[KRYS] AEGIS-128

Krzysztof Kołcz Rafał Kulus Filip Stępkowski

27 stycznia 2023

Spis treści

1 Wstęp **2**

2 Specyfikacja **2**
2.1 Inicjalizacja . . . 2
2.2 Aktualizacja stanu . . . 3
2.3 Przetwarzanie uwierzytelnionych danych . . . 3
2.4 Szyfrowanie wiadomości . . . 3
2.5 Finalizowanie . . . 4
2.6 Odszyfrowanie i weryfikacja . . . 4

3 Ataki **4**
3.1 Ataki na inicjację . . . 4
3.1.1 Atak różnicowy . . . 4
3.2 Ataki na szyfrowanie . . . 4
3.2.1 Atak statystyczny . . . 4
3.2.2 Losowość klucza . . . 4
3.3 Ataki na uwierzytelnienie . . . 5
3.3.1 Odzyskanie klucza lub stanu . . . 5
3.3.2 Wewnętrzne kolizje . . . 5
3.4 Inne ataki . . . 5

4 Uzasadnienie bezpieczeństwa **5**

5 Implementacja **5**
5.1 Interfejs użytkownika . . . 6
5.2 Testy . . . 6
5.3 Pomiary wydajności . . . 6

1 Wstęp

Jako temat projektu wybrana została analiza algorytmu AEGIS-128[1]. Zaprezentowany on został wraz z pozostałymi dwoma wariantami (AEGIS-128L oraz AEGIS-256) podczas konkursu CAESAR w kategorii aplikacji o wysokiej wydajności. Jego autorami są Hongjun Wu i Bart Preneel.

Zadaniem algorytmu AEGIS jest uwierzytelnione szyfrowanie (ang. *authenticated encryption*), czyli ochrona zarówno treści wiadomości, jak i jej autentyczności. Autorzy wspominają o trzech podejściach do tego problemu: szyfr blokowy w specjalnym trybie, szyfr strumieniowy ze strumieniem klucza składającym się z dwóch części odpowiedzialnych za inne zadania oraz dedykowany algorytm do uwierzytelnionego szyfrowania. Autorzy postanowili wykorzystać ostatnie podejście. Można w nim wykorzystać szyfrowaną wiadomość do aktualizacji stanu szyfru, a zatem uwierzytelnienie wiadomości dostanie się prawie bezkosztowo.

Jak zostało wspomniane wyżej AEGIS stara się połączyć zadania szyfrowania i uwierzytelniania w celu zaoszczędzenia zasobów poprzez implementację dedykowanego algorytmu. AEGIS wykorzystuje rundową funkcję szyfrującą z AESa (bez ostatniej rundy). Wariant AEGIS-128 przetwarza wiadomości w blokach 16-bajtowych z wykorzystaniem pięciu rund szyfrujących AESa. Twórcy twierdzą, że koszt obliczeniowy ich szyfru jest o połowę mniejszy od AESa i dla wariantu AEGIS-128 wynosi on 0.66*cpb* (na procesorze Intel Core i5 Sandy Bridge). Twórcy twierdzą też, że ich algorytm oferuje bardzo wysokie bezpieczeństwo. O ile nonce nie zostanie użyty więcej niż raz, nie jest możliwe odzyskanie stanu szyfru i jego klucza w sposób inny niż przeszukiwanie wyczerpujące.

AEGIS jest przystosowany do komunikacji sieciowej ze względu na łatwość jego użycia do zabezpieczenia pakietu mimo braku szyfrowania jego nagłówków.

Niniejszy dokument zawiera specyfikację tego algorytmu, uzasadnienie jego bezpieczeństwa, przeprowadzone na nim ataki oraz opis implementacji.

2 Specyfikacja

AEGIS wykorzystuje rundową fukncję szyfrującą z AESa (bez ostatniej rundy), która ma postać *AESRound(A, B)*, gdzie A to 16-bajtowy stan, a B to 16-bajtowy klucz rundy. Funkcja zwraca na wyjściu 16 bajtów. Można ją zaimplementować wydajnie na architekturze x86 z wykorzystaniem instrukcji AESowej `__m128_aesenc_si128(A, B)`, gdzie A i B są 128-bitowymi liczbami całkowitymi typu `__m128i`.

Wariant AEGIS-128 wykorzystuje 128-bitowy klucz i wektor inicjalizujący. Szyfrowana wiadomość musi mieć mniej niż 2^{64} bitów. Tag uwierzytelniający składa się z nie więcej niż 128 bitów, ale 128 bitów jest bardzo zalecane.

2.1 Inicjalizacja

Inicjalizacja polega na wczytaniu klucza i IV do stanu w następujący sposób:

$$
\begin{aligned}
S_{-10,0} &= K_{128} \oplus IV_{128} \\
S_{-10,1} &= const_1 \\
S_{-10,2} &= const_0 \\
S_{-10,3} &= K_{128} \oplus const_0 \\
S_{-10,4} &= K_{128} \oplus const_1
\end{aligned}
$$

gdzie:

- S_i to 80-bajtowy stan w kroku i
- $S_{i,j}$ to 16-bajtowy j-ty blok stanu S_i
- K_{128} to 128-bitowy klucz
- IV_{128} to 128-bitowy wektor inicjalizujący
- $const_0$ i $const_1$ to odpowiednio pierwsze i ostatnie 16 bajtów stałej *const*

Stała *const* składa się z 32 bajtów i ma wartość początku ciągu Fibonacciego mod 256:

`0x000101020305080d1522375990e97962db3d18556dc22ff12011314273b528dd`

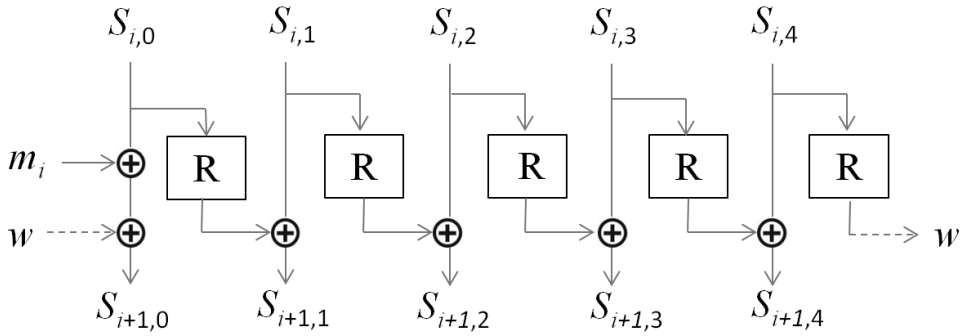
Następnie należy uruchomić szyfr na 10 kroków z zachowaniem następujących warunków:

- dla i od -5 do -1: $m_{2i} = K_{128}$, $m_{2i+1} = K_{128} \oplus IV_{128}$, gdzie m_i to i-ty 16-bajtowy blok wiadomości
- dla i od -10 do -1: $S_{i+1} = StateUpdate128(S_i, m_i)$

2.2 Aktualizacja stanu

Funkcja aktualizująca stan S ma postać $S_{i+1} = StateUpdate128(S_i, m_i)$ i działa następująco:

$S_{i+1,0} = AESRound(S_{i,4}, S_{i,0} \oplus m_i)$
$S_{i+1,1} = AESRound(S_{i,0}, S_{i,1})$
$S_{i+1,2} = AESRound(S_{i,1}, S_{i,2})$
$S_{i+1,3} = AESRound(S_{i,2}, S_{i,3})$
$S_{i+1,4} = AESRound(S_{i,3}, S_{i,4})$



Rysunek 1: Funkcja aktualizacji stanu dla AEGIS-128. R oznacza AESową rundową funkcję szyfrującą bez XORowania z kluczem rundy. w to tymczasowa wartość 16-bajtowa.

2.3 Przetwarzanie uwierzytelnionych danych

Po inicjalizacji powiązane dane AD, czyli dane, które nie podlegają szyfrowaniu / odszyfrowaniu są używane do aktualizacji stanu. Pełnią one funkcję uwierzytelnienia. Muszą składać się z mniej niż 2^{64} bitów. Jeśli ich długość to 0 bitów, wtedy poniżej opisanych kroków się nie wykonuje.

1. Uzupełnij AD zerami (w znaczeniu bitów) na końcu, aby dane składały się z pełnych bloków 128-bitowych, jeśli jest taka potrzeba.
2. Dla i od 0 do $\lceil \frac{adlen}{128} \rceil - 1$ wykonuj $S_{i+1} = StateUpdate128(S_i, AD_i)$

2.4 Szyfrowanie wiadomości

Po przetworzeniu powiązanych danych AD, w każdym kroku szyfrowania 16-bajtowy blok tekstu P_i po szyfrowaniu staje się C_i. Jeśli długość wiadomości to 0 bitów, krok jest pomijany.

1. Jak w kroku 1. dla AD, ale wykonywane na wiadomości. Uzupełniony blok jest używany do aktualizacji stanu, ale szyfrowany jest tylko blok pierwotny (nieuzupełniony).
2. $u = \lceil \frac{adlen}{128} \rceil$ oraz $v = \lceil \frac{msglen}{128} \rceil$. Dla i od 0 do $v - 1$ wykonywane jest szyfrowanie i aktualizacja stanu:
   $C_i = P_i \oplus S_{u+i,1} \oplus S_{u+i,4} \oplus (S_{u+i,2} \& S_{u+i,3})$
   $S_{u+i+1} = StateUpdate128(S_{u+i}, P_i)$

2.5 Finalizowanie

Po zaszyfrowaniu wiadomości należy wygenerować tag uwierzytelniający.

1. $tmp = S_{u+v,3} \oplus (adlen||msglen)$, gdzie $||$ jest konkatenacją, a $adlen$ oraz $msglen$ są reprezentowane jako liczby 64-bitowe.
2. Dla i od $u+v$ do $u+v+6$ wykonuj $S_{i+1} = StateUpdate128(S_i, tmp)$
3. $T' = \oplus_{i=0}^{4} S_{u+v+7,i}$
4. Tag uwierzytelniający T to pierwsze t bitów T', gdzie $64 \leq t \leq 128$

2.6 Odszyfrowanie i weryfikacja

W celu odszyfrowania i weryfikacji dokładne wartości długości klucza, IV oraz tagu są wymagane. Odszyfrowanie zaczyna się od inicjalizacji i przetworzenia uwierzytelnionych danych. Następnie odszyfrowanie wiadomości przebiega w następujący sposób:

1. Jeśli ostatni blok szyfrogramu nie jest pełny (128 bitów), odszyfruj tylko część bloku. Do aktualizacji stanu użyj pełnego bloku tekstu uzupełnionego zerami (w znaczeniu bitów).
2. Dla i od 0 do $v-1$ wykonywane jest odszyfrowanie i aktualizacja stanu:
   $P_i = C_i \oplus S_{u+i,1} \oplus S_{u+i,4} \oplus (S_{u+i,2} \& S_{u+i,3})$
   $S_{u+i+1} = StateUpdate128(S_{u+i}, P_i)$

Na koniec odbywa się finalizacja, która jest taka sama jak dla szyfrowania.

Należy podkreślić, że jeśli weryfikacja zakończy się niepowodzeniem, szyfrogram oraz nowo wygenerowany tag uwierzytelniający nie mogą zostać zwrócone (ujawnione). W przeciwnym wypadku stan algorytmu jest podatny na ataki *known-plaintext* oraz *chosen-ciphertext* (z wykorzystaniem stałego IV).

3 Ataki

3.1 Ataki na inicjację

3.1.1 Atak różnicowy

Różnica w IV może przeniknąć do szyfrogramu, zatem istnieje możliwość wykonania ataku różnicowego. Jeżeli wprowadzono by różnicę w IV, to musiałaby ona przejść przez ponad 10, 16 i 20 funkcji rundy AESa w przypadkach odpowiednio AEGIS-128, AEGIS-256 i AEGIS-128L. W celu zapobiegnięcia wyeliminowaniu różnicy w stanach podczas inicjalizacji, wspomniana różnica w IV jest wstrzykiwana do stanów kilkukrotnie (5 razy dla AEGIS-128, 8 razy dla AEGIS-256 i 10 razy dla AEGIS-128L). Dzięki temu atak różnicowy powinien być bardziej kosztowny od przeszukiwania klucza.

3.2 Ataki na szyfrowanie

3.2.1 Atak statystyczny

Przy założeniu, że IV jest używane tylko raz, atak na inicjalizację jest niemożliwy do wykonania. Nieliniowa natura AEGIS-a powoduje, że trudno jest zastosować korelację do odszyfrowania wiadomości.

3.2.2 Losowość klucza

Według analizy Brince'a Minauda wymagane jest 2^{140} i 2^{188} danych, aby odróżnić strumienie klucza dla wersji AEGIS-a odpowiednio 128 i 256.

3.3 Ataki na uwierzytelnienie

3.3.1 Odzyskanie klucza lub stanu

Atakujący może dodać różnicę do stanu w procesie weryfikacji tagu i otrzymać odszyfrowaną wiadomość (w przypadku, gdy możliwy jest wielokrotny atak podrabiający przy takiej samej parze IV i klucza). W ataku podrabiającym prawdopodobieństwo znajomości oryginalnej wiadomości wynosi 2^{-t} (t to długość tagu uwierzytelniającego w bitach). Twórcy algorytmu rekomendują używanie tagu o maksymalnej długości w celu podniesienia liczby wymaganych prób podrobienia do 2^{128} do uzyskania stanu.

3.3.2 Wewnętrzne kolizje

Znanym atakiem na algorytmy MAC (*Message authentication code*) jest atak dnia urodzin (*Birthday attack*). Atakujący mógłby wprowadzić różnicę w procesie odszyfrowania i weryfikacji tagu przez modyfikację szyfrogramu, jednak duży rozmiar stanu zwiększa bezpieczeństwo tego algorytmu.

3.4 Inne ataki

Istnieje możliwość, że wszystkie bloki stanu będą sobie równe, co prowadzi do zrównania wszystkich bloków w następnym stanie. Istnieje jedynie 2^{128} takich stanów, zatem prawdopodobieństwo wystąpienia takiego stanu wynosi 2^{-512} w przypadku AEGIS-128, 2^{-640} w AEGIS-256 i 2^{-896} w AEGIS-128L.

Istnieje również przypadek, gdy cztery kolumny w każdym 16-bajtowym elemencie posiadają taką właściwość. Taka sytuacja zachodzi w 2^{160} stanach w AEGIS-128 (prawdopodobieństwo wystąpienia wynosi 2^{-480}), 2^{192} stanach w AEGIS-256 (prawdopodobieństwo wystąpienia wynosi 2^{-608}) i 2^{256} stanach w AEGIS-128L (prawdopodobieństwo wystąpienia wynosi 2^{-768}).

4 Uzasadnienie bezpieczeństwa

W celu osiągnięcia bezpiecznego szyfrowania, IV losowany jest na etapie inicjalizacji, a stan ten nie może być odzyskany z szyfrogramu. Inicjalizacja AEGIS składa się z 10 kroków, więc możemy przyjąć, że inicjalizacja AEGIS jest silna. W celu zapewnienia, że stan nie będzie mógł być odzyskany z szyfrogramu szybciej niż w wypadku ataku bruteforce, zostało zapewnione że co najmniej 20 rund funkcji AES jest zaangażowanych w atak odzyskania stanu AEGIS-128.

W celu osiągnięcia wysokiego poziomu bezpieczeństwa uwierzytelnienia, zapewnione zostało że jakakolwiek różnica wprowadzona do stanu doprowadzi do znaczącej różnicy z odpowiednio małym prawdopodobieństwem, tak aby trudno było wykonać atak typu *forgery attack*. Rozwiązanie to było zainspirowane podobnym rozwiązaniem użytym w Pelican MACu. W Pelican MACu różnica przechodzi przez 4 rundy funkcji AES przed spotkaniem się z inną różnicą, w efekcie przynajmniej 25 SBoxów jest zaangażowanych. Dowód bezpieczeństwa przeciwko atakowi *differential forgery attack* jest prosty dla Pelican MAC. W AEGIS pierwsza różnica w stanie przechodzi przez co najmniej 4 rundy funkcji AES zanim zacznie na nią wpływać inna różnica. Dodatkowo, kiedy różnica przechodzi przez funkcje rundy w AES, to różnice są wstrzykiwane do co najmniej czterech bloków w stanie, co sprawia, że usunięcie różnicy stanu staje się trudne.

5 Implementacja

Szyfr AEGIS-128 został zaimplementowany w języku Python 3.10. Implementacja znajduje się na GitHubie pod adresem https://github.com/BlueAlien99/krys-aegis-128. Zostały wykorzystane biblioteki *pwntools* (funkcja *xor*) oraz *numpy* (funkcja *transpose*). Cały AEGIS został zaimplementowany w pliku `src/aegis_128.py` w formie modułu eksportującego dwie funkcje:

- `encrypt(k: str, iv: str, ad: str, p: str)` — zwraca krotkę zawierającą szyfrogram oraz tag
- `decrypt(k: str, iv: str, ad: str, c: str, tag: str)` — zwraca odszyfrowany tekst

Wszystkie parametry muszą być hex-stringami (t.j. zawierać tylko znaki z przedziału [a-fA-F0-9]).

Mapowania etapów ze specyfikacji na funkcje:

- inicjalizacja — `_initialize(k: bytes, iv: bytes)`
- aktualizacja stanu — `_state_update_128(s: bytes, m: bytes)`
- przetwarzanie uwierzytelnionych danych — `_process_ad(s: bytes, ad: bytes)`
- szyfrowanie wiadomości — `_encrypt_msg(s: bytes, p: bytes)`
- deszyfrowanie wiadomości — `_decrypt_msg(s: bytes, c: bytes)`
- finalizowanie — `_finalize(s: bytes, ad: bytes, p: bytes)`

Funkcja `_state_update_128` wykorzystuje funkcję rundy z AESa `aes_round(state, key)`, która znajduje się w pliku `src/aes/aes.py` (przedstawiona poniżej). W celu zapewnienia przenośności postanowiliśmy nie korzystać z instrukcji AESowej dla architektury x86. Ponadto mieliśmy już częściowo przygotowany kod napisany na potrzeby laboratoriów, z którego zostały pozostawione tylko niezbędne elementy.

```
def aes_round(state, key):
    x = utils.transpose(utils.bytes2matrix(state))

    x = utils.sub_bytes(x)
    x = utils.shift_rows(x)
    x = utils.mix_columns(x)
    x = utils.add_round_key(utils.bytes2matrix(key), x)

    return utils.matrix2bytes(utils.transpose(x))
```

5.1 Interfejs użytkownika

Po uruchomieniu pliku `src/__main__.py` poleceniem `python ./src` można pobawić się implementacją szyfru AEGIS lub uruchomić testy (opisane w następnym podrozdziale). Można np. zaszyfrować wiadomość bezpośrednio z wiersza poleceń (przykład poniżej). W celu uzyskania pomocy należy wywołać skrypt z flagą `-h`.

```
$ python ./src -m e --key my_secret_key_12 --iv my_secret_vector --ad xd1337 \
--text "this is super secret message"
{"ct": "4e6f8ebe5e211283aab686a2e6365f21353058c795556c680c6679e4",
"tag": "b495d121a81e44b77f44c2954fa2ff1a"}

$ python ./src -m d --key my_secret_key_12 --iv my_secret_vector --ad xd1337 \
--text 4e6f8ebe5e211283aab686a2e6365f21353058c795556c680c6679e4 \
--tag b495d121a81e44b77f44c2954fa2ff1a
this is super secret message
```

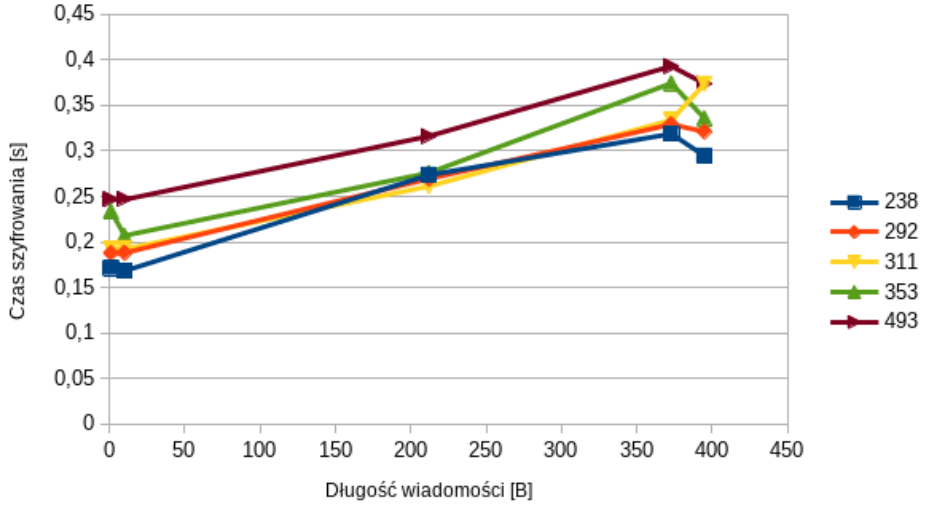
5.2 Testy

Przypadki testowe zostały zawarte w pliku `test-vectors.json`. Zawiera on wektory z dokumentacji AEGISa oraz wektory z repozytorium z przykładową implementacją. Wektory z dokumentacji były niewystarczające, ponieważ nie zawierały przypadków brzegowych (np. dla samego k i iv lub dla danych wymagających paddingu). W sumie zebraliśmy 13 wektorów testowych. Testy można uruchomić poleceniem `python ./src -m t`. Przypadki brzegowe (w szczególności te proste) okazały się niezwykle przydatne podczas debugowania i szukania błędów w implementacji.

5.3 Pomiary wydajności

Wykonane zostały pomiary wydajności zaimplementowanego algorytmu. Napisano skrypt *data_generator.py* dzięki któremu wygenerowane zostały wektory do testów, które zostały zapisane w *enc_vectors.py*. Uzyskano po 5 wektorów inicjalizujących, kluczy o długościach 16 bajtów oraz 5 wiadomości do zaszyfrowania i dodatkowych danych o losowych długościach od 1 do 500 bajtów.

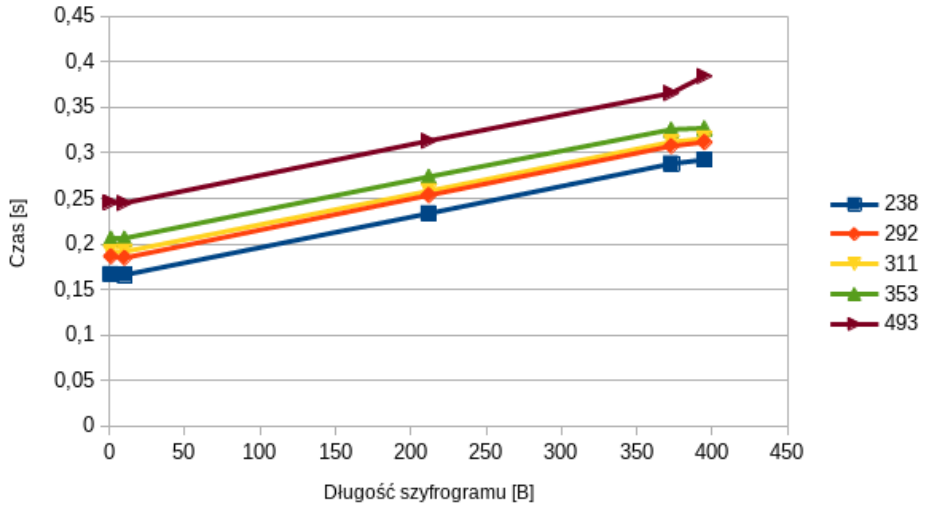
Testy szyfrowania zaimplementowane w skrypcie *enc_ benchmark.py* przeprowadzono dla wszystkich kombinacji wektorów inicjalizujących, kluczy, wiadomości i danych dodatkowych. Operacja zaszyfrowania 625 kombinacji zajęła łącznie 209,3 sekund, czyli średnia szyfrowania wynosiła 0,335 sekundy. Po drobnej modyfikacji skryptu otrzymano wyniki pomiaru czasu szyfrowania w zależności od długości wiadomości i dodatkowych danych. Poniżej przedstwaiono wykres zarejestrowanych rezultatów.



Rysunek 2: Zależność czasu szyfrowania od długości wiadomości i dodatkowych danych.

Zauważyć można fakt zwiększania czasu szyfrowania wraz ze wzrostem długości wiadomości oraz dodatkowych danych. Ciekawym zjawiskiem jest spadek czasu szyfrowania przy najdłuższej oryginalnej wiadomości. Zaszło to w 4 na 5 prób.

Aby umożliwić testowanie odszyfrowywania należało zaszyfrować dane z *enc_ vectors.py* przy pomocy skryptu *dec_ vectors_ generator.py* oraz zapisać uzyskane szyfrogramy i tagi w pliku *dec_ vectors.py*. Testy deszyfrowania opisano w skrypcie *dec_ benchmark.py*. Deszyfrowanie poprzednio zaszyfrowanych danych zajęło 208,3 sekund, czyli pojedyncze deszyfrowanie trwało średnio 0.33 sekundy. Ponownie zmodyfikowano skrypt pomiarowy, aby móc zmierzyć zależność czasu deszyfrowania od długości szyfrogramu oraz danych dodatkowych. Rezultaty przedstawiono na poniższym wykresie.



Rysunek 3: Zależność czasu deszyfrowania od długości szyfrogramu i dodatkowych danych.

Po raz kolejny można zaobserwować wydłużenie czasu przetwarzania danych wraz z ich wzrostem.

Literatura

[1] Hongjun Wu oraz Bart Preneel. A fast authenticated encryption algorithm (full version). https://eprint.iacr.org/2013/695.pdf. Dostęp zdalny: 2022-12-20.