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

AVR to rodzina mikroprocesorów opracowana i rozwijana przez firmę Atmel. Oparta o nią jest między innymi platforma Arduino, która – jak przedstawiono na rysunku 1 – z roku na rok zyskuje popularność. Platforma Arduino zaprojektowana została z myślą o projektach tworzonych nie tylko przez inżynierów, lecz także artystów i projektantów [1]. Jest ona też często używana do prototypowania urządzeń wpisujących się w koncepcję Internetu Rzeczy *(ang. Internet of Things, IoT)*.

Rys. 1. Relatywna liczba wyszukiwań frazy „Arduino” w ostatnich pięciu latach. Źródło: Google Trends

Urządzenia wbudowane podłączone do Internetu są szczególnie narażone na ataki. W 2016 roku podatne urządzenia wbudowane zostały wykorzystane do przeprowadzenia masywnych ataków typu DDoS [2]. Zagrożone są też rozwiązania oparte o komunikację bezprzewodową jak Wi-Fi oraz Bluetooth. Istotne jest więc dostarczenie narzędzi, które pozwalają nie tylko na szybkie prototypowanie, ale także na zachowanie bezpieczeństwa takich rozwiązań jak bezprzewodowe tokeny, czujniki, sprzętowe menadżery haseł czy inteligentne domy.

W niniejszej pracy przedstawiono protokół bezpiecznej komunikacji oraz bibliotekę programistyczną zaprojektowane z myślą o prostocie integracji. Wybrane zostały zestawy algorytmów, które zapewniają

bezpieczeństwo komunikacji. Ich złożoność została ukryta za interfejsem programistycznym, który ogranicza możliwość wprowadzenia błędów zmniejszających bezpieczeństwo. Zaproponowane rozwiązanie zapewnia poufność, autentyczność oraz integralność przesyłanych danych.

W rozdziale 1 przedstawione zostały różne metody uwierzytelniania i uzasadniony został wybór konkretnych algorytmów. Implementacja została szczegółowo opisana w rozdziale 2. Całość została zweryfikowana poprzez strzorzenie przykładowego oprogramowania oraz porównanie z implementacją na inną platformę, co opisano w rozdziale 3. W podsumowaniu przedstawiono całe rozwiązanie oraz jego ograniczenia i słabe strony.

Całość kodu źródłowego dostępna jest na licencji MIT w serwisie GitHub[1].

[1] *https://github.com/kacperzuk/seconn*

1 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 [3]. Klucze w przypadku kryptografii symetrycznej nie muszą mieć konkretnych właściwości poza ich nieprzewidywalnością i zapewniają porównywalne bezpieczeństwo przy krótszych kluczach. Kluczowi RSA o długości 2048 bitów odpowiada klucz 112 bitowy dla szyfrów symetrycznych.

Ważną różnicą jest też wydajność. Kryptografia asymetryczna jest dużo bardziej złożona obliczeniowo od symetrycznej [4]. 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 [4].

1.1. 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 bezpieczeństwa na co najmniej 5 lat.

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

[1] *https://trac.cryptolib.org/avr-crypto-lib/browser*
[2] *http://www.emsign.nl/*

zapewnienia bezpieczeństwa. Komercyjna biblioteka *LightCrypt-AVR8-ECC* oraz biblioteka *micro-ecc* dostarczają implementację kryptografii opartej o krzywe eliptyczne[3]. Brak jest na rynku implementacji 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 [5].

Krzywe eliptyczne wymagają najkrótszych kluczy. Rekomendacje NIST [3] (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 i że taki klucz wystarczy do 2030 roku.

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)*.

1.2. Kryptografia symetryczna

Przy wyborze algorytmu dla potrzeb pracy istotne były:

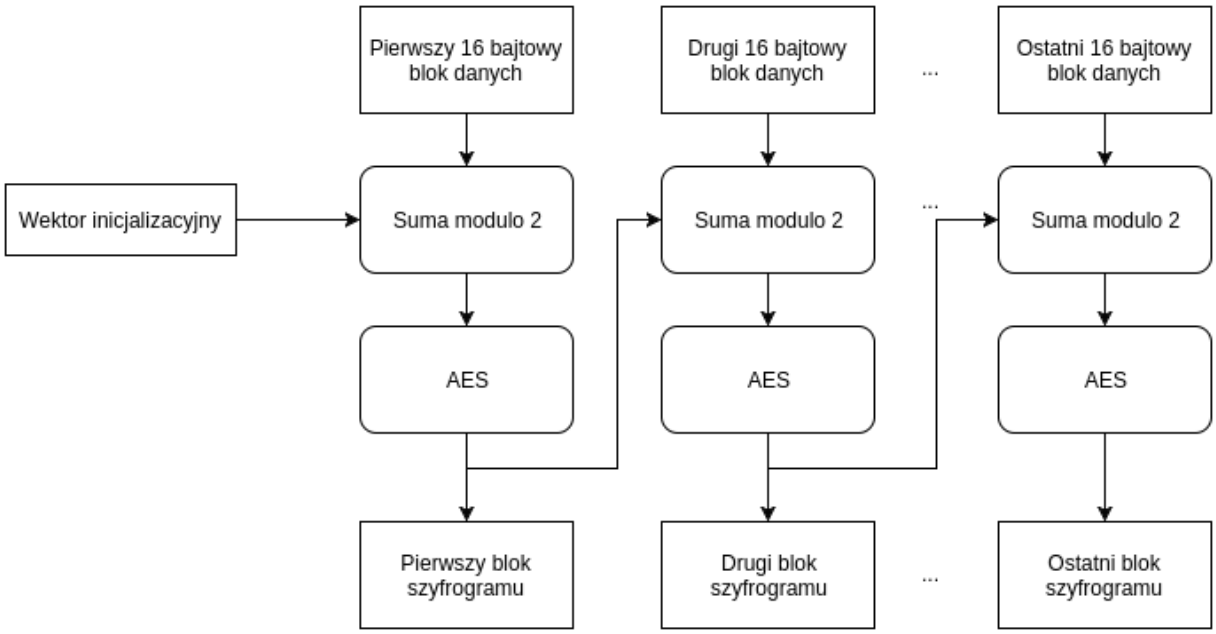
– jakość implementacji dostępnych na mikroprocesory AVR,

– 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 należy 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.

W trybie CBC z każdego 16 bajtowego blok danych oraz szyfrogramu bloku poprzedzającego liczona jest suma modulo 2, co przedstawiono na rysunku 1.1. Wektor inicjalizacyjny jest stosowany, by dwie wiadomości o identycznym pierwszym 16 bajtowym bloku po zaszyfrowaniu nie miały identycznego pierwszego bloku szyfrogramu. By wektor inicjalizacyjny poprawnie pełnił taką rolę, musi być losowy i przesyłany do odbiorcy. Nie musi być on tajny. W przypadku trybu CBC-MAC używany jest jedynie ostatni blok szyfrogramu, a więc wektor inicjalizacyjny nie jest potrzebny. Zwyczajowo korzysta się więc z wektora inicjalizacyjnego wypełnionego zerami.

[3] *http://industrial.crypto.cmmsigma.eu/lightcrypt_avr8/lc_avr8_ecc.pl.html*



Rys. 1.1. Schemat działania trybu CBC.

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 [6].

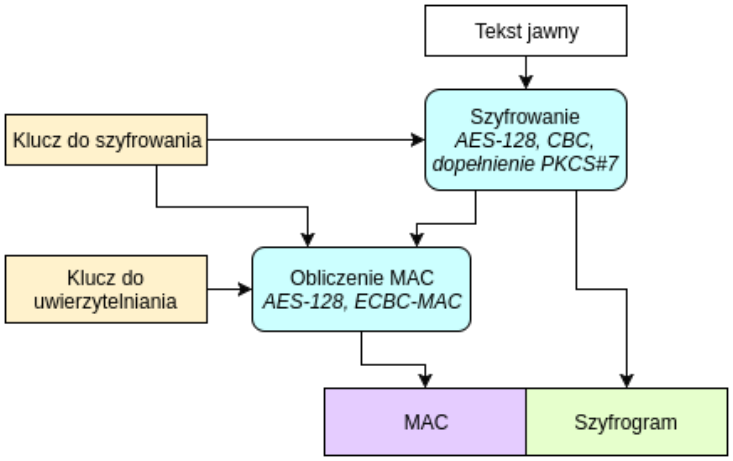
W pracy do uwierzytelniania wybrano AES w trybie ECBC-MAC. Zaletą tego rozwiązania jest możliwość użycia tej samej implementacji trybu CBC zarówno do szyfrowania jak i jako element trybu ECBC-MAC.

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 dla wiadomości o długości niebędącej wielokrotnością długości bloku wymagane jest dopełnienie. Oznacza to że do szyfrowanej wiadomości należy 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* [7]. Całość procesu została przedstawiona na rysunku 1.2.



Rys. 1.2. Proces szyfrowania i uwierzytelniania danych

2 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.

2.1. Charakterystyka platformy sprzętowej

Mikropocesory Atmel AVR są w większości 8-bitowe i na takich skupia się praca. Rodzina AVR jest szeroka, kilka wybranych modeli przedstawiono w tabeli 2.1. W pracy wykorzystany został model ATmega32u4 z 2,5 kilobajta SRAM [8].

Tabela 2.1. Wybrane modele AVR wraz z ich parametrami

Nazwa	**SRAM**[1]	**Wymagane napięcie**	**Taktowanie procesora**	**Liczba linii I/O**
ATtiny4 [9]	32 B	1.8 - 5.5 V	do 12 MHz	4
ATmega32u4 [8]	2,5 KB	2.7 - 5.5 V	do 16 MHz	26
ATxmega384C3 [10]	32 KB	1.6 - 3.6 V	do 32 MHz	50

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.

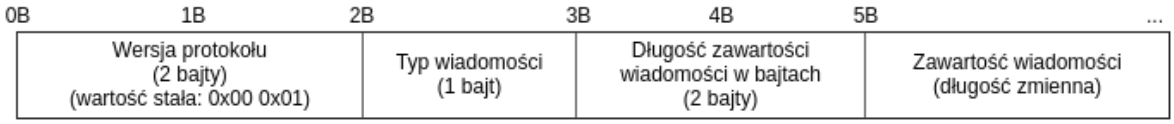
Istotnym elementem jest też wielkość domyślnych buforów. *Arduino* w modułach *Serial* oraz *SoftwareSerial* domyślnie używa 16- lub 64-bajtowego (w zależności od ilości dostępnej pamięci) buforu na przychodzące dane[2]. Przy wiadomościach dłuższych niż 32 bajty oznacza to, że zbyt długie przetwarzanie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

[1] ang. Static Random Access Memory

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

2.2. Podstawowe struktury protokołu

Podstawową jednostką protokołu są wiadomości zbudowane według schematu zaprezentowanego na Rys. 2.1. Zdefiniowane typy wiadomości są przedstawione w tabeli 2.2. Budowę przykładowych wiadomości przedstawiono na rysunkach A.1, A.2 oraz A.3 w dodatku A.



Rys. 2.1. Budowa wiadomości w protokole komunikacji.

Tabela 2.2. Typy wiadomości wraz z ich charakterystyką

Typ wiadomości	**Wartość pola typ**	**Długość bloku danych**	**Blok danych jest zaszyfrowany**	**Blok danych jest uwierzytelniony**
HelloRequest	0x00	64 bajty	nie	nie
HelloResponse	0x01	96 bajtów	tak	tak
EncryptedData	0x02	zmienna, minimum 32 bajty	tak	tak

Odbiorca powinien zweryfikować:

– zgodność wersji protokołu – wymagane bajty 0x00 oraz 0x01,

– prawidłowość bajtu określającego typ – wymagana wartość 0x00, 0x01 lub 0x02,

– zgodność zadeklarowanej długości bloku danych z typem,

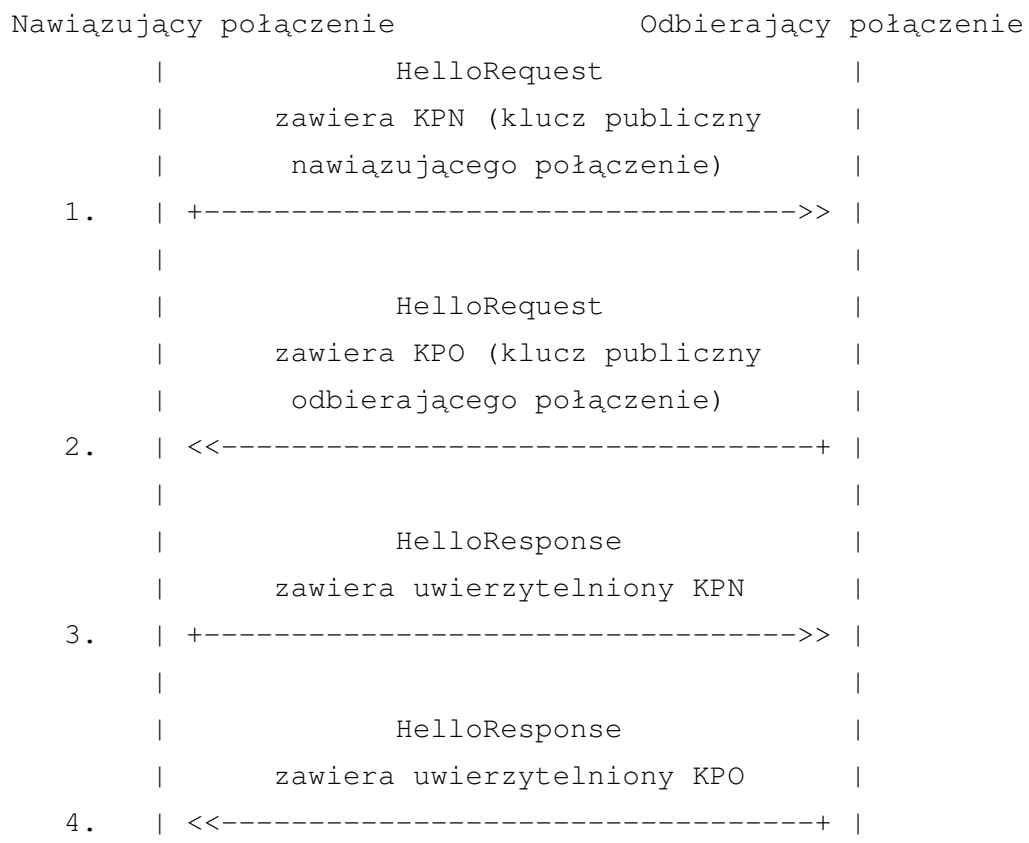
– w przypadku typów HelloResponse oraz EncryptedData – prawidłowość kodu uwierzytelniającego.

W przypadku niezgodności któregokolwiek elementu wiadomość powinna zostać zignorowana.

Narzut pamięci operacyjnej implementacji protokołu wynosi około 2 KB. Narzut na rozmiar programu to 14.6 KB.

2.3. Nawiązywanie połączenia

Kolejność przesyłania wiadomości w celu nawiązania połączenia przedstawiona została na rysunku 2.2. 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.



Rys. 2.2. Kolejność wymiany wiadomości w procesie nawiązywania połączenia

Po ustaleniu wspólnego klucza węzły mogą wysłać HelloResponse, który zawiera zaszyfrowany i uwierzytelniony klucz publiczny pochodzący z wysyłającego węzła. 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.

2.4. 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 *(ang. Elliptic curve Diffie–Hellman)* oraz proponowanej przez NIST krzywej eliptycznej P-256 [11] (w RFC 5480 nazwaną krzywą secp256r1 [12]).

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. Wygenerowanie wspólnego sekretu na mikroprocesorze ATmega32u4 trwa do 4350 ms, nie uwzględniając generowania liczb losowych, używanych przez bibliotekę *micro-ecc* do zapobiegania atakom typu *side-channel*. Przy uwzględnieniu generowania liczb losowych metodą opisaną w Dodatku C czas ten rośnie do 4500ms.

2.5. Szyfrowanie i deszyfrowanie danych

Szyfrowanie bloku danych w wiadomości odbywa się za pomocą szyfru blokowego AES ze 128-bitowym kluczem używanym w trybie CBC. Wektor inicjalizacyjny jest losowy i dołączany do danych przesyłanej wiadomości przed szyfrogramem. Tekst jawny jest dopełniany do pełnego bloku według algorytmu zdefiniowanego w PKCS#7 [13].

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 danych 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 2.3.

Kod implementujący szyfrowanie danych przedstawiono w tabeli B.1 w dodatku B. Zaszyfrowanie 32 bajtów danych na mikroprocesorze ATmega32u4 trwa do 8 ms bez uwzględnienia czasu potrzebnego

Tabela 2.3. Dopełnanie danych do pełnego bloku. Dopełnienie zaznaczone zostało kolorem niebieskiem i pogrubieniem.

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
```

do wygenerowania losowego wektora inicjalizacyjnego. Przy generowaniu losowych danych metodą opisaną w Dodatku C czas ten rośnie do 64 ms.

Właściwe kroki potrzebne do odszyfrowania danych 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. Zdeszyfrowanie 32 bajtów danych na mikroprocesorze ATmega32u4 trwa do 8 ms.

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.

2.6. Uwierzytelnienie wiadomości

Uwierzytelnienie wiadomości odbywa się poprzez dołączenie MAC do bloku danych. Dla danej wiadomości MAC 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. Wygenerowanie kodu uwierzytelniającego dla 32 bajtów danych na mikroprocesorze ATmega32u4 trwa do 8 ms.

Implementacja trybu ECBC-MAC została zrealizowana w ramach pracy.

2.7. Złożoność implementacji

Stworzona biblioteka implementująca protokół posiada cztery zależności:

– moduł *AES* z biblioteki *AVR-Crypto-Lib*,

– moduł *gf256mul* wymagany do obliczeń na dużych liczbach z biblioteki *AVR-Crypto-Lib*,

– moduł *SHA-256* z biblioteki *AVR-Crypto-Lib*,

– biblioteka *micro-ecc*.

Za pomocą programu *ctags* obliczono, że całość biblioteki wraz z zależnościami składa się ze 131 funkcji oraz 13 struktur. Dla kodu stworzonego w ramach pracy – po odliczeniu zależności – liczba funkcji to 13 a struktur to 5. Dodatkowe statystyki dotyczące linii kodu zaprezentowano w tabelach 2.4a

oraz 2.4b, odpowiednio uwzględniając i pomijając zależności. Statystyki te wygenerowano przy pomocy programu *cloc*.

Tabela 2.4. Statystyki złożoności kodu źródłowego

Typ pliku	**Liczba plików**	**Liczba linii komentarzy**	**Liczba linii kodu**
Nagłówek C/C++	24	921	2466
C	13	402	1772
C++	3	11	303
Asembler	1	49	39
Suma	**41**	**1383**	**4580**

(a) Statystyki złożoności kodu źródłowego biblioteki wraz z zależnościami.

Typ pliku	**Liczba plików**	**Liczba linii komentarzy**	**Liczba linii kodu**
Nagłówek C/C++	3	83	98
C++	3	11	303
Suma	**6**	**94**	**401**

(b) Statystyki złożoności kodu źródłowego biblioteki z wyłączeniem zależności.

3 Test opracowanej biblioteki

Zweryfikowane zostały trzy aspekty:

– poprawność zaprojektowania interfejsu programistycznego stworzonej biblioteki,

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

– poprawność implementacji algorytmów kryptograficznych.

3.1. Poprawność interfejsu programistycznego

Poprawny interfejs programistyczny biblioteki musi umożliwiać implementację 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. Argumentami funkcji inicjalizującej bibliotekę są wskaźniki na 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 3.1. 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 węzłem 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. Całość oprogramowania wraz z funkcją generującą liczby losowe składa się ze 103 linii kodu, ośmiu funkcji i trzech plików.

W tabeli 3.1 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...”.

3.2. Poprawność implementacji algorytmów kryptograficznych

W trakcie tworzenia implementacje algorytmów były weryfikowane poprzez porównywanie przykładowych zestawów danych wejściowych i wejściowych z niezależnymi implementacjami.

Algorytm AES oraz tryb CBC zostały przetestowane przy użyciu aplikacji internetowej *OnlineDomainTools*[1]. Ze względu na brak dokumentacji dotyczącej dopełniania do pełnych bloków w aplikacji *OnlineDomainTools*, dane wejściowe podawano już dopełnione. Zweryfikowano:

– zgodność tekstu jawnego po zaszyfrowaniu na urządzeniu AVR i zdeszyfrowaniu w aplikacji,

[1] http://aes.online-domain-tools.com/ [dostęp 5.01.2017]

– zgodność tekstu jawnego po zaszyfrowaniu w aplikacji i zdeszyfrowaniu na urządzeniu AVR,

– zgodność szyfrogramów po zaszyfrowaniu tekstu jawnego w aplikacji oraz na urządzeniu AVR przy zastosowaniu tego samego wektora inicjalizacyjnego.

Wszystkie powyższe testy wykonano zarówno dla danych wejściowych o długości jednego jak i wielu bloków.

Poprawność ustalenie wspólnego sekretu algorytmem ECDH sprawdzono poprzez porównanie sekretu obliczonego na urządzeniu z sekretem obliczonym przez aplikację internetową *JavaScript ECDH Key Exchange Demo*[2]. Operacje dokonywano na kluczach wygenerowanych na urządzeniu AVR przez bibliotekę *micro-ecc*.

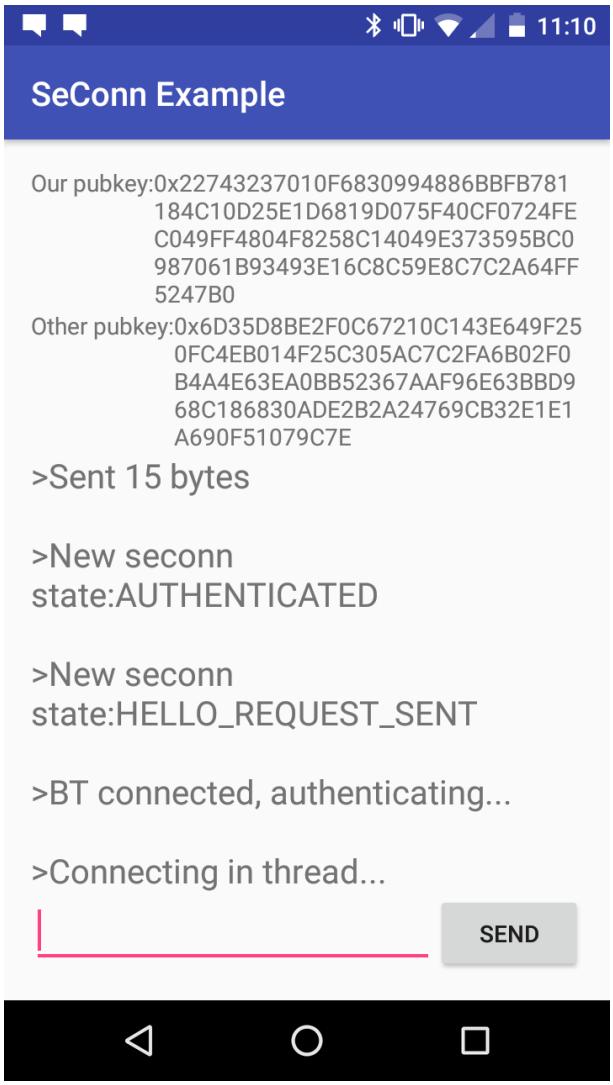
Poprawność implementacji algorytmów CBC i dopełniania według PKCS#7 stworzonych w ramach pracy oraz algorytmów AES, SHA-256 oraz ECDH dostarczonych przez zewnętrzne biblioteki została także zweryfikowana poprzez stworzenie drugiej implementacji protokołu w języku Java.

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 „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 3.1.

[2]http://www-cs-students.stanford.edu/ tjw/jsbn/ecdh.html [dostęp 5.01.2017]



Rys. 3.1. Zrzut ekranu z przykładowej aplikacji implementującej protokół na platformę Android

Podsumowanie

Rodzina mikroprocesorów AVR jest często używana do prototypowania urządzeń, które przesyłają poufne dane lub odbierają zdalne komendy, takich jak bezprzewodowe tokeny, sensory oraz inteligentne domy. Takie urządzenia powinny być zabezpieczone zarówno przed podsłuchiwaniem informacji jak i przed wykonywaniem nieupoważnionych poleceń.

W pracy przedstawiono rozwiązanie w postaci protokołu komunikacji oraz biblioteki programistycznej dla urządzeń AVR, które ma na celu zapewnienie poufności, autentyczności oraz integralności przesyłanych danych. Szczególny nacisk postawiono na łatwość integracji.

Zaprezentowano różne metody uwierzytelniania i wybrano algorytmy ECDH *(ang. Elliptic Curve Diffie–Hellman)* oraz AES użytym w trybie ECBC-MAC *(ang. Encrypt-last-block Cipher Block Chaining - Message Authentication Code)*. Za ich przewagę nad innymi rozwiązaniami uznano dużą wydajność, niskie wymagania dotyczące pamięci operacyjnej oraz jakość implementacji dostępnych na urządzenia AVR. Dodatkowo w celu zapewnienia poufności danych wybrano szyfrowanie algorytmem AES użytym w trybie CBC. Ograniczeniem tej metody jest brak utajnienia przekazywania *(ang. forward secrecy)*, a więc ujawnienie długoterminowego klucza prywatnego dowolnego węzła pozwala na odszyfrowanie całej przeszłej komunikacji.

todo implementacja i testy

Przed wdrożeniem rozwiązania powinnien zostać przeprowadzony pełen audyt bezpieczeństwa zarówno protokołu jak i jego implementacji. Prawidłowy audyt powinien zostać przeprowadzony przez niezależny zespół, który nie był powiązany z projektem protokołu oraz stworzeniem biblioteki, więc wykracza on poza zakres tej pracy.

Bibliografia

[1] Massimo Banzi and Michael Shiloh. *Getting Started with Arduino: The Open Source Electronics Prototyping Platform*. Sebastopol: Maker Media, Inc., 2014.

[2] Martin McKeay i in. *Q3 2016 State of the Internet Security Report*. Spraw. tech. Akamai Technologies, Inc., 2016.

[3] *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).

[4] 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.

[5] 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*.

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

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

[8] *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).

[9] *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).

[10] *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).

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

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

[13] 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
0x1F 0x92 0xDE 0xFF 0x6B 0xEE 0xFC 0x4D
0x51 0x2A 0x62 0xF4 0x60 0x1D 0x36 0x73
0x6F 0xEB 0x3F 0x1F 0x56 0x90 0xFD 0x85
0xB0 0x3C 0x56 0xD0 0xC0 0x52 0x6E 0x9B

# 32 bajty współrzędnej Y klucza publicznego
0xE8 0x60 0x84 0xB4 0xDE 0x73 0x65 0xB2
0x48 0xA6 0x15 0x79 0x7C 0xD9 0x4C 0xB6
0x56 0xE6 0xFA 0x3C 0x2F 0x3C 0x1C 0x8F
0xB2 0xE6 0x25 0xF8 0x66 0x2A 0x00 0xE4
```

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
0x99 0x7B 0x57 0x10 0x7F 0x9C 0x1E 0xB3
0x1C 0x92 0x1B 0xFC 0x99 0x0C 0xCF 0x7D

# Zaszyfrowany klucz publiczny (64 bajty)
# z~uwzględnieniem dopełnienia PKCS#7 (16 bajtów)
0xF2 0x8A 0x7B 0xD4 0x04 0x80 0xAE 0xDC
0x7A 0x1D 0x04 0xEE 0x98 0x6D 0x9F 0xC5
0x22 0x4B 0x92 0x48 0x3E 0x65 0x79 0xC7
0xCB 0xCA 0xA9 0xC0 0xA1 0x7D 0x35 0x1F
0xFD 0x6F 0xE4 0x9E 0x62 0xBE 0x3F 0x1E
0xAC 0x32 0x03 0xF5 0x50 0x15 0xBE 0x0F
0x84 0xB9 0xF4 0xB6 0xF7 0x36 0x1D 0x3E
0x2D 0xD5 0xE3 0x10 0xBE 0xC4 0x39 0xC7
0x98 0x86 0x65 0x46 0x65 0xA9 0x5D 0xDE
0x4C 0x9D 0x03 0x00 0x55 0xE9 0xAF 0xC2
```

Tabela A.3. Budowa wiadomości typu EncryptedData

```
# Nagłówek:
0x00 0x01 # wersja protokołu
0x02      # typ wiadomości: EncryptedData
0x00 0x30 # długość zawartości: 48 bajtów

# Zawartość:
# 16 bajtów kodu uwierzytelniającego
0x6B 0xE7 0xBD 0xB6 0x23 0x98 0x1A 0x35
0xFF 0xBE 0xBB 0x38 0x3F 0xB1 0xF9 0x56

# Zaszyfrowane dane (dopełnione do~pełnego bloku AES)
0xD2 0x53 0xFF 0x58 0x99 0x94 0x36 0x24
0x1D 0x6B 0x4D 0x41 0x45 0xEF 0xD3 0x91
0x26 0x09 0xAB 0x91 0xC0 0x27 0x7C 0xAC
0x8E 0xFA 0xDA 0x24 0x9F 0xC6 0x22 0xBD
```

Dodatek B. Przykładowe bloki kodu biblioteki

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

```
size_t _seconn_crypto_encrypt(uint8_t *dest, uint8_t *src, size_t length,
    aes128_key_t enc_key) {
    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(uint8_t *dest, uint8_t *src, size_t length,
    aes128_key_t enc_key) {
    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, uint8_t *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, 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.

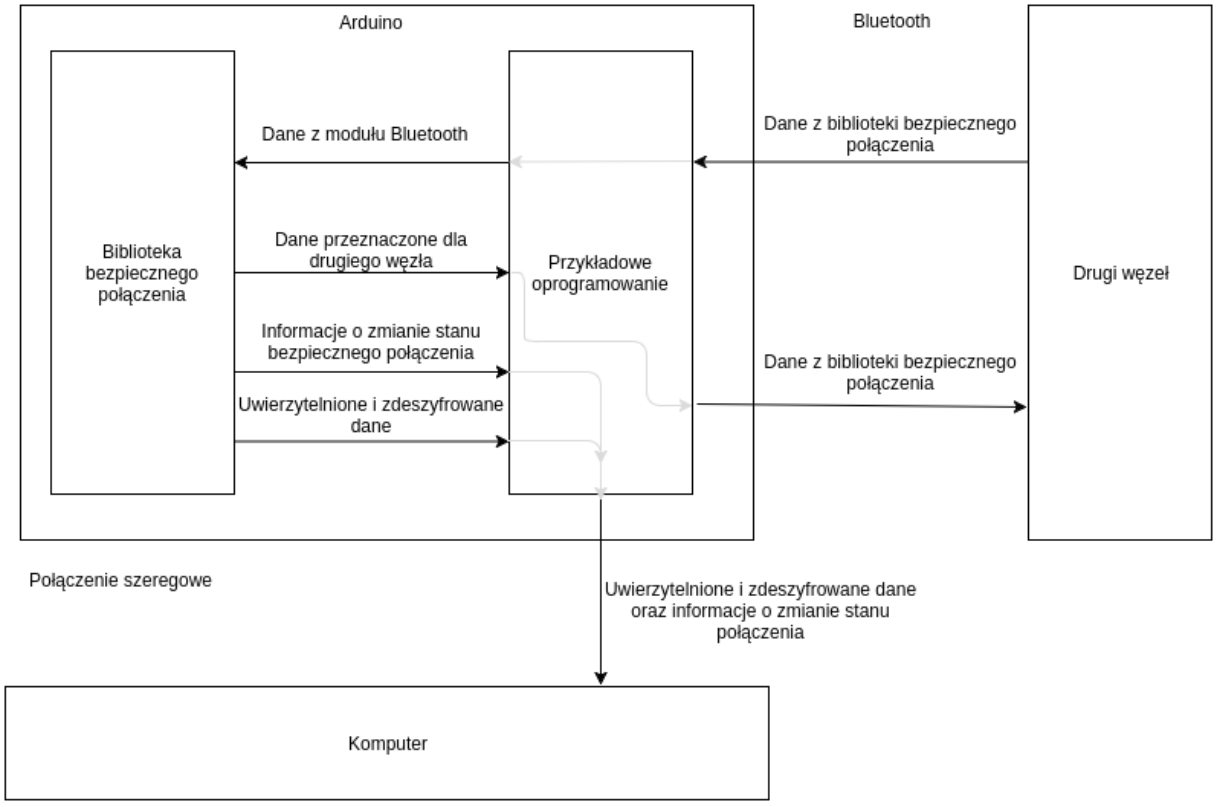
Najbezpieczniejszą metodą jest użycie dedykowanego, zewnętrznego generatora liczb losowych, lecz możliwe jest też wykorzystanie modułów dostępnych standardowo w mikropocesorach AVR. W bibliotece micro-ecc proponowana jest metoda wykorzystująca szum na niepodłączonym przetworniku analogowo-cyfrowym. Jej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

Dodatek E. Dokumentacja biblioteki

E.1. Przykład użycia

W tabeli E.1 przedstawiono przykład użycia biblioteki. W przykładzie założono istnienie funkcji `RNG` służącej generowaniu losowych danych oraz funkcji `network_read` oraz `network_write`, które odpowiednio odczytują i wysyłają dane z i do sieci.

Tabela E.1. Przykład użycia biblioteki

```
#include <seconn.h>

struct seconn sconn;

int c_seconn_write_data(void *src, size_t bytes) {
    return network_write(src, bytes);
}

void c_seconn_data_received(void *src, size_t bytes) {
    /*
     * src contains decrypted and authenticated data from other node that for
     * example could be shown to user
     */
}

void c_seconn_state_changed(seconn_state prev, seconn_state cur) {
    if(cur == AUTHENTICATED) {
        /*
         * sconn.public_key now contains authenticated public key of other
         * node. It's the best place to show it to user and ask for confirmation
         * that this is a~correct key.
         */
    }
}

int main() {
    // initialize sconn struct
    seconn_init(&sconn, c_seconn_write_data, c_seconn_data_received,
                c_seconn_state_changed, &RNG, 0);
```

```
    // fetch local public key, for example to show it to user
    uint8_t pubkey[64];
    seconn_get_public_key(&sconn, pubkey);

    // read data from network
    char buffer[64];
    unsigned bytes_read;
    while((bytes_read = network_read(buffer, bytes_read)) > 0) {
        // and pass it to seconn library.
        // seconn library will call functions passed to seconn_init if required
        seconn_new_data(&sconn, buffer, bytes_read);
    }
}
```

E.2. Dokumentacja Doxygen

Spis treści

Rozdział 1

Dokumentacja struktur danych

1.1 Dokumentacja struktury seconn

Pola danych

- **seconn_state state**

  Obecny stan połączenia.
- uint8_t **public_key** [512]
- int(∗ **writeData**)(void ∗src, size_t bytes)
- void(∗ **onDataReceived**)(void ∗src, size_t bytes)
- void(∗ **onStateChange**)(**seconn_state** prev_state, **seconn_state** cur_state)

1.1.1 Opis szczegółowy

Główna struktura opisująca bezpieczne połączenie.

Nie powinna być wypełniania ręcznie, lecz za pośrednictwem funkcji seconn_init.

1.1.2 Dokumentacja pól

1.1.2.1 onDataReceived

```
void(* seconn::onDataReceived) (void *src, size_t bytes)
```

Wskaźnik na funkcję, do której przekazywane będą uwierzytelnione i zdeszyfrowane dane pochodzące od drugiego węzła.

Pierwszy argument zawiera wskaźnik na początek danych, drugi argument zawiera liczbę bajtów.

1.1.2.2 onStateChange

```
void(* seconn::onStateChange) (seconn_state prev_state, seconn_state cur_state)
```

Wskaźnik na funkcję, do której przekazywane będą informacje o zmianie stanu połączenia.

Pierwszy argument zawiera poprzedni stan, drugi argument zawiera obecny stan.

1.1.2.3 public_key

```
uint8_t seconn::public_key[512]
```

Klucz publiczny drugiego węzła. Wypełniany dopiero po zmianie stanu połączenia na AUTHENTICATED.

Uwaga! Nie mylić z wartością zwracaną przez funkcję seconn_get_public_key, która zwraca lokalny klucz publiczny.

1.1.2.4 state

seconn_state `seconn::state`

Obecny stan połączenia.

1.1.2.5 writeData

```
int(* seconn::writeData) (void *src, size_t bytes)
```

Wskaźnik na funkcję, która zostanie wywołana przez bibliotekę, gdy zajdzie potrzeba przesłania danych do drugiego węzła.

Pierwszy argument zawiera wskaźnik na początek danych do przesłania, a drugi liczbę bajtów które powinny zostać przesłane.

Funkcja powinna zwrócić liczbę bajtów, które rzeczywiście udało się przesłać.

Rozdział 2

Dokumentacja plików

2.1 Dokumentacja pliku seconn/seconn.h

Struktury danych

- struct **seconn**

Wyliczenia

- enum **seconn_state** {
  NEW, **HELLO_REQUEST_SENT**, **INVALID_HANDSHAKE**, **SYNC_ERROR**,
  AUTHENTICATED }

Funkcje

- void **seconn_init** (struct **seconn** ∗conn, int(∗writeData)(void ∗src, size_t bytes), void(∗onData↩ Received)(void ∗src, size_t bytes), void(∗onStateChange)(**seconn_state** prev_state, **seconn_state** cur_↩ state), int(∗rng)(uint8_t ∗dest, unsigned size), int eeprom_offset)
- void **seconn_new_data** (struct **seconn** ∗conn, const void ∗data, size_t bytes)
- void **seconn_write_data** (struct **seconn** ∗conn, const void ∗source, size_t bytes)
- void **seconn_get_public_key** (struct **seconn** ∗conn, uint8_t ∗public_key)

2.1.1 Dokumentacja typów wyliczanych

2.1.1.1 seconn_state

`enum` **seconn_state**

Typ wyliczeniowy definiujący możliwe stany bezpiecznego połączenia.

Używany do określenia obecnego stanu połączenia w strukturze seconn oraz w wywołaniach fukcji onStateChange.

Wartości wyliczeń

NEW	żadne dane nie zostały jeszcze wysłane ani odebrane.
HELLO_REQUEST_SENT	wysłana została wiadomość HelloRequest
INVALID_HANDSHAKE	wystąpił problem w czasie nawiązywania połączenia, np. została odebrana nieprawidłowo uwierzytelniona wiadomość HelloResponse
SYNC_ERROR	odebrane zostały dane, które są niezgodne z protokołem lub zostały nieprawidłowo uwierzytelnione
AUTHENTICATED	połączenie zostało poprawnie nawiązane, uwierzytelniony został klucz publiczny drugiego węzła

2.1.2 Dokumentacja funkcji

2.1.2.1 seconn_get_public_key()

```
void seconn_get_public_key (
            struct seconn * conn,
            uint8_t * public_key )
```

Funkcja odczytująca lokalny klucz publiczny.

Pierwszym argumentem jest zainicjalizowana funkcją seconn_init struktura typu seconn.

Drugim argumentem jest wskaźnik na miejsce w pamięci, do którego ma zostać wpisany klucz publiczny. Wymagane są 64 bajty pamięci.

2.1.2.2 seconn_init()

```
void seconn_init (
            struct seconn * conn,
            int(*)(void *src, size_t bytes) writeData,
            void(*)(void *src, size_t bytes) onDataReceived,
            void(*)(seconn_state prev_state, seconn_state cur_state) onStateChange,
            int(*)(uint8_t *dest, unsigned size) rng,
            int eeprom_offset )
```

Funkcja inicjalizująca strukturę seconn. Należy ją wywołać jako pierwszą.

Pierwszy argument (writeData) zawiera wskaźnik na funkcję, która zostanie wywołana przez bibliotekę, gdy zajdzie potrzeba przesłania danych do drugiego węzła. Pierwszym argumentem tej funkcji (src) jest wskaźnik na początek danych do przesłania, a drugim (bytes) liczba bajtów które powinny zostać przesłane.

Drugim argumentem (onDataReceived) jest wskaźnik na funkcję, do której przekazywane będą uwierzytelnione i zdeszyfrowane dane pochodzące od drugiego węzła. Pierwszy argument tej funkcji (src) to wskaźnik na początek danych, drugi argument (bytes) zawiera liczbę bajtów.

Trzecim argumentem (onStateChange) jest Wskaźnik na funkcję, do której przekazywane będą informacje o zmianie stanu połączenia. Pierwszy argument (prev_state) zawiera poprzedni stan, drugi argument (cur_state) zawiera obecny stan.

Czwartym argumentem (rng) jest wskaźnik na funkcję, która będzie wywoływana przez bibliotekę w celu wygenerowania losowych danych. Pierwszym argumentem tej funkcji (desc) jest wskaźnik na miejsce w pamięci, do którego mają być wpisane losowe dany, drugim argumentem (size) jest liczba bajtów które powinny być wpisane. Funkcja powinna zwrócić wartość 1.

Ostatnim, piątym argumentem (eeprom_offset) jest miejsce w pamięci EEPROM do którego zapisywane i z którego odczytywane mają być lokalne klucze kryptograficzne. W tym miejscu powinno być 96 bajtów nieużywanej, ciągłej pamięci.

2.1.2.3 seconn_new_data()

```
void seconn_new_data (
            struct seconn * conn,
            const void * data,
            size_t bytes )
```

Funkcja którą należy wywołać w celu przekazania danych pochodzących od drugiego węzła do biblioteki.

Pierwszym argumentem jest zainicjalizowana funkcją seconn_init struktura typu seconn.

Drugim argumentem jest wskaźnik na początek danych pochodzących od drugiego węzła.

Trzecim argumentem jest liczba bajtów danych.

2.1.2.4 seconn_write_data()

```
void seconn_write_data (
            struct seconn * conn,
            const void * source,
            size_t bytes )
```

Funkcja którą należy wywołać w celu przesłania do drugiego węzłą danych. Dane te zostaną zaszyfrowane i uwierzytelnione, a następnie w postaci wiadomości EncryptedData przekazane do funkcji writeData ze struktury seconn.

Pierwszym argumentem jest zainicjalizowana funkcją seconn_init struktura typu seconn.

Drugim argumentem jest wskaźnik na początek danych, które mają zostać przesłane.

Trzecim argumentem jest liczba bajtów danych.