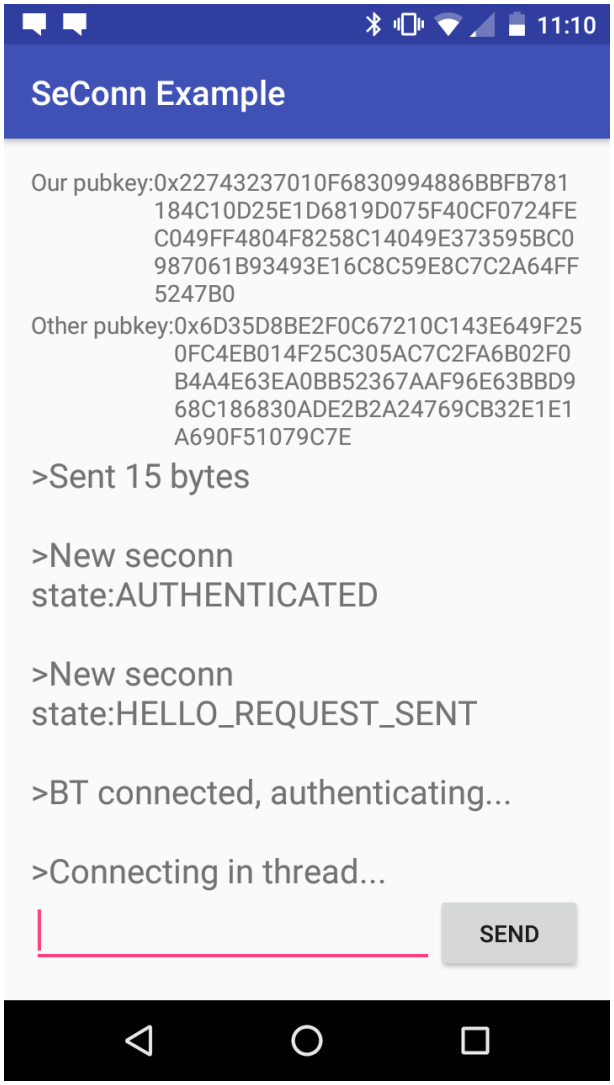
W tabeli 4.1 przedstawiono przykładowe dane, jakie zostają przesłane przez połączenie szeregowe. W tym przypadku drugi węzeł był odpowiedzialny za rozpoczęcie połączenia (przesłanie pierwszego HelloRequest), a po poprawnym uwierzytelnieniu przesłał uwierzytelnioną i zaszyfrowaną wiadomość o treści „Some message...”.

4.2. Poprawność implementacji algorytmów kryptograficznych

Poprawność implementacji algorytmów CBC i dopełniania według PKCS#7 stworzonych w ramach pracy oraz algorytmów AES, SHA-256 oraz ECDH dostarczonych przez zewnętrzne biblioteki została zwalidowana poprzez stworzenie drugiej implementacji protokołu w języku Java.

Implementacje algorytmów AES, CBC, dopełniania według PKCS#7, SHA-256 oraz ECDH pochodzą z pakietów java.security oraz javax.crypto biblioteki standardowej języka Java. Implementacja ECBC-MAC została wykonana w ramach pracy w oparciu o implementację CBC dostarczoną przez bibliotekę standardową. Użyte konfiguracje szyfrów to *AES/CBC/NoPadding* i *AES/ECB/NoPadding* do obliczania sygnatury oraz *AES/CBC/PKCS7Padding* do szyfrowania i deszyfrowania.

W oparciu o tak stworzoną bibliotekę w języku Java napisano przykładową aplikację na platformę Android. Aplikacja po uruchomieniu stara się nawiązać połączenie Bluetooth z urządzeniem o nazwie



Rys. 4.1. Zrzut ekranu z przykładowej aplikacji implementującej protokół na platformę Android

„seconn". Po nawiązaniu połączenia Bluetooth wywoływana jest metoda biblioteki służąca nawiązaniu bezpiecznego połączenia (wysłaniu pierwszego HelloRequest). Wszystkie zmiany stanu połączenia są na bieżąco wyświetlane na ekranie, a po nawiązaniu bezpiecznego połączenia wyświetlane są klucze publiczne obu węzłów i możliwe jest przesyłanie uwierzytelnionych i zaszyfrowanych danych do drugiego węzła.

Przykładowa aplikacja mobilna skutecznie połączyła się z urządzeniem AVR, a przesyłane dane były prawidłowo deszyfrowane, co potwierdza że implementacja algorytmów z zewnętrzych bibliotek AVR oraz implementacje wykonane w ramach pracy są zgodne z referencyjnymi implementacjami dostępnymi w bibliotece standardowej języka Java na platformie Android. Zrzut ekranu z aplikacji przedstawiono na rysunku 4.1.

Podsumowanie

?

Bibliografia

[1] *ATtiny4 / ATtiny5 / ATtiny9 / ATtiny10 - Datasheet Summary*. Atmel. 2016. URL: *http://www.atmel.com/Images/Atmel-8127-AVR-8-bit-Microcontroller-ATtiny4-ATtiny5-ATtiny9-ATtiny10_Datasheet-Summary.pdf* (dostęp dnia 2016-12-06).

[2] *ATmega16U4/ATmega32U4 - Datasheet*. Atmel. 2016. URL: *http://www.atmel.com/Images/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Datasheet.pdf* (dostęp dnia 2016-12-06).

[3] *ATxmega384C3 - Datasheet*. Atmel. 2016. URL: *http://www.atmel.com/Images/Atmel-8361-8-and-16-bit-AVR-XMEGA-Microcontrollers-ATxmega384C3_Datasheet.pdf* (dostęp dnia 2016-12-06).

[4] *Recommendation for Key Management Part 1: General*. National Institute of Standards and Technology, U.S. Department of Commerce. 2016. URL: *http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4* (dostęp dnia 2016-12-07).

[5] Abdullah Al Hasib i Abul Ahsan Md Mahmudul Haque. „A comparative study of the performance and security issues of AES and RSA cryptography". W: *Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on*. T. 2. IEEE. 2008, s. 505–510.

[6] Nils Gura i in. „Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs". W: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Wyed. Marc Joye i Jean-Jacques Quisquater. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, s. 119–132. ISBN: 978-3-540-28632-5. DOI: *10.1007/978-3-540-28632-5_9*.

[7] Hugo Krawczyk, Ran Canetti i Mihir Bellare. „HMAC: Keyed-hashing for message authentication". W: (1997).

[8] John Black. „Authenticated encryption". W: *Encyclopedia of Cryptography and Security*. Springer, 2011, s. 52–61.

[9] Cameron F Kerry. „Digital Signature Standard (DSS)". W: *National Institute of Standards and Technology* (2013).

[10] Sean Turner i in. „Elliptic Curve Cryptography Subject Public Key Information". W: (2009).

[11] Burt Kaliski. „Pkcs# 7: Cryptographic message syntax version 1.5". W: (1998).

Dodatek A. Przykładowe wiadomości protokołu komunikacji

Tabela A.1. Budowa wiadomości typu HelloRequest

```
# Nagłówek:
0x00 0x01 # wersja protokołu
0x00      # typ wiadomości: HelloRequest
0x00 0x40 # długość zawartości: 64 bajty

# Zawartość:
# 32 bajty współrzędnej X klucza publicznego
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0

# 32 bajty współrzędnej Y klucza publicznego
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
```

Tabela A.2. Budowa wiadomości typu HelloResponse

```
# Nagłówek:
0x00 0x01 # wersja protokołu
0x01      # typ wiadomości: HelloResponse
0x00 0x60 # długość zawartości: 96 bajtów

# Zawartość:
# 16 bajtów kodu uwierzytelniającego
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0

# Zaszyfrowany klucz publiczny (64 bajty)
# z uwzględnieniem dopełnienia PKCS#7 (16 bajtów)
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
```

Tabela A.3. Budowa wiadomości typu EncryptedData

```
# Nagłówek:
0x00 0x01 # wersja protokołu
0x02      # typ wiadomości: EncryptedData
0x00 0x20 # długość zawartości: 32 bajty

# Zawartość:
# 16 bajtów kodu uwierzytelniającego
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0

# Zaszyfrowane dane (dopełnione do pełnego bloku AES)
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
0x12 0x34 0x56 0x78 0x9A 0xBC 0xDE 0xF0
```

Dodatek B. Przykładowe bloki kodu biblioteki

Tabela B.1. Szyfrowanie CBC wraz z obsługą dopełnienia PKCS#7

```
size_t _seconn_crypto_encrypt(void *destination, void *source, size_t length,
    aes128_key_t enc_key) {
    uint8_t *dest = (uint8_t*)destination;
    uint8_t *src = (uint8_t*)source;

    rng(dest, 16); // random initialization vector
    memset(&ctx, 0, sizeof(aes128_ctx_t));

    aes128_init(enc_key, &ctx);

    aes128_enc(dest, &ctx);

    size_t i = 0;
    for(; i+16 <= length; i += 16) {
        memcpy(dest+16+i, src+i, 16);
        _seconn_crypto_xor_block(dest+16+i, dest+i);
        aes128_enc(dest+16+i, &ctx);
    }

    size_t pad_length = 16 - (length % 16);
    memset(dest+16+i, pad_length, 16);
    memcpy(dest+16+i, src+i, length - i);
    _seconn_crypto_xor_block(dest+16+i, dest+i);
    aes128_enc(dest+16+i, &ctx);

    return i+32;
}
```

Tabela B.2. Odszyfrowanie CBC wraz z obsługą dopełnienia PKCS#7

```
size_t _seconn_crypto_decrypt(void *destination, void *source, size_t length,
    aes128_key_t enc_key) {
    uint8_t *src = ((uint8_t*)source);
    uint8_t *dest = (uint8_t*)destination;

    memset(&ctx, 0, sizeof(aes128_ctx_t));
    aes128_init(enc_key, &ctx);

    size_t i = 0;
    for(; i+16 < length; i += 16) {
        memcpy(dest+i, src+i+16, 16);
        aes128_dec(dest+i, &ctx);
        _seconn_crypto_xor_block(dest+i, src+i);
    }

    size_t pad_length = dest[i-1];
    for(size_t j = 2; j <= pad_length; j++) {
        if (dest[i-j] != pad_length) {
            return 0;
        }
    }

    return i-pad_length;
}
```

Tabela B.3. Obliczanie MAC dla wiadomości

```
void _seconn_crypto_calculate_mac(uint8_t *mac, void *message,
size_t length, aes128_key_t mac_key, aes128_key_t enc_key) {
    memset(&ctx, 0, sizeof(aes128_ctx_t));
    aes128_init(mac_key, &ctx);

    uint8_t *block = mac;

    memset(block, 0, 16); // initialization vector filled with zeros

    // calculating last block of CBC encryption
    size_t i = 0;
    for(; i+16 <= length; i += 16) {
        _seconn_crypto_xor_block(block, ((uint8_t*)message)+i);
        aes128_enc(block, &ctx);
    }

    // encrypting last block with separate key
    memset(&ctx, 0, sizeof(aes128_ctx_t));
    aes128_init(enc_key, &ctx);
    aes128_enc(block, &ctx);
}
```

Dodatek C. Uzyskiwanie liczb losowych

Biblioteka zaimplementowana w ramach pracy nie posiada żadnego źródła liczb losowych, mimo że jest ono potrzebne do jej prawidłowego funkcjonowania. Użycie biblioteki wymaga więc dostarczenia źródła liczb losowych przez osobę używającą biblioteki.

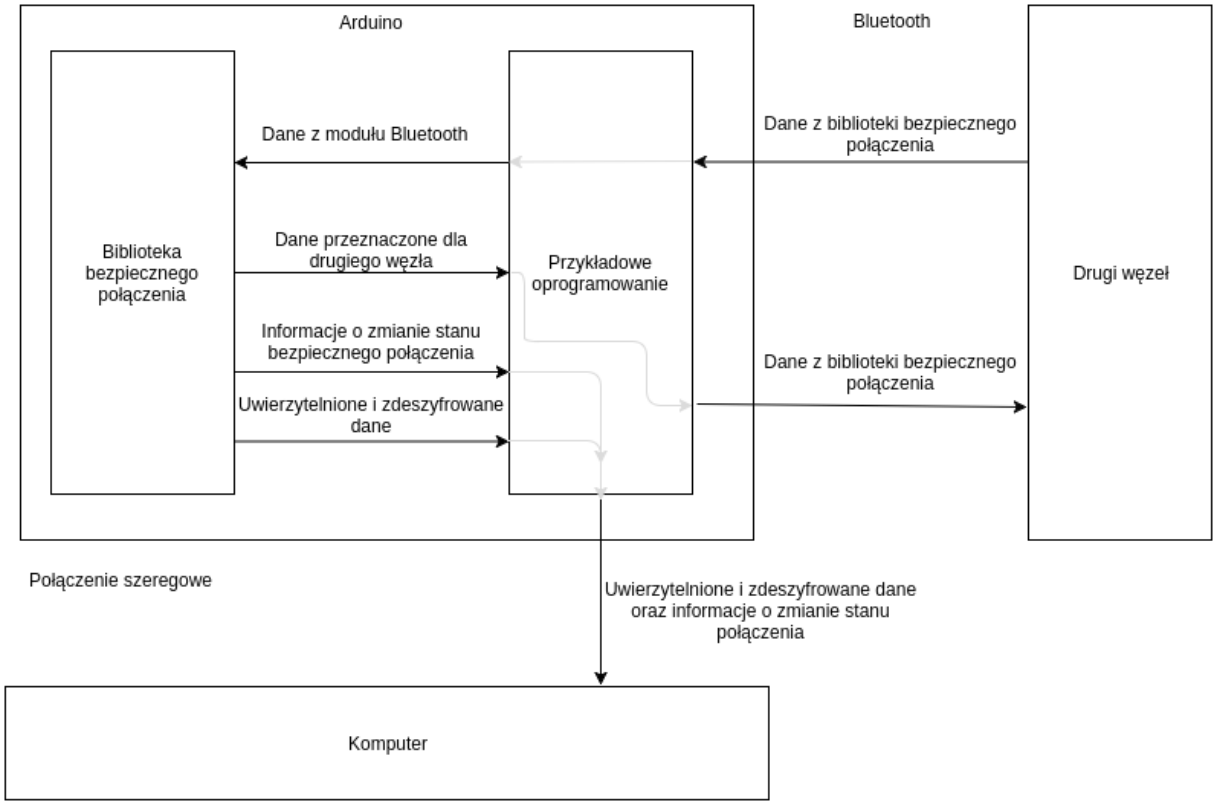
Najbezpieczniejszą metodą jest użycie dedykowanego, zewnętrznego generatora liczb losowych, lecz możliwe jest też wykorzystanie modułów dostępnych standardowo w mikropocesorach AVR. W bibliotece micro-ecc proponowana jest metoda wykorzystująca szum na niepodłączonym przetworniku analogowo-cyfrowym. Jej implementacja zaprezentowana jest w tabeli C.1.

Tabela C.1. Generowanie liczb losowych w oparciu o wbudowany przetwornik analogowo-cyfrowy. Źródło: biblioteka micro-ecc

```
static int RNG(uint8_t *dest, unsigned size) {
  // Use the least-significant bits from the ADC for an unconnected pin (or
    connected to a source of
  // random noise). This can take a long time to generate random data if the result
    of analogRead(0)
  // doesn't change very frequently.
  while (size) {
    uint8_t val = 0;
    for (unsigned i = 0; i < 8; ++i) {
      int init = analogRead(0);
      int count = 0;
      while (analogRead(0) == init) {
        ++count;
      }

      if (count == 0) {
         val = (val << 1) | (init & 0x01);
      } else {
         val = (val << 1) | (count & 0x01);
      }
    }
    *dest = val;
    ++dest;
    --size;
  }
  return 1;
}
```

Dodatek D. Przepływ danych w przykładowym oprogramowaniu



Rys. D.1. Przepływ danych między komponentami w przykładowym oprogramowaniu