

**WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI,  
INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ**

KATEDRA AUTOMATYKI I ROBOTYKI

Praca dyplomowa magisterska

*Porównanie i implementacja zaawansowanych szyfrów blokowych*

*Comparison and implementation of advanced block ciphers*

Autor: *Jarosław Jałocha*

Kierunek studiów: Automatyka i Robotyka

Opiekun pracy: Prof. *dr. Marek Ogiela*

Kraków, 2019

Spis treści

[1. Wstęp 2](#_Toc12214905)

[2. Charakterystyka szyfrów blokowych 4](#_Toc12214906)

[3. DES 12](#_Toc12214907)

[4. AES 19](#_Toc12214908)

[5. RC6 25](#_Toc12214909)

[6. Opis zaimplementowanego programu 32](#_Toc12214910)

[7. Porównanie algorytmów 35](#_Toc12214911)

[8. Wnioski i spostrzeżenia 43](#_Toc12214912)

[Literatura 45](#_Toc12214913)

# Wstęp

Utajnianie informacji nie jest czymś, co zostało wynalezione w ostatnich latach, a nawet biorąc pod uwagę okres stuletni nie jest niczym nowym. Od zarania dziejów dbano, żeby wiadomości nie trafiały w niepowołane ręce, gdyż mogło by się to skończyć katastrofą. W tym celu zaczęto stosować szyfry zmieniające początkowy tekst wiadomości na postać całkowicie niezrozumiałą dla niepowołanej osoby. Oczywiście, przekształcenie było znane dla docelowego adresata, zatem mógł odwrócić transformację i odczytać ukryty sens przekazu. Działania te dały początek dziedzinie nauki zwanej **kryptologią** [15]. Wyróżnia się w niej dwie odrębne gałęzie:

* kryptografię (gałąź badająca sposoby utajniania wiadomości),
* kryptoanalizę (gałąź przełamywania szyfru nie posiadając o nim pełnej wiedzy, np. nie znając klucza tudzież szczegółów implementacji).

Jak już wspomniano, szyfrowanie używane już było w czasach starożytnych. Ze względu na oczywiste ograniczenia technologiczne stosowano proste szyfry: przestawieniowe (sztandarowym przypadkiem jest tutaj szyfr Cezara), podstawieniowe lub permutacyjne. Składały się one zazwyczaj z jednej operacji i były dość proste do złamania, nawet w owych, zamierzchłych czasach.

Rozwój techniki zwiększył możliwości algorytmów kryptograficznych. Zaczęto tworzyć maszyny szyfrujące o skomplikowanych strukturach, w wyniku działania których powstawał szyfrogram w ogóle nieprzypominający początkowej wiadomości, a zlepek losowych znaków (co jest jednym z aksjomatów skutecznego szyfrowania). Kluczową własnością takich urządzeń był ukryty algorytm szyfrujący, zatem głównym celem kryptoanalityków było poznanie sposobu tworzenia szyfru i odwrócenie go, żeby uzyskać pierwotną wiadomość. Przykładem takiej maszyny jest niemiecka Enigma stosowana szeroko w czasie II Wojny Światowej. Jak pokazała historia tworzenie skomplikowanych algorytmów jest z punktu widzenia bezpieczeństwa nieefektywne, gdyż przechwycenie maszyny prędzej czy później zakończy się złamaniem algorytmu.

Wraz z pojawieniem się komputerów oraz niedoskonałością rozwiązań mechanicznych zdecydowano się zmienić podejście. Stwierdzono, że szyfrowanie powinno odbywać się w zgodzie z tzw. **Zasadą** **Kerckhoffsa** [17], która mówi, że dobry system kryptograficzny powinien pozostać bezpieczny nawet wtedy, kiedy szczegóły jego działania (w szczególności przeprowadzone operacje szyfrujące) są znane. Jedyną nieznaną, z punktu widzenia atakującego, informacją powinien być tzw. **klucz**, czyli sekretna wartość niezależna od treści wiadomości, która zostaje użyta w procesie szyfrowania oraz deszyfrowania. Ze względu na sposób użycia klucza wyróżnia się szyfry [1]:

* symetryczne – klucz używany do szyfrowania i deszyfrowania jest jednakowy (tajny),
* asymetryczne – klucz używany do szyfrowania jest znany (tzw. klucz publiczny), jest on różny od klucza używanego do odszyfrowania wiadomości (tzw. klucz prywatny, który pozostaje tajny).

Szyfry dzielą się również ze względu na sposób kodowania informacji. Do podziału tego włączają się m.in.:

* szyfry blokowe – w jednej iteracji utajniona zostaje z góry określona grupa bitów zwana blokiem,
* szyfry strumieniowe – utajniony zostaje każdy bit z osobna.

W pracy oraz programie komputerowym z nią związanym znajdzie się studium nad szyframi należącymi do grupy **symetrycznych szyfrów blokowych**. Przyszłe rozdziały dotyczyć będą krótkiego, ogólnego wprowadzenia do szyfrów blokowych, następnie przybliżone zostaną szczegóły implementacyjne każdego z zaimplementowanych szyfrów. Kolejny rozdział będzie dotyczył opisu stworzonego programu komputerowego, który pozwoli użyć każdego z opisanych algorytmów do zaszyfrowania określonego pliku oraz umożliwi przeprowadzenie badań na temat szybkości algorytmów. Ich opis zostanie zawarty w rozdziale 7., dzięki którym wysnute zostaną wnioski opisane w kolejnym rozdziale.

# Charakterystyka szyfrów blokowych

We wstępie przedstawiono intuicyjną, aczkolwiek nieformalną definicję szyfru oraz związanym z nim pojęć. W niniejszym rozdziale zostanie zawarta formalizacja pewnych podstawowych pojęć związanych z kryptografią, a także przedstawiony zostanie zarys działania szyfrów blokowym, będący wstępem do opisu poszczególnych rodzajów implementacji danego systemu.

Każdy system kryptograficzny związany jest z następującą piątką elementów [3]:

* przestrzenią wiadomości jawnych M,
* przestrzenią szyfrogramów C,
* przestrzenią kluczy K,
* rodziną funkcji szyfrujących takich, że: ,
* rodziną funkcji deszyfrujących takich, że: .

Szyfrem potocznie nazywa się konkretnego przedstawiciela rodziny . Zachodzi również bardzo istotna własność między odpowiadającymi sobie funkcjami z rodziny oraz . Stanowi ona następująco (2.1):



Warunek (2.1) mówi, iż jeżeli do zaszyfrowania wiadomości użyto funkcji , to zastosowanie funkcji do odszyfrowania kryptogramu spowoduje uzyskanie pierwotnego tekstu jawnego.

Oczywistym jest również fakt, iż funkcje z rodzin i muszą być różnowartościowe, w przeciwnym razie niemożliwym byłoby jednoznaczne powiązanie szyfrogramu z wiadomością. Bardziej złożone, ogólne rozważania matematyczne nad strukturą funkcji szyfrujących oraz deszyfrujących można znaleźć w [17].

Zgodnie z zasadą Kerckhoffsa, największą trudnością stojącą na drodze przechwycenia oryginalnej wiadomości jest klucz szyfrowania. Pierwszym, oczywistym zagrożeniem z tego wynikającym jest fakt, iż zarówno nadawca, jak i odbiorca muszą posiadać tę samą wartość klucza, zatem konieczne jest przekazanie jego wartości. Jednym z rozwiązań jest zaszyfrowanie klucza innym algorytmem i przesłanie go w postaci szyfrogramu. Jest to rozwiązanie prowadzące do nieskończonej rekurencji, albowiem kolejny algorytm również korzysta z kluczy, który należy przekazać, najlepiej poprzez następny szyfrowany kanał. Efektem tego jest konieczność jawnego przekazania klucza, która odbywa się poprzez tzw. kanał bezpieczny [1].

Kolejnym aspektem dotyczącym klucza jest jego postać. Funkcje szyfrujące dążą do tego, ażeby szyfrogramy miały postać całkowicie losową, a korelacja poszczególnych znaków była jak najmniejsza. W związku z tym, wartość klucza powinna być inicjalizowana losowo. Bardzo poważnym błędem w formowaniu klucza jest użycie wbudowanego generatora liczb losowych bez upewnienia się, że jest on przystosowany do zastosowań kryptograficznych. Wyróżniamy generatory RNG (ang. *Random Number Generator,* generator liczb losowych), który korzystając z danych wejściowych (pomiar temperatury, częstotliwość kliknięć myszką itp.) generuje losowe liczby. Jest on przystosowany kryptograficznie, ponieważ trudno przewidzieć liczbę przezeń wygenerowaną, aczkolwiek jest to proces stosunkowo wolny oraz podatny na manipulację (poprzez wymuszanie pewnych zachowań czujników). Drugim rodzajem generatorów są generatory PRNG (ang. *Pseudo Random Number Generator*, generator liczb pseudolosowych). Szczególnie ten typ generatorów może być podatny na atak poprzez nieprzystosowanie do zadań kryptograficznych, albowiem korzysta on z tzw. ziarna. Znając wartość ziarna (w generatorach niekryptograficznych) możliwym jest przewidzenie ciągu wygenerowanych liczb, co ewidentnie pozbawia algorytm szyfrujący jakiegokolwiek bezpieczeństwa. Należy więc się upewnić, że używa się właściwego generatora [1].

Równie ważną właściwością klucza jest jego długość. Odpowiednia liczba bajtów zapewnia zarówno dużą różnorodność wartości, co pozwala na częste zmiany, ale również uodparnia na tzw. ataki brute-force. Jest to sposób, w którym wykonuje się przeszukiwanie zupełne przestrzeni kluczy aż do momentu, w którym z szyfrogramu uzyskuje się zrozumiałą wiadomość. Wydawać by się mogło, że taki atak jest trudny w wykonaniu, aczkolwiek jak udowodniono w dziele [19], złamanie 6-znakowego hasła metodą przeszukiwania zupełnego kończy się sukcesem po około sekundzie (przy użyciu zaawansowanego układu GPU zrównoleglającego czynności). Klucze szyfrów używanych w przeszłości nierzadko miało porównywalną długość w skali bajtów (np. klucz szyfru DES ma 7 bajtów), co uwydatnia fakt, iż wraz ze wzrastającą mocą obliczeniową, powinna rosnąć również wielkość klucza.

Liczba operacji wymaganych do wyszukania klucza i zweryfikowania jego poprawności rośnie niemal eksponencjalnie wraz ze wzrostem jego wielkości, zatem w obecnych czasach stosuje się klucze o wielkości 16 bajtów [1], jednak liczba ta jest zależna od implementacji. Należy również wspomnieć, że odpowiednia długość klucza nie zapewnia całkowitego bezpieczeństwa (zwłaszcza przy błędnej implementacji algorytmu lub złych założeniach). Co więcej, bezpieczeństwo obecnie stosowanych szyfrów nie jest bezpieczeństwem bezwarunkowym [1]. Przyjmuję się zatem dowód heurystyczny, tj. fakt, iż nikomu nie udało się złamać danego systemu, pomimo wielu przeprowadzonych prób, czyni szyfr akceptowalnym do użytku.

Systemami kryptograficznymi, nad którymi będą prowadzone rozważania w niniejszej pracy są szyfry blokowe. Jak już wspomniano wcześniej, jest to rodzaj symetrycznych szyfrów, w którym przetwarzane są bloki danych. Sama wielkość bloku zależy od implementacji, może to być 8 bajtów jak w algorytmie DES czy też 16 jak w przypadku algorytmu AES [14]. Nie powinny być one jednak za duże (ze względu na oszczędność pamięci), ale również zbyt małe, gdyż spowodowało by to podatność na atak tzw. książki kodowej, czyli stworzenia tabel, które przypisywałyby konkretny blok danych danemu szyfrogramowi – przy większych blokach taka operacja wymaga zbyt dużo nakładu obliczeniowo/pamięciowego, zatem zwyczajowo uznaje się 64 bity za minimum [1]. Nie istnieje „złoty środek” w kwestii kreacji szyfrów blokowych, jednak główną ideą jest stworzenie algorytmu, w którym powtarza się określone bloki instrukcji. Jeden blok instrukcji zwany jest **rundą**, która sama w sobie nie zapewnia bezpieczeństwa szyfru, aczkolwiek powtórzona kilka lub kilkanaście razy tworzy szyfrogram przypominający losowy zlepek znaków. W kwestii konstrukcji kroków rundy, wyróżnia się dwie główne techniki: sieci podstawieniowe-permutacyjne (AES) lub sieci Feisela (DES). Szczegóły na temat każdej z technik zostaną zawarte w przyszłych rozdziałach.

Wymóg co do określonego rozmiaru bloku niesie za sobą konsekwencje. Patrząc ze strony praktycznej, rozmiar wiadomości do zaszyfrowania bardzo rzadko jest równy wielokrotności rozmiaru bloku. W rzeczywistości, ostatni blok zawiera mniejszą liczbę danych. W takim wypadu blok należy uzupełnić odpowiednimi danymi. Wyróżnia się dwie wiodące techniki [15]:

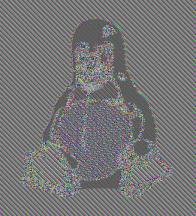
* „kradzież szyfrogramu”, czyli dopełnienie bloku danymi z poprzedniego szyfrogramu,
* padding, technika, w której wypełnia się wiadomość pewnym charakterystycznym ciągiem.

Bardzo popularną techniką paddingu jest technika PKCS#7, w której ciągiem danych jest powtórzona liczba brakujących bajtów (przykładowo, jeśli brakuje 10 bajtów to ciąg uzupełniający stanowi dziesięć liczb 10). Metoda ta ma jedną wadę, mianowicie nawet w przypadku idealnego pokrycia bloków przez dane wejściowe, dodaje się blok dodatkowy (szesnaście razy powtórzona liczba 16), w celu późniejszego usunięcia dopełnienia. W związku z powyższym, rozmiar danych wejściowych może zostać zwiększony o jeden bajt.

Rozmiar wiadomości do zaszyfrowania prawie zawsze przekracza rozmiar jednego bloku, w związku z czym w algorytmie należy zdefiniować schemat przetwarzania kolejnych bloków. Owy schemat nosi nazwę **trybu pracy szyfru blokowego**. Tryb pracy określa zależności (bądź ich brak) pomiędzy kolejnymi blokami tekstu. Istnieje kilka podstawowych trybów pracy, jednak w pracy zostanie zawarty opis dwóch z nich, jako że zachowują one własności szyfru blokowego, podczas gdy pozostałe zmieniają szyfr blokowy w pełnoprawny szyfr strumieniowy. Opis pozostałych szyfrów można znaleźć w [15], zaś wcześniej wspomniane dwa tryby to:

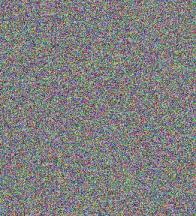
* **ECB** (ang. *electronic code book,* tryb elektronicznej książki kodowej) – w trybie tym każdy blok wiadomości, zarówno podczas szyfrowania jak i deszyfrowania jest przetwarzany zupełnie niezależnie. Ma to swoją zaletę w możliwości całkowitego zrównoleglenia szyfrowania, jednak niesie ze sobą dużo większą wadę w postaci niskiego bezpieczeństwa. Niezależność bloków powoduje, iż ten sam blok znajdujący się w różnych miejscach wiadomości zostanie przetworzony na taki sam szyfrogram. Doskonale jest to widoczne w przypadku szyfrowania obrazów, pomimo zaszyfrowania pikseli, wciąż można zobaczyć zarys tego, co było przedstawione na obrazie. Sławny cytat opisujący tę wadę w przetłumaczeniu na polski brzmi „widać Pingwina” [1] i odnosi się do wyniku szyfrowania logotypu systemu operacyjnego Linux. Przykład ten zaprezentowano na rysunkach 2.1 oraz 2.2.

Rysunek 2.1. Obraz poddany szyfrowaniu.



Rysunek 2.2. Szyfrogram obrazu.

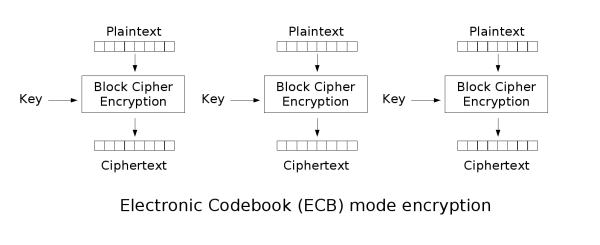
* **CBC** (ang. *cipher block chaining*, wiązanie bloków zaszyfrowanych) – w trybie tym bloki szyfrowane są sekwencyjnie, natomiast przed szyfrowaniem następnej wiadomości, następuje dodanie do bloku (poprzez operację alternatywy wykluczającej) szyfrogramu uzyskanego z poprzedniej wiadomości. Uniemożliwia to zrównoleglenie szyfrowania, albowiem następny blok jest zależny od poprzedniego, aczkolwiek przeprowadzenie deszyfrowania może odbywać się w sposób równoległy (pod warunkiem przechowania wszystkich szyfrogramów, co jest częstym przypadkiem). Co oczywiste, pierwszy blok nie posiada poprzedniego szyfrogramu, zatem przy procesie szyfrowania dodaje się do niego tzw. blok inicjalizacyjny IV (od angielskiego *initialization vector* – wektor inicjalizacyjny). Powinien być on generowany podobnie do klucza, a więc w sposób losowy [15]. Tryb CBC jest również odporny na wady trybu ECB. Dzięki zmianie bloku przed szyfrowaniem, te same bloki wiadomości jawnej uzyskają różny szyfrogram, co z kolei spowoduje niemożliwość szukania schematów, jako że wynik działania szyfru znacznie bardziej przypomina losowy ciąg (szum). Wynik szyfrowania obrazu w trybie CBC zaprezentowano na rysunku 2.3.



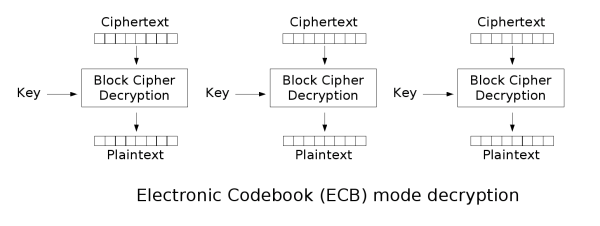
Rysunek 2.3. Wynik szyfrowania obrazu 2.1 w trybie CBC.

Mając na względzie bezpieczeństwo, nie powinno stosować się trybu ECB w praktyce [1]. Ułatwia on kryptoanalizę, co może doprowadzić do złamania wiadomości. Niekoniecznie może to prowadzić do złamania całego szyfru (w szczególności klucza), aczkolwiek narusza poufność danych. W związku z powyższym, rekomendowanym trybem pracy dla szyfrów blokowych (wyłączając tryby zmieniające tryby blokowe na strumieniowe) jest tryb CBC. Istnieje również jego modyfikacja, tryb PCBC, w której poza szyfrogramem dodaje się tekst jawny poprzedniego bloku, ale jest to zmiana w sposobie działania na tyle nieistotna, że szczegółowy opis jest bezzasadny (samo zaś dodanie tekstu jawnego może mieć zastosowanie w przypadku, gdy szyfrogram poszczególnych bloków jest podobny).

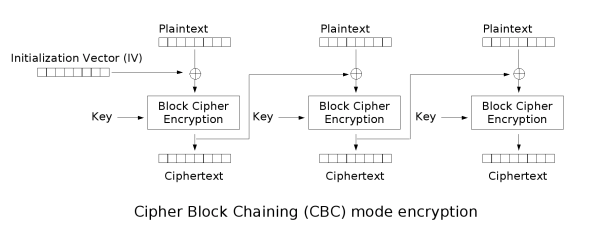
Obrazowo tryby ECB oraz CBC (w procesie szyfrowania i deszyfrowania) zostały przedstawione na rysunkach 2.4-2.7.



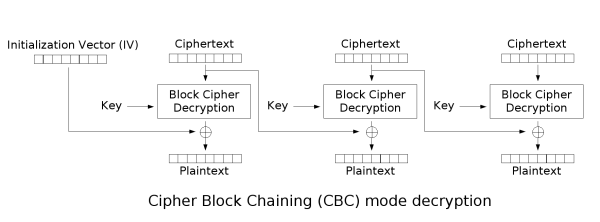
Rysunek 2.4. Schemat szyfrowania w trybie pracy ECB.



Rysunek 2.5. Schemat odszyfrowania w trybie pracy ECB.



Rysunek 2.6. Schemat szyfrowania w trybie pracy CBC.



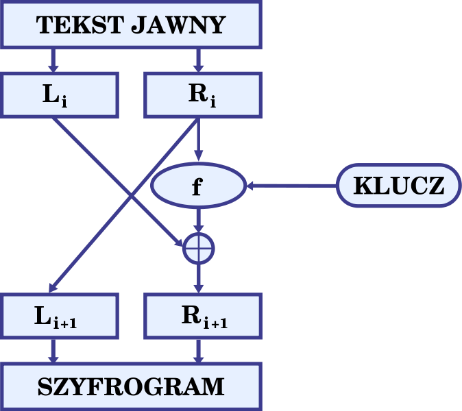
Rysunek 2.7. Schemat deszyfrowania w trybie pracy CBC.

W latach 70 XX wieku w Stanach Zjednoczonych pojawiła się potrzeba ustandaryzowania szyfrów. Algorytmów do wyboru było wiele, aczkolwiek nie były one satysfakcjonujące, zatem powołano agencję NBS (ang. *National Bureau of Standard*, narodowe biuro standardów przekształcone potem w NIST, Instytut Standardów i Technologii). Biuro przyjęło za standard szyfr opracowany przez firmę IBM i nadało mu nazwę DES (ang *Data Encryption Standard*, standard enkrypcji danych). Od początku był on krytykowany za wielkość zastosowanego klucza, aczkolwiek w tamtych latach moc obliczeniowa nie pozwalała na skuteczne ataki typu brute force. Wraz z upływem lat i postępem technologicznym, obawy o bezpieczeństwo algorytmu DES zaczęły być coraz bardziej uzasadnione. Stan ten sprawił, iż w 1997 NIST ogłosił konkurs na najlepszy algorytm, który miał zostać nowym standardem w kryptografii oraz uzyskać nazwę AES (ang. *Advanced Encryption Standard*, zaawansowany standard enkrypcji). Do konkursu, który zakończył się w 2001 roku, wpłynęło 15 propozycji zaś 5 zostało wybranych do finału. W niniejszej pracy znajdzie się opis dwóch z nich, algorytmu Rijandel, czyli ostatecznego zwycięzcy oraz algorytmu RC6. Opisany i poddany porównaniu zostanie również pierwszy standard, a więc algorytm DES.

# DES

**DES**, jak już wspomniano w rozdziale 2, został stworzony, aby zaspokoić potrzebę posiadania algorytmu będącego standardem szyfrowania. Algorytm powstał w 1975 roku i został opracowany przez firmę IBM, bazując na znanej wcześniej metodzie szyfrowania Lucifer [17]. Standardem pozostawał do roku 2001, kiedy stwierdzono, iż szyfr nie daje już odpowiedniego poziomu bezpieczeństwa, co jest głównie spowodowane małym rozmiarem klucza (64 bity, w czym tylko 56 efektywnie używane w szyfrowaniu). Zgodnie z informacjami z rozdziału 2., wzrastająca moc obliczeniowa komputerów sprawia, iż nieodpowiedni rozmiar klucza naraża algorytm na ataki typu brute force.

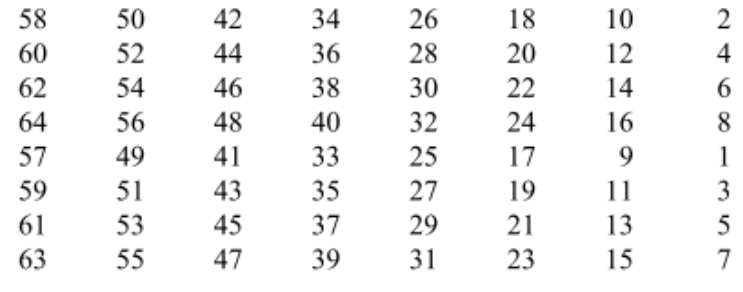
Agencja zlecająca zaprojektowanie algorytmu, a więc NIST, skonsultowało postać ostatecznego produktu z amerykańską agencją wywiadowczą NSA (ang. *National Security Agency,* Agencja Bezpieczeństwa Narodowego), co wywołało kontrowersje. Sądzono, iż skoro agencja wywiadowcza ma wpływ na postać algorytmu to z pewnością umieści w nim ukryte luki, co pozwoli osobom je znającym, na łatwe złamanie szyfru i przechwycenie wiadomości. Pomimo tego, algorytm został opatentowany, a wątpliwości nigdy nie zostały potwierdzone bądź zdementowane [15].

Głównym mechanizmem odpowiadającym za szyfrowanie w algorytmie DES są tzw. **sieci Feistela**. Zostały one opracowane przez pracownika IBM Horsta Feistela w latach siedemdziesiątych. Ich głównym zadaniem jest uogólnienie procesu szyfrowania bądź deszyfrowania do jednego formatu, bez względu na to, czy funkcja użyta w danym procesie jest odwracalna. Schemat podstawowej sieci Feistela przedstawiono na rysunku 3.1, a szerszy ich opis wraz z bardziej rozwiniętymi przykładami znajduje się w pracy [6].

Rysunek 3.1. Schemat blokowy sieci Feistela.

Jak widać na rysunku 3.1, blok tekstu jawnego dzielony jest na dwie równe części – w przypadku algorytmu DES są to dwa 32-bitowe bloki powstałe poprzez podział 64-bitowego bloku wejściowego. Prawa część () podawana jest na wejście pewnej funkcji, która tak naprawdę odpowiedzialna jest za szyfrowanie, a wynik jej z kolei jest poddawany operacji alternatywy wykluczającej wraz z lewą stroną () bloku wejściowego. Dane wyjściowe z szeregu tych operacji stają się nową prawą stroną, a pierwotna prawa strona danych wejściowych staje się lewą stroną. Oba bloki są następnie złożone w całość, co stanowi nowy szyfrogram (zgodnie z rysunkiem 3.1) lub wejście na następną rundę algorytmu (na przykład w algorytmie DES, co zostanie opisane w późniejszej części rozdziału).

DES w procesie szyfrowania stosuje 16 rund Feistela, jednak dane wejściowe przed przystąpieniem do nich są poddawane tak zwanej **permutacji początkowej** (ang. IP *initial permutation*). Bity zgromadzone w bloku o długości 64 liczb przestawiane są na z góry określone miejsca – na pierwsze miejsce trafia 58 bit, na drugie 50 itd. Całość permutacji można zaobserwować na rysunku 3.2.

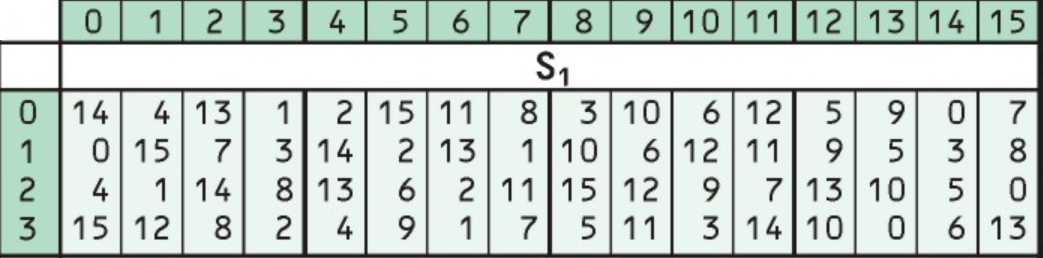
Po przeprowadzeniu permutacji początkowej, następuje podział zgodny z siecią Feistela, a szyfrowanie, zgodnie z zamysłem tej sieci, odbywa się w głównej mierze dzięki funkcji Feistela (F). W algorytmie DES składa się ona z następujących operacji:

Rysunek 3.2. Kolejność bitów w permutacji początkowej.

* Operacja E (od angielskiego słowa expansion oznaczającego rozszerzenie) – jako że blok tekstu jawnego ma 32 bity długości, a klucz rundy 48, należy zrównać ich rozmiar. Osiąga się to za pomocą odpowiedniej permutacji, w której niektóre bity są duplikowane. Kolejność bitów uzyskanych w wyniku permutacji przedstawiono na rysunku 3.3.



Rysunek 3.3. Kolejność bitów w operacji E.

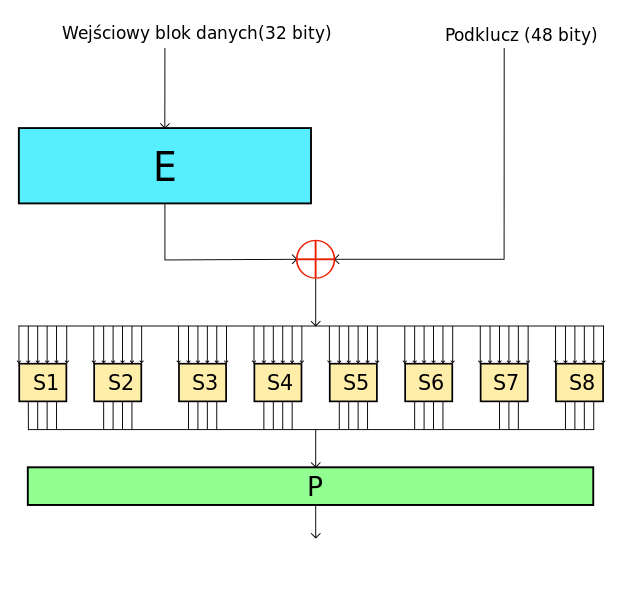
* Operacja XOR z kluczem rundy (proces budowania klucza rundy opisany zostanie w dalszej części rozdziału).
* Operacja zamiany – 48 bitów wynikowych z poprzedniej operacji zostaje podzielonych na osiem 6-bitowych grup. Następnie każdej grupie w wyniku nieliniowej operacji s-box (przypominającej look-up table znaną z elektroniki) przypisywana jest 4-bitowa liczba zgodna z tabelą. Każda tabela składa się z 4 wierszy i 16 kolumn. O numerze wiersza stanowi liczba powstała w wyniku złożenia pierwszego i ostatniego bitu grupy (liczba z zakresu 0-3), a o kolumnie decyduje liczba powstała ze złożenia bitów z miejsc 2-5 (liczba z zakresu 0-15). Przykładowy s-box zaprezentowano na rysunku 3.4, a pozostałe znaleźć można w różnych źródłach literatury, np. w [17].

Rysunek 3.4. Przykładowa tablica s-box dla pierwszej grupy 6-bitów. Zaprezentowane liczby są w postaci decymalnej

* Operacja P (od angielskiego *permutation,* czyli permutacji) – jako że w operacji s-box w każdej grupie z 6 bitów powstaje 4-bitowa liczba, cały blok danych skraca się z 48 do 32 bitów. Zostaje on poddany kolejnej z góry określonej permutacji. Kolejność bitów zaprezentowano na rysunku 3.5.

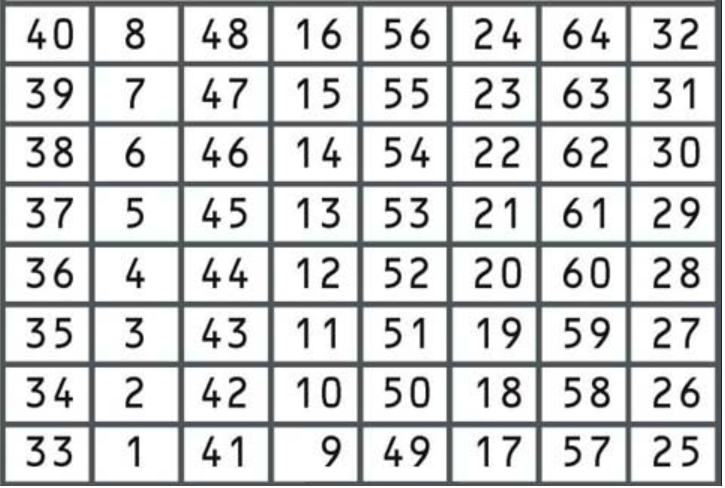


Rysunek 3.5. Tabela permutacji P.

Schemat blokowy powyższych operacji stanowiących funkcję F przedstawiono na rysunku 3.6.

Rysunek 3.6. Schemat blokowy funkcji F algorytmu DES.  
Źródło: https://commons.wikimedia.org/wiki/File:Data\_Encryption\_Standard\_InfoBox\_Diagram-pl.svg

Zgodnie z rysunkiem 3.1, wyjście funkcji Feistela jest dodawana modulo 2 z lewą stroną podziału na 32 bity, a wynik tej operacji stanowi nową prawą stronę w następnej rundzie. Lewą stroną zaś zostaje pierwotna prawa strona danej rundy. Schemat ten, jak już wspomniano, powtarzany jest 16 razy. Na koniec, lewa połowa (32 bity) jest zamieniana z prawą, a algorytm wieńczy ostatnia permutacja FP (od angielskiego *final permutation* – permutacja końcowa, zwana również ). Kolejność bitów w tej operacji przedstawiono na rysunku 3.7.

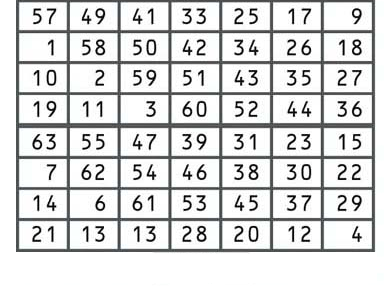


Rysunek 3.7. Kolejność bitów w permutacji końcowej.

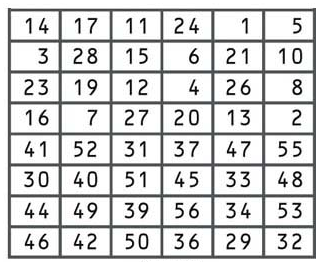
Wyjście z permutacji końcowej stanowi szyfrogram. Dzięki strukturze sieci Feistela, przekształcenie szyfrogramu w tekst jawny dla algorytmu DES polega na powtórzeniu tych samych operacji w tej samej kolejności, z tym, że klucze rundy podawane są w odwrotnej kolejności niż w procesie szyfrowania.

Kluczowym etapem będącym wstępem do procesu szyfrowania jest mechanizm tworzenia klucza rundy. Działa on według następujących kroków:

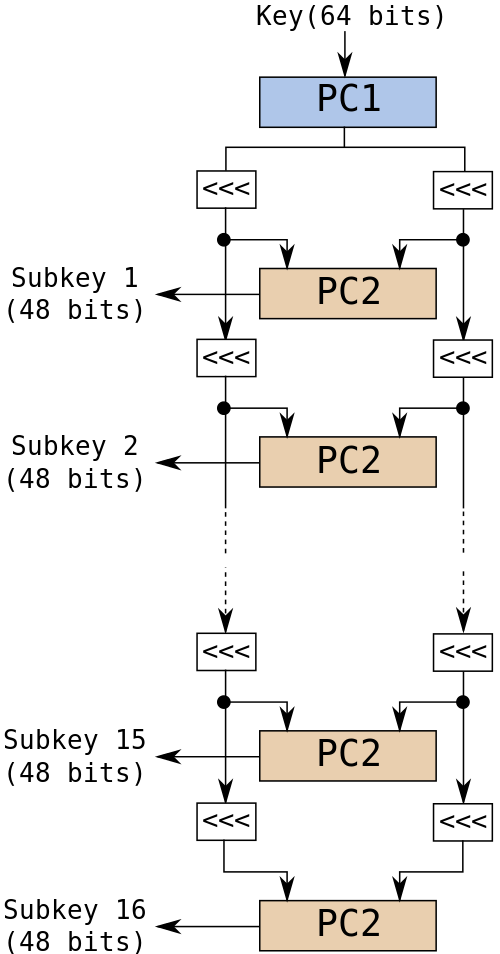
* Na wejście algorytmu podaje się 64 bitowy klucz, w którym co ósmy bit stanowi bit parzystości – bity parzystości są ustawione w taki sposób, żeby każdy bajt składał się z nieparzystej liczby jedynek. Jeśli ta zasada nie jest spełniona, klucz jest uznany jako niepoprawny.
* Bity parzystości są odrzucane, a pozostałe 56 poddawane są operacji permutacji PC1. Kolejność bitów przedstawiono na rysunku 3.8 (podana numeracja odnosi się do 64 bitowego klucza, aczkolwiek jak łatwo można zauważyć, bity parzystości nie są używane)



Rysunek 3.8. Kolejność permutacji PC1.

* Uzyskane 56 bitów dzieli się na pół, a uzyskane podciągi oznacza się kolejno i .
* Dla oblicza się:
* W powyższych wzorach oznacza klucz rundy, stanowi operację cyklicznego przesunięcia w lewo o jedno bądź dwa miejsca (jedno dla , dwa w przeciwnym przypadku). to permutacja przeprowadzona na 56 bitowym ciągu powstałym w wyniku połączenia oraz . Kolejność bitów w permutacji pokazano na rysunku 3.9.

Rysunek 3.9. Kolejność bitów w permutacji PC2

Schemat blokowy tworzenia podkluczy zaprezentowano na rysunku 3.10. Należy zwrócić uwagę, że jedyne operacje wykonywane na oryginalnym kluczu to przesunięcia oraz zamiana miejsc – bity nie zmieniają swojej wartości. W związku z powyższym, istnieją predefiniowane tablice, które bity początkowego klucza używane są w danej rundzie (taka informacja zawarta jest przykładowo w [17]). Oznacza to, że przejęcie jednego klucza rundy, powoduje znajomość większości bitów oryginalnego klucza i znaczne zawężenie dalszych poszukiwań. Ponadto, niektóre klucze algorytmu są tzw. kluczami słabymi (klucze rundy są identyczne), a niektóre tzw. półsłabymi (dla tej samej danej wejściowej zwracany jest identyczny szyfrogram).

Rysunek 3.10. Schemat blokowy tworzenia ciągu podkluczy w algorytmie DES

Jak już wspomniano wcześniej, algorytm DES od początku budził wątpliwości co do swojego bezpieczeństwa, między innymi ze względu na postać s-boxów czy też na niewielką liczbę bitów klucza. W celu utrudnienia złamania szyfru, powstał jego wariant zwany 3DES polegający na trzykrotnym zastosowaniu na tekście jawnym pierwotnego wariantu – najpierw następuje szyfrowanie za pomocą jednego klucza, następnie szyfr jest deszyfrowany przy użyciu drugiego, a następnie ponownie szyfrowany za pomocą trzeciego. Klucze mogą być różne, jeden może być inny od pozostałych bądź wszystkie mogą być takie same (otrzymuje się wtedy algorytm wstecznie kompatybilny z szyfrem DES). Pomimo rozszerzenia długości klucza i poprawy bezpieczeństwa, 3DES również był podatny na ataki, w związku z czym w 2005 roku wycofano go jako standard, a jego miejsce wybrany został algorytm AES, zwycięzca konkursu z 2001 roku. Szczegóły dotyczące tegoż algorytmu znajdują się w następnym rozdziale.

# AES

Algorytmem, który po czterech latach trwania konkursu został wybrany przez NIST oraz otrzymał tytuł *Advanced Encryption Standard* był algorytm Rijndael. Nazwa owego algorytmu pochodzi od połączonych nazwisk jego twórców – Joana Daemena oraz Vincenta Rijmena. Dzięki swojej stosunkowo prostej budowie, można stworzyć jego implementację na wielu platformach i architekturach. Pomimo tej prostoty można go uznać za bezpieczny, albowiem do dzisiaj nie udało się stworzyć skutecznej metody ataku, która pozwoliłaby odtworzyć klucz.

Pod względem wysokopoziomowym, algorytm składa się z 3 kroków. Są to kolejno:

* rozszerzenie klucza – z klucza wejściowego (może być on 128, 192 lub 256 bitowy) podanego do algorytmu, tworzy się ciąg kluczy, inny dla każdej rundy,
* runda wstępna – do bloku wejściowego (16-elementowej macierzy bajtów, zwanej stanem) za pomocą operacji XOR dodawany jest klucz rundy (w tym przypadku, oryginalny klucz),
* runda – ciąg czterech operacji. Liczba rund zależna jest od długości klucza – 10 rund przeprowadza się dla klucza 128-bitowego, 12 dla 192 oraz 14 dla 256. Operacje, które są wykorzystywane w rundzie to:
  + zamiana bajtów,
  + zamiana wierszy,
  + zamiana kolumn,
  + dodanie klucza rundy.

Ostatnia runda odbywa się bez operacji zamiany kolumn. W dalszej części rozdziału znajdzie się opis każdej operacji, a także procesu rozszerzania klucza.

Rozszerzenie klucza jest operacją ważną z punktu widzenia bezpieczeństwa. Pozwala ono na użycie unikalnego klucza w każdej rundzie, w związku z czym każda runda jest inna. Gdyby była identyczna, wynikałaby z tego następująca zależność (4.1):



Gdzie:

 – dwa dane bloki tekstu jawnego – przekształcenie rundy – szyfrogramy odpowiadające .

Z oczywistych względów, zależność taka jest niebezpieczna i wyklucza użycie szyfru na szerszą skalę.

W celu opisania procesu rozszerzania klucza, należy zdefiniować następujące wielkości i transformacje [4]:

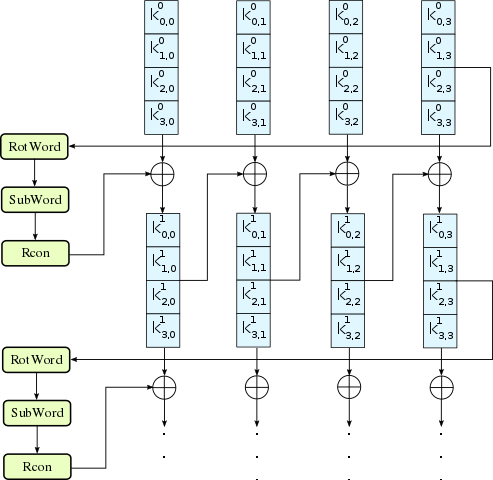
* – długość klucza liczona w 32-bitowych słowach, odpowiednio 4 dla klucza 128, 6 dla 192 i 8 dla klucza 256-bitowego,
* – 32-bitowe słowa oryginalnego klucza,
* – liczba rund, dla których potrzebny jest klucz rundy (11 dla klucz 128, 13 dla 129 oraz 15 dla klucza 256-bitowego),
* – 32-bitowe słowa klucza rozszerzonego,
* Transformacja RotWord przesuwająca bajty w słowie według następującego schematu:
  + , gdzie:  
     to kolejne bajty w słowie
* Transformacja SubWord przypisująca danemu bajtowi określone pole z tzw. tablicy   
  S-box. Szerzej ten proces zostanie opisany przy okazji wyjaśnienia operacji zamiany bajtów w rundzie algorytmu AES.

Przy powyższych założeniach, klucz rozszerzony dany jest zależnością (4.2) [4].



We wzorze (4.2) jest stałą zdefiniowaną z tabeli (4.3). Wartość oznacza indeks, zaś odpowiadającą mu wartość w systemie szesnastkowym.

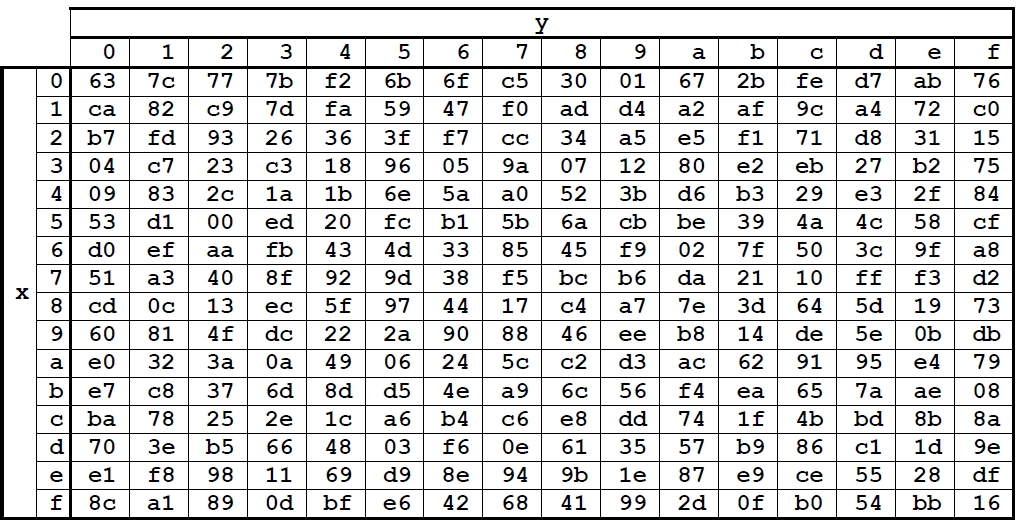


Maksymalny indeks potrzebny w procesie rozszerzania klucza wynosi 10, aczkolwiek można znaleźć wartość dla indeksu większego. Aby to uczynić, należy skorzystać z formuły matematycznej zawartej m.in. w [4]. Graficzna reprezentacja procesu tworzenia kluczy znajduje się na rysunku 4.1.

Rysunek 4.1. Schemat rozszerzania klucza w algorytmie AES.  
Źródło: https://commons.wikimedia.org/wiki/File:AES-Key\_Schedule\_128-bit\_key.svg

Jak już wcześniej wspomniano, dzięki zastosowaniu innego klucza dla każdej rundy, wynik jej działania zależy nie tylko od danych wejściowych, ale także od numeru rundy, w którym są one przetwarzane. We wcześniejszej części rozdziału, etapy rundy zostały wymienione, natomiast poniżej znajdzie się krótki opis każdego z nich.

Pierwszą przeprowadzaną w każdej rundzie operacją jest **zamiana bajtów** (ang. *sub bytes*). Jest to jedyne nieliniowe przekształcenie w szyfrze, bez niego odwrócenie rundy wymagałoby użycia bardzo podstawowych przekształceń algebraicznych, co wykluczałoby użycie szyfru [1]. Operacja polega na podstawieniu danej (liczby w systemie szesnastkowym) z tabeli w zamian za liczbę wejściową – pierwsza cyfra liczby wejściowej odpowiada za numer wiersza, a druga za numer kolumny. Dane w tabeli podstawień (tak zwany **s-box**) zostały dobrane tak, żeby jak najmniej ze sobą korelować. Rysunek 4.2 przedstawia oryginalną tabelę zawartą w standardzie [4].



Rysunek 4.2. Tablica s-box używana w operacji zamiany bajtów.

Jako że s-box jest bardzo ważnym punktem algorytmu (ze względu na wprowadzenie jedynej nieliniowości), forma tabeli podlega wielu rozważaniom na temat poprawy lub modyfikacji. Ich forma bazuje na wielomianach, zatem przestrzeń poszukiwań jest relatywnie duża. Badania na temat wpływu rzędu wielomianu oraz jego formy na wariancję w danych wypełniających tabelę można znaleźć w [20]. Równie ciekawe rozwiązanie korzystające z nietypowych technik (algorytm genetyczny i głębokie sieci neuronowe) można znaleźć w [7].

Kolejnym etapem rundy jest operacja **zamiany wierszy** (ang. *shift rows*). Działa ona bliźniaczo podobno do operacji RotWord używanej w procesie rozszerzania klucza. W obu przypadkach stosuje się cykliczne przesunięcie w lewo, z tą różnicą, iż w rundzie algorytmu przesunięciu ulegają bajty w wierszu, a liczba miejsc, o które następuje przesunięcie zależne jest od numeru wiersza – pierwszy wiersz przesuwany jest w lewo o jedno, drugi o dwa, a trzeci o trzy miejsca. Czwarty wiersz nie jest przesuwany (jest to de facto równe przesunięciu o 4 miejsca, co zachowuje spójność operacji). Dzięki zamianie wierszy, kolumny stają się zależne od siebie, co zwiększa kryptograficzną moc algorytmu.

Przedostatnią operacją w rundzie (ze względu na swoją liniowość i ograniczenie obliczeń, nie występuje w ostatniej rundzie cyklu) jest operacja **zamiany kolumn** (ang. *mix columns*). W przeciwieństwie do operacji zamiany wierszy, gdzie przekształceniu poddawane były wiersze macierzy stanu, jak sama nazwa wskazuje, zamiana kolumn bazuje na kolumnach. Operacja ta sprowadza się do mnożenia wektora wejściowego z macierzą współczynników zgodnie ze wzorem (4.4)



Operacja mnożenia wykonywana jest w tzw. **przestrzeni Galoisa**, gdzie operuje się na liczbach maksymalnie 8-bitowych, a operacje zdefiniowane są tak, iż wynik również należy do tej przestrzeni (szczególnie ważne jest to przy operacji mnożenia, gdzie wynik może łatwo wykroczyć poza zakres). Szczegóły implementacji mnożenia w tej przestrzeni można znaleźć w [18]. Dzięki zamianie kolumn, zmiana jakiegokolwiek bajtu w macierzy stanu algorytmu, zmienia wszystkie pozostałe.

Ostatnią operacją rundy jest **dodanie klucza rundy** za pomocą operacji XOR. Jak już wspomniano przy opisie rozszerzania klucza, jest to kluczowa czynność, albowiem powoduje dywersyfikacje rund względem siebie, jako że każda posiada unikalną daną będącą właśnie kluczem rundy.

Jedna runda algorytmu Rijndael nie jest szczególnie bezpieczna kryptograficznie, aczkolwiek kilkunastokrotne powtórzenie tworzy algorytm, którego do tej pory nie udało się efektywnie złamać [1].

Proces odkodowania szyfrogramu przebiega odwrotnie do procesu szyfrowania przedstawionego powyżej. Odwrotnością operacji dodania klucza jest jego ponowne dodanie (ze względu na własność operacji XOR). Żeby odwrócić zamianę kolumn należy pomnożyć poprzez macierz odwrotną do macierzy z wzoru (4.4). Zamiana wierszy jest trywialnie odwracalna poprzez przesunięcie cykliczne w prawo, zaś operacja s-box posiada odwrotną do tabeli z rysunku 4.2 tablicę przekształceń, którą można znaleźć m.in. w [4]. Operacje szyfrowania czy też deszyfrowania całej wiadomości, a nie jednego bloku są zgodne ze schematami przedstawionymi w rozdziale 2 na rysunkach 2.4-2.7. Algorytm AES można zaimplementować, aby wspierał oba przedstawione tryby pracy szyfru (a także inne niewspomniane w pracy).

Opisana powyżej struktura algorytmu Rijndael jest strukturą oryginalną, zgłoszoną w konkursie NIST. Jak podaje źródło [1], nie jest ono bezpośrednio stosowane. Opracowano rozwiązanie bazujące na specjalnych tablicach oraz instrukcjach XOR, które znacząco usprawnia działanie algorytmu wykonując operacje tożsame pod względem końcowego efektu. Ponadto, stosuje się również instrukcje natywne dla procesora danej architektury, co również przyspiesza proces szyfrowania i deszyfrowania. W projekcie związanym z pracą zostanie zawarta implementacja oryginalnego algorytmu, a uzyskane rezultaty zostaną porównane z wynikami zawartymi w innych źródłach literatury.

# RC6

Algorytm RC6 powstał jako rozwinięcie swojej poprzedniej wersji RC5, w taki sposób, aby spełniała założenia konkursowe AES opublikowane przez NIST. Nazwa szyfru rozwija się jako Rivest Cipher 6 i pochodzi od głównej osoby odpowiedzialnej za jego opracowanie – Ronalda L. Rivesta.

Często (robi to sam autor w [13]) używa się sformułowania RC6 jako szyfru zgłoszonego w konkursie, aczkolwiek RC6 (tak samo jak poprzednik) jest klasą algorytmów, a precyzyjną nazwą poszczególnych metod jest **RC6-w/r/b**, gdzie „w” oznacza bitową długość każdego z czterech rejestrów, w którym przechowywany jest blok wejściowy (zmienna w stanowi więc również o długości bloku wejściowego), „r” oznacza liczbę rund, a „b” liczbę bajtów klucza podanego przez użytkownika, który ulega procesowi rozszerzenia w fazie inicjalizacyjnej algorytmu. Do konkursu zgłoszona została wersja o 20 rundach oraz 32-bitowej długości rejestru. Klucz był wielkości 16, 24 lub 32 bajtów.

Algorytm RC6, jako rozwinięcie algorytmu RC5, miał również spełniać założenia dla swojej starszej wersji. Są to w szczególności [12]:

* Symetryczność szyfru – ten sam sekretny klucz powinien służyć zarówno do szyfrowania, jak i deszyfrowania.
* Dostępność algorytmu zarówno w rozwiązaniach sprzętowych, jak i oprogramowaniu. Oznacza to, iż szyfr powinien korzystać z podstawowych operacji dostępnych na typowych mikroprocesorach.
* Szybkość – implikuje to, iż algorytm operuje na słowach bitowych i to na nich wykonywane są operacje.
* Łatwość przenoszenia na procesory o różnej długości słowa bitowego. Przykładowo, zakładano, iż szyfr powinien z łatwością korzystać z całej długości 64-bitowego słowa i czerpać z tego korzyści pod względem szybkości. Ponadto, jako że długość bitów jest zmienną algorytmu, zmiana tego parametru powinna tworzyć nowy, niezależny algorytm.
* Algorytm powinien mieć strukturę iteracyjną – zmienna liczba rund, która pozwala wybrać użytkownikowi pomiędzy szybkością szyfrowania, a bezpieczeństwem algorytmu.
* Zmienna długość klucza – użytkownik powinien mieć kontrolę nad bezpieczeństwem i wybrać taką długość, która zapewnia bezpieczeństwo jego aplikacji.
* Prostota – dzięki prostemu schematowi działania, algorytm powinien być prosty w implementacji. Dodatkowo, autorzy zakładali, iż mały stopień skomplikowania zachęci innych do ewaluacji algorytmu pod względem bezpieczeństwa.
* Wysokie bezpieczeństwo przy zapewnieniu odpowiednich parametrów (liczby rund, długości klucza itp.)

Rozwiązanie przygotowane na konkurs NIST posiada jedną istotną różnicę – liczba bitów, o które następuje przesunięcie jest silniej zależne od danych wejściowych. Pozwala to zwiększyć losowość i utrudnić kryptoanalizę. Aby to osiągnąć, zwiększono liczbę rejestrów z dwóch do czterech oraz wprowadzono dwie rotacje na rundę.

Jak już wcześniej wspomniano, przy opracowaniu algorytmów dużą wagę przywiązano do szybkości działania. Uzyskano ją dzięki skorzystaniu z prostych operacji natywnych dla procesorów – przesunięć bitowych oraz mnożenia liczb całkowitych (w szczególności 32 bitowych, co okazało się również dużą korzyścią związaną z podwojeniem liczby rejestrów). W pracy [13] zdefiniowano następujące działania (5.1) – (5.6)

 - suma modulo

 - różnica modulo

 - alternatywa wykluczająca (XOR) w-bitowych słów

 - całkowitoliczbowy iloczyn modulo

 - przesunięcie bitowe a w lewo o lgw najbardziej znaczących bitów b

 - przesunięcie bitowe a w prawo o lgw najbardziej znaczących bitów b

Gdzie: . Mając tak zdefiniowane działania można przejść do opisu pseudokodu algorytmu. Jest on następujący:

**Wejście:**

* Blok tekstu jawnego przechowywany w czterech w-bitowych rejestrach: *A, B, C, D*. Pierwszy bajt tekstu jawnego trafia do najmniej znaczącego bajta A, a ostatni do najbardziej znaczącego bajta D.
* Liczba rund *r*
* Lista w-bitowych kluczy rund przechowywana w tablicy *S[0..2\*r+3]*

**Wyjście:**

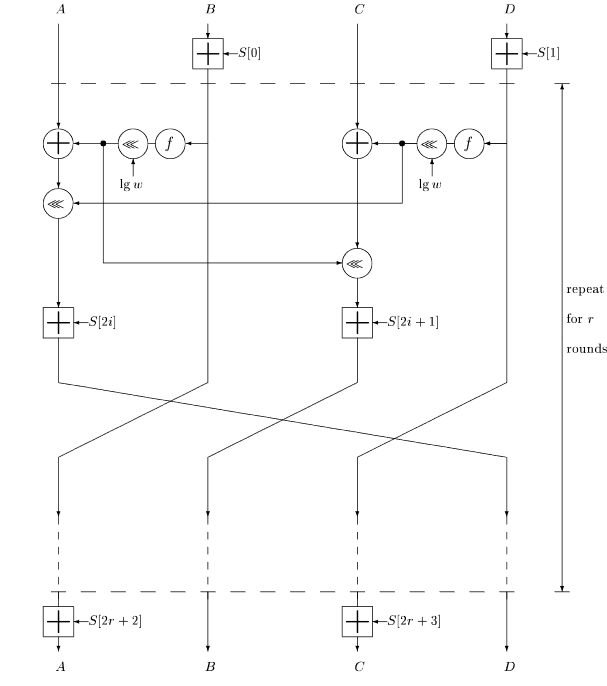
* Szyfr przechowywany w rejestrach *A, B, C, D.*

**Pseudokod:**

****

Gdzie  oznacza równoległe podstawienie rejestrów z prawej strony równania do lewej.

Schemat blokowy procesu szyfrowania przedstawiono na rysunku 5.1.



Rysunek 5.1. Schemat blokowy procesu szyfrowania RC6

Proces odszyfrowania wiadomości polega na odwróceniu czynności wykonanych podczas szyfrowania. Pseudokod funkcji odszyfrowującej jest zatem następujący:



**Wejście:**

* Szyfrogram przechowywany w czterech w-bitowych rejestrach: *A, B, C, D*.
* Liczba rund *r*
* Lista w-bitowych kluczy rund przechowywana w tablicy *S[0..2\*r+3]*

**Wyjście:**

* Tekst jawny przechowywany w rejestrach *A, B, C, D.*

**Pseudokod:**

****

Kluczowym punktem w inicjalizacyjnych krokach algorytmu jest rozszerzanie klucza. Na podstawie ciągu bajtów dostarczonych przez użytkownika tworzony jest zestaw kluczy, gdzie oznacza liczbę rund. W procesie rozszerzania klucza wprowadza się dwie stałe pomocnicze, tzw. **„*magic numbers****”*, których wartość zależy od zmiennej wprowadzonej wcześniej jako liczba bitów rejestrów A, B, C, D. Owe zmienne definiuje się według następujących wzorów (5.7) – (5.8).





gdzie:

Dla 16 bajtowego bloku danych (wielkość bloku, którą miał szyfrować algorytm w konkursie AES), , . Przy powyższych założeniach (oraz zachowując symbole działań z procesu szyfrowania), pseudokod procesu rozszerzania klucza jest następujący:

**Wejście:**

* Dostarczony przez użytkownika b-bajtowy klucz podstawowy. Jest on ładowany w tablicę c-słów *L[0..c-1]* w konwencji little-endian – pierwszy bajt klucza jest przechowywany jako bajt niskiego rzędu *L[0]*, a *L[c-1]* w razie potrzeby zostaje wypełnione zerami.
* Liczba rund *r*

**Wyjście:**

* Tablica w-bitowych kluczy rund *S[0..2r+3]*

**Pseudokod:**

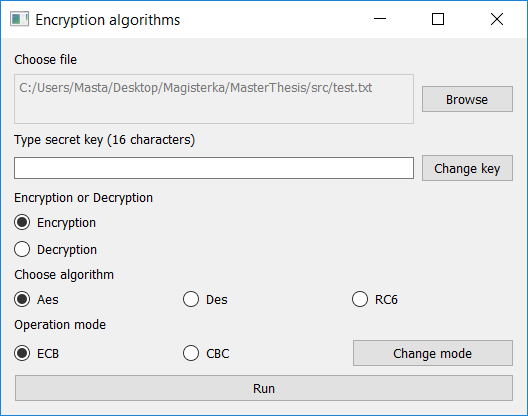
****

Liczba jest równa liczbie bajtów klucza podzielonej przez liczbę bajtów pojedynczego rejestru używanego w procesie szyfrowania. W wyjątkowym przypadku, kiedy klucz jest pusty, przyjmuje ona wartość 1.

Dokładne studium nad architekturą algorytmu można znaleźć w [13]. Autorzy opisują krok po kroku każdą operację zaimplementowaną w algorytmie wraz z wyjaśnieniem sposobu, w jaki sposób została ona wybrana. Praca ponadto zawiera testy wydajnościowe na ówczesnej architekturze dla różnych rozwiązań (w szczególności języków) oprogramowania – głównie zawarte są rozważania na temat szybkości działania, aczkolwiek można również znaleźć krótkie podsumowanie dotyczące zużycia pamięci. Należy zauważyć, iż algorytm RC6, w przeciwieństwie do większości algorytmów, w tym dwóch opisanych w poprzednich rozdziałach, nie korzysta z predefiniowanych s-boxów, co znacząco zmniejsza nakład pamięciowy. Na koniec, dla osób, które chcą zaimplementować własną wersję algorytmu, autorzy zawarli wektory testowe (dane wejściowe, klucz oraz oczekiwany szyfrogram), co pozwoliło przetestować algorytm stworzony w ramach projektu związanego z niniejszą pracą oraz potwierdzić poprawność działania szyfru.

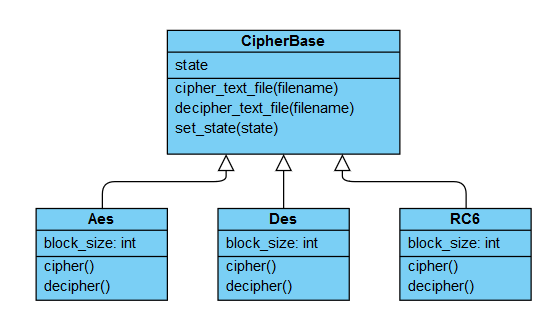
# Opis zaimplementowanego programu

W ramach projektu związanego z pracą został zaimplementowany, przy użyciu języka Python w wersji 3.6, program komputerowy pozwalający na szyfrowanie bądź odszyfrowanie pliku tekstowego kodowanego w trybie ASCII. Posiada on graficzny interfejs użytkownika zaprezentowany na rysunku 6.1.



Rysunek 6.1 Graficzny interfejs programu szyfrującego.

Jak widać, menu jest dosyć intuicyjne i pozwala na szybki wybór między dostępnymi algorytmami, wybranie ścieżki do pliku tekstowego, na którym przeprowadzona ma zostać operacja szyfrowania bądź szyfrowania (kolejna z opcji do wyboru). Opcjonalnie, użytkownik może podać swój klucz do kodowania wiadomości, co pozwoli na przechowywanie wiadomości w postaci zaszyfrowanej. Jeśli użytkownik nie poda klucza, zostaje użyty domyślny, wygenerowany losowo (w sposób kryptograficzny) przy starcie programu. Operacja zmiany klucza oraz trybu pracy algorytmu wymagają dodatkowego potwierdzenia poprzez naciśnięcie odpowiedniego przycisku. Po ustawieniu preferowanych opcji oraz wyborze ścieżki do pliku, należy przycisnąć przycisk „Run”. W zależności od wybranej akcji, w katalogu wywołania programu pojawi się plik tekstowy *cipher.txt* (w przypadku szyfrowania) bądź *decipher.txt* w przypadku deszyfrowania. Szczególną uwagę należy zwrócić przy operacjach na domyślnym kluczu – jest on tworzony na początku programu i ważny tylko przez jedno uruchomienie. Oznacza to, że po wyłączeniu programu, niezaszyfrowana treść pliku *cipher.txt*, w którym znajdują się dane zaszyfrowane domyślnym kluczem, zostanie utracona.

 Z punktu widzenia programistycznego, prostota ta jest możliwa dzięki zastosowaniu prostej wirtualizacji klas – każdy algorytm posiada dedykowaną klasę, która dziedziczy po klasie bazowej CipherBase. Klasa ta odpowiada za przechowywanie wspólnych parametrów dla szyfrów oraz tworzy bazę pod metody, które klasa pochodna przeciąża. Dodatkowo, w klasie bazowej zaimplementowane zostały cechy wspólne dla algorytmów takie jak padding czy obsługa plików. Diagram UML opisanych klas znajduje się na rysunku 6.2.

Rysunek 6.2 Diagram klas szyfrów zaimplementowanych w programie.

Rysunek 6.2 obrazuje, w jaki sposób następuje wirtualizacja procesu szyfrowania – metody klasy bazowej cipher\_text\_file oraz decipher\_text\_file­ odpowiadają za odczyt danych z pliku tekstowego oraz wydzielają blok danych (wielkości zmiennej block\_size unikalnej dla danej klasy). Następnie, blok ładowany jest do zmiennej state (za pomocą funkcji set\_state, której implementacja również zależna jest od klasy), a na koniec wywoływane są funkcje cipher bądź decipher, w których znajduje się właściwa implementacja działania danego algorytmu. Zaprezentowany schemat dziedziczenia jest prosty, aczkolwiek umożliwia konsolidację klas, co z kolei przekłada się na prostotę wyboru algorytmu w graficznym interfejsie.

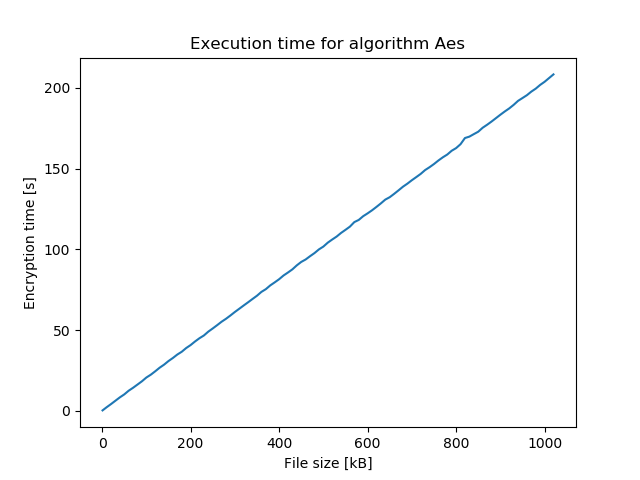
Do uruchomienia programu potrzebne są następujące komponenty:

* Interpreter języka Python w wersji 3.6
* Zainstalowane następujące pakiety:
  + Numpy (używane do operacji na macierzach, liczbach oraz bitach),
  + PyQt4 (używane do interfejsu graficznego),
  + Matplotlib oraz pickle (opcjonalnie, w celu włączenia skryptów testowych).

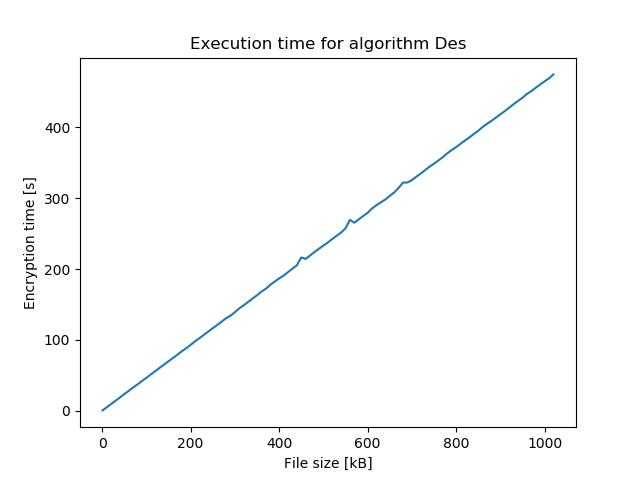
Przy spełnieniu powyższych wymagań, uruchomienie programu sprowadza się do pobrania kodu źródłowego (dostępnego w załącznikach do niniejszej pracy bądź pod adresem <https://github.com/Mastah95/MasterThesis>). Po pobraniu, należy uruchomić interpretację pliku algorithms\_gui.py, np. za pomocą instrukcji Python3 algorithms\_gui.py wywołanej z odpowiedniej konsoli (mającej dostęp do interpretera). Dodatkowo, w repozytorium znajdują się skrypty pomocnicze, które pozwoliły wygenerować wykresy zaprezentowane w rozdziale 7.

# Porównanie algorytmów

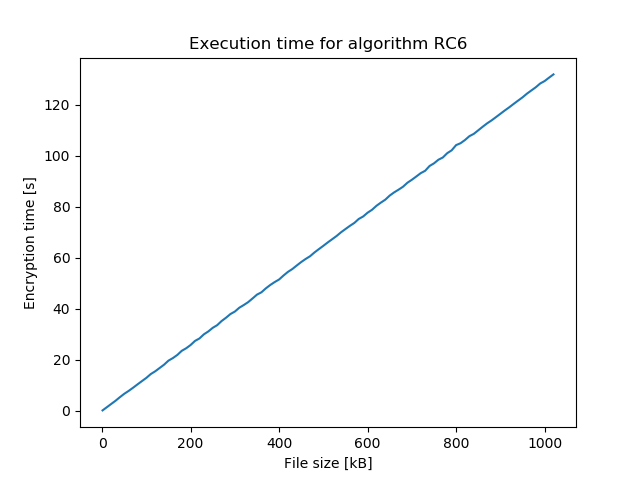
W celu ewaluacji algorytmów zaimplementowanych w programie opisanym w rozdziale 6. Oraz niejako potwierdzeniu badań NIST podczas konkursu na algorytm AES przeprowadzono badania na temat szybkości algorytmów, co było jednym z kryteriów wyboru najlepszego algorytmu. Stworzono pliki tekstowe, wypełnione losowymi znakami tekstowymi, o wielkości od 10 do 1020KB z krokiem co 10KB oraz jeden plik o wielkości 1KB. Podane pliki były następnie szyfrowane przez każdy z algorytmów, a po ukończonym procesie zostały zbierane dane dotyczące czasu szyfrowania. Należy zwrócić uwagę, że czas dotyczy tylko przebiegu enkrypcji, jako że dla domyślnej wielkości klucza (16-bajtów), czas tworzenia kluczy rund był pomijalnie mały, zatem został zignorowany. Wykresy czasu szyfrowania w funkcji wielkości pliku przedstawiono na rysunkach 7.1 – 7.3.



Rysunek 7.1. Czas szyfrowania dla algorytmu AES

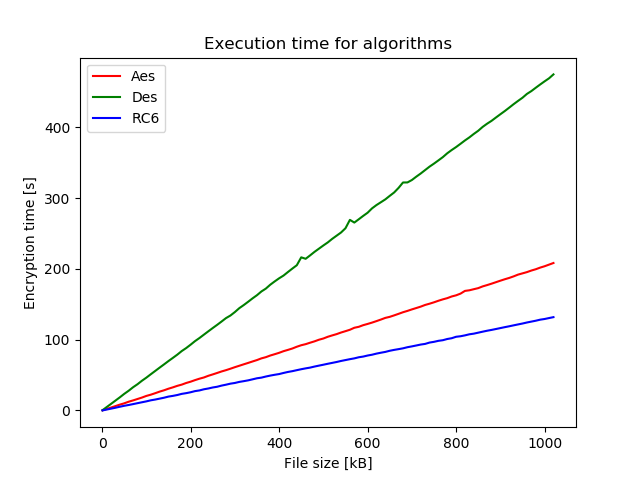


Rysunek 7.2. Czas szyfrowania dla algorytmu DES



Rysunek 7.3. Czas szyfrowania dla algorytmu RC6

Połączenie powyższych wykresów w jeden znajduje się na rysunku 7.4.



Rysunek 7.4. Czas szyfrowania poszczególnych algorytmów

Sprzęt, na którym dokonano pomiarów ma następujące parametry:

* Procesor: Intel Core i5-7300HQ 2.50 GHZ.
* Pamięć ram: 8GB.
* Typ systemu: 64-bitowy system operacyjny, procesor x64.
* Dysk: SSD o pojemności 120GB (w tym 28GB wolne).

Tabelaryczne zestawienie wyników z ograniczoną liczbą pomiarów (co dziesiąty) oraz zawierający wyliczoną średnią liczbę bloków na sekundę dla każdego algorytmu zawiera tabela 7.1.

Tabela 7.1. Liczbowe zestawienie czasu szyfrowania dla poszczególnych algorytmów oraz średnia liczba bloków na sekundę.

|  |  |  |  |
| --- | --- | --- | --- |
| Wielkość pliku [KB] | Czas szyfrowania AES [s] | Czas szyfrowania DES [s] | Czas szyfrowania RC6 [s] |
| 1 | 0,22 | 0,47 | 0,12 |
| 10 | 2,12 | 4,65 | 1,28 |
| 100 | 20,57 | 46,38 | 12,87 |
| 200 | 40,68 | 92,93 | 25,76 |
| 300 | 61,16 | 139,00 | 38,87 |
| 400 | 81,45 | 186,68 | 51,44 |
| 500 | 101,67 | 233,06 | 64,64 |
| 600 | 122,24 | 279,30 | 77,62 |
| 700 | 142,76 | 325,47 | 90,53 |
| 800 | 162,65 | 372,07 | 104,11 |
| 900 | 183,32 | 418,39 | 116,38 |
| 1000 | 203,72 | 464,88 | 129,22 |
| Bloki na sekundę (średnia wartość) | 313,50 | 274,72 | 494,69 |

Na podstawie wykresów 7.1 – 7.4 oraz tabeli 7.1 można wysnuć kilka wniosków. Po pierwsze, algorytmom zaimplementowanym w programie daleko do wersji optymalnych. Przykładowo na architekturze z 2000 roku o dużo gorszych parametrach, wyniki liczby bloków na sekundę dla algorytmu RC6, zaprezentowanym w [13], są 30 razy lepsze (i to dla najwolniejszej wersji algorytmu, zaimplementowanej w Javie). Również w innych źródłach porównujących algorytmy, np. w [8] osiągi, zarówno dla szyfru RC6, jak i Rijndael, są znacznie lepsze niż w zaproponowanym w ramach niniejszej pracy rozwiązaniu, aczkolwiek w pracy tej autorzy nie podali ani architektury, ani środowiska, w którym zaimplementowany został kod.

Przyczyn nieoptymalności rozwiązań jest kilka. Pierwszą i najważniejszą jest język wybrany do implementacji algorytmów. Pomimo swojej intuicyjności i wielu zalet, Python w środowisku programistów uważany jest za wolny. Składa się na to kilka czynników, m.in.:

* Python jest językiem interpretowanym, a nie kompilowanym. W językach kompilowanych, przy zamianie kodu w danym języku na kod maszynowy, nowoczesne kompilatory dokonują optymalizacji rozwiązań i potrafią przemienić nieoptymalne rozwiązanie na optymalny kod maszynowy, co przyczynia się do skrócenia czasu egzekucji
* Jeden proces (program) języka Python może mieć uruchomiony tylko jeden wątek. Jest to zasługa mechanizmu GIL (ang. *Global Interpreter Lock*), który dba o to, żeby interpreter obsługiwał jedynie jedną operację bez względu na to, ile wątków stworzonych jest w programie. Dla szyfrów, wada ta może nie mieć dużego znaczenia ze względu na to, że wielowątkowość nie jest bardzo pożądana (zwłaszcza dla trybu pracy CBC szyfru).
* Dynamiczne typowanie. W języku Python przy deklaracji zmiennych nie używa się ich typów, są one dedukowane. Jest to wygodne, jednak powoduje duży nakład czasowy, albowiem interpreter musi sam wydedukować typ zmiennej na podstawie danych do niej przekazanych lub operacji jakim dana zmienna jest poddawana. Również dla każdej operacji wielu zmiennych, typ każdej z niej musi być odczytany oraz zdefiniowany musi zostać typ wyniku działania, co również spowalnia działanie programu.

Wpływ na długi czas szyfrowania mają też aspekty architektury algorytmów oraz sposobu ich zaimplementowania. Jak wspomniano w rozdziale 4., algorytm Rijndael w swojej pierwotnej wersji można znacząco zoptymalizować zmieniając jego strukturę na operacje natywne dla procesora (co de facto jest główną zaletą innego algorytmu, RC6). W porównaniu opisanym w tym rozdziale nie dokonano takich usprawnień kierując się oryginalnym działaniem szyfru.

Głównym problemem dla algorytmu DES był brak bezpośredniej obsługi przez język Python pojedynczych bitów. Należało więc gromadzić dane w jednym bajcie, a następnie przemieniać go w tablicę pojedynczych bitów, co zwiększyło nakład obliczeniowy i mogło spowolnić szyfrowanie.

Szyfr RC6 w porównaniu dotyczącym szybkości wypadł najlepiej w porównaniu ze swoimi poprzednikami. Z pewnością wpływ na to miała dość prosta struktura używająca elementarnych operacji, głównie przesunięć bitowych oraz mnożenia całkowitoliczbowego.

Wyniki przedstawione i opisane powyżej stanowią dobre, aczkolwiek względne porównanie algorytmów między sobą. Nie powinny być jednak uznawane za wiążące w kwestii przydatności danego algorytmu, albowiem przeprowadzone zostały jedynie na jednej platformie oraz jednym środowisku. Jak pokazują badania zawarte w [16], czas szyfrowania potrafi być mocno zależny od użytego procesora, zwłaszcza dla szyfrów korzystających bezpośrednio z jego instrukcji. Przykładem takiego algorytmu jest RC6, którego szybkość znacząco zależy od procesora i może spać nawet ponad trzykrotnie (w niezmienionym oprogramowaniu uruchamianym na różnych procesorach).

Na podstawie powyższych rozważań można stwierdzić, iż szybkość szyfrowania bądź deszyfrowania nie powinna być głównym wyznacznikiem efektywności algorytmu. Doskonale tego świadoma była komisja NIST, która zdefiniowała znacznie szersze kryteria oceny algorytmów kandydujących do miana AES. Zdefiniowanymi kryteriami były: [10]

* Ogólne bezpieczeństwo algorytmu.
* Możliwości rozwiązań programowych.
* Działanie w środowiskach o ograniczonych zasobach
* Możliwości rozwiązań sprzętowych.
* Ataki na implementacje
* Różnice pomiędzy szyfrowaniem i deszyfrowaniem
* „Zwinność” klucza, czyli możliwość jego zmiany z jak najmniejszym nakładem czasowym oraz przy użyciu jak najmniejszej ilości zasobów.
* Inne zagrożenia oraz elastyczność algorytmu.
* Możliwość zrównoleglenia instrukcji

W tabeli 7.2 zawarto krótkie podsumowanie konkluzji zwartych w [10] dla każdego z kryteriów dla dwóch finalistów opisanych w niniejszej pracy, algorytmu RC6 oraz Rijndael.

Tabela 7.2. Porównanie algorytmów Rijndael oraz RC6 według kryteriów NIST.

|  |  |  |
| --- | --- | --- |
| Aspekt | Rijndael | RC6 |
| Generalne bezpieczeństwo | Brak znanych ataków, wystarczający margines bezpieczeństwa, aczkolwiek krytykowany za algebraiczną, przewidywalną strukturę. | Brak znanych ataków, wystarczający margines bezpieczeństwa. Pochwały za prostą strukturę umożliwiającą łatwą analizę bezpieczeństwa. |
| Możliwości rozwiązań programowych | Zadowalające osiągi w szyfrowaniu i deszyfrowaniu dla różnych platform (różno-bitowych procesorów oraz procesorów sygnałowych) – w grupie najlepszych algorytmów dla de/szyfrowania oraz rozszerzania klucza. Nieznaczny spadek osiągów dla wersji o większym kluczu. | Najszybszy na 32-bitowych platformach, osiągi spadają na 64-bitowych i generalnie są mocno zależne od architektury danego procesora bądź języka programowania. W grupie najlepszych algorytmów dla de/szyfrowania, w drugiej grupie dla procesu rozszerzania klucza. |
| Działanie w środowiskach o ograniczonych zasobach | Dobrze dostosowany do pracy w środowisku o niskich możliwościach pamięci, zwłaszcza przy rozdzieleniu procesu szyfrowania i deszyfrowania. Przy symultaniczności tych procesów, wzrasta wymaganie pamięci ROM, aczkolwiek nie przekroczyło dostępnych środków. | Małe zapotrzebowanie na pamięć ROM plusem w środowiskach o ograniczonych zasobach. Brak możliwości generacji kluczy na żądanie powoduje duże zapotrzebowanie na pamięć RAM, co powoduje niedostosowanie algorytmu do środowisk o ograniczonej pamięci. |
| Możliwości rozwiązań sprzętowych | Największa przepustowość ze wszystkich algorytmów oraz najmniejsze opóźnienie. Duża zużytych zasobów dla pełnej implementacji (szyfrowanie i deszyfrowanie), zwłaszcza przy zwiększonej liczbie bajtów klucza. | Dostosowany do implementacji sprzętowej, przy małym zużyciu zasobów. Przeciętna przepustowość, niezależna od długości klucza. |
| Ataki na implementacje | Duże możliwości w kwestii obrony przed znanymi atakami przy zachowaniu dość dużej szybkości działania. | Niskie możliwości obronne przy implementacji na platformach narażonych na ataki siłowe. Pomimo tego, niskie wymagania pamięciowe zaszumiania. |
| Różnice pomiędzy szyfrowaniem i deszyfrowaniem | Funkcje szyfrujące i deszyfrujące różnią się w takim stopniu, że zasoby w platformie FPGA konieczne do użycia szyfru zwiększają się o 60% kiedy chce się implementować zarówno szyfrowanie jak i deszyfrowanie. | Funkcje szyfrujące i deszyfrujące nieznacznie się różnią, zatem ich efektywność nie różni się znacząco. Implementacja równoczesnego szyfrowania i deszyfrowania zwiększa zasoby FPGA jedynie o 10%. |
| Zwinność klucza | Możliwość generowania klucza rundy na żądanie przy niewielkim obciążeniu zasobów. | Możliwość generacji klucza rundy jedynie dla szyfrowania (wymagane około 100 bajtów wartości pośrednich), klucze dla deszyfrowania muszą być wyliczone z wyprzedzeniem. |
| Inne aspekty i elastyczność | Szyfr może zaadaptować się do dowolnego rozmiaru bloku i klucza będącego wielokrotnością 32 bajtów, a także dowolnej zmianie liczby rund. | Rozmiar bloku, klucza oraz liczba rund są parametryzowane. Może wspierać klucze znacznie dłuższe niż 256 bitów. |
| Możliwość zrównoleglenia instrukcji | Doskonale przystosowany do zrównoleglenia szyfrowania jednego bloku | Ograniczone możliwości zrównoleglenia jednego szyfrowania jednego bloku |

W porównaniu do powyższych algorytmów DES odstaje zwłaszcza pod względem bezpieczeństwa i szybkości. Niewielka długość klucza powodująca podatność na atak siłowy oraz architektura bazująca na sprzęcie z późnych lat 70 nie pozwala wykorzystywać pełni potencjału nowoczesnych urządzeń i nawet niewielkie adaptacje, jak np. w pracy [11] nie dają zadowalających (w porównaniu do finalistów AES) rezultatów, albowiem rdzeń pozostał ten sam, a proces poszukiwania następcy wydaje się jak najbardziej rozsądny.

Ostatecznie, w ocenie sędziów NIST, algorytm Rijndael przewyższył RC6 (oraz innych finalistów) i otrzymał tytuł AES. Wybór ten spowodowany został satysfakcjonującym poziomem bezpieczeństwa (co generalnie dotyczyło wszystkich finalistów), dość dobrą szybkością szyfrowania i deszyfrowania, a przede wszystkim wszechstronnością algorytmu w odniesieniu do platformy (z czym RC6 miał problemy, np. 8-bitowymi urządzeniami typu smart-card). Po niemalże dwudziestu latach od wyboru, standard nie został zmieniony, zatem można stwierdzić, iż dokonano prawidłowego wyboru.

# Wnioski i spostrzeżenia

Niniejsza praca zawiera opis i porównanie trzech popularnych algorytmów szyfrujących – algorytmu DES, który został pierwszym standardem szyfrowania oraz dwóch finalistów w konkursie na nowy standard AES (algorytm mający zastąpić szyfr DES) – RC6 i Rijndael. Drugi z tych algorytmów został wybrany jako nowy standard i pozostaje nim do dzisiaj.

Na podstawie różnych źródeł potwierdzono, iż algorytm DES utracił miano bezpiecznego, zatem jego wymiana była konieczna. Ponadto, został on stworzony w czasach zamierzchłych z punktu widzenia postępu technologicznego, zatem jego architektura nie jest dostosowana do obecnych procesorów, a sam algorytm nie może w pełni skorzystać z rosnącej mocy obliczeniowej nowych urządzeń.

Porównano dwóch z pięciu finalistów konkursu AES oraz potwierdzono ogólną wyższość algorytmu Rijndael nad RC6. Pomimo tego, iż w przeprowadzonym badaniu, większą szybkością szyfrowania wykazał się drugi z algorytmów, wynik ten jest niemiarodajny, gdyż szyfr RC6 ma prędkość bardzo zależną od platformy, a ponadto jest to jedynie jedno z wielu kryteriów, które brano pod uwagę przy wyborze nowego standardu. Algorytm Rijndael wypadł lepiej w innych kategoriach, a w szczególności w możliwościach działania na wielu platformach – zarówno urządzeniach FPGA, jak i kartach o ograniczonych zasobach sprzętowych.

W ramach projektu związanego z pracą stworzono prosty program umożliwiający szyfrowanie bądź odszyfrowanie wiadomości za pomocą opisanych algorytmów. Dokonano sprawdzenia szybkości zaprogramowanych algorytmów, aczkolwiek wynik jest względnie niezadowalający. Sytuacja ta, szerzej opisana w rozdziale 7., spowodowana jest wyborem środowiska oraz brakiem głębszej optymalizacji (szczególnie dla algorytmu Rijndael, który został zaimplementowany w wersji pierwotnej, a nie tablicowej zaprezentowanej w [1]). W związku z powyższym, zaimplementowany program jest funkcjonalny jedynie dla niewielkich plików tekstowych.

Dalsze kroki rozwijania projektu nasuwają się same. Pierwszym i oczywistym, byłaby optymalizacja algorytmów, aczkolwiek w związku z ograniczeniami języka Python czas szyfrowania dalej mógłby być niezadowalający, zatem przepisanie programu w języku niższego poziomu (np. C++) również mogło by rozwinąć projekt. Kolejną oczywistą możliwością jest dodanie kolejnego algorytmu do porównania, przykładowo jednego z trzech pozostałych finalistów AES. Dzięki zastosowanej hierarchii klas opisanej w rozdziale 6., zadanie takie ograniczałoby się do implementacji samego algorytmu oraz dodania pola wyboru w graficznym interfejsie użytkownika. Pozostały kod został zaimplementowany w taki sposób, żeby dołączenie nowego algorytmu było proste i szybkie. W taki sam sposób napisany został kod odpowiedzialny za testy, których wyniki opisane zostały w rozdziale 7., więc dołączenie następnego algorytmu do porównania również jest możliwe.

Dzisiejszy świat bazuje na szyfrowaniu w praktycznie każdej dziedzinie życia. Tajność informacji była ważna od niepamiętnych czasów, ale wraz z cyfryzacją współczesnego świata, staje się ona kluczowa. Posiadanie standardu szyfrowania niesie za sobą wiele korzyści, ponieważ łatwiej śledzić badania na temat jego bezpieczeństwa oraz wprowadzać ewentualne poprawki. Współczesne szyfry budowane są w taki sposób, żeby atak siłowy był prawie niemożliwy. Niestety, dokładna ocena bezpieczeństwa algorytmu jest bardzo trudna bądź niemożliwa i bazuje się na empiryce, oznaczającej, że szyfr pozostaje bezpieczny, dopóki nikt go nie złamał. Jak podano w [1], szacuje się, że rozpowszechnienie komputerów kwantowych może być przełomem, który naruszy bezpieczeństwo dotychczasowych algorytmów. Zapewne ogłoszony zostanie wtedy nowy konkurs, który być może przyniesie nowy algorytm, niemożliwy do złamania nawet przy użyciu najnowocześniejszej technologii.

# Literatura

1. Aumasson J. P.: *Nowoczesna kryptografia. Praktyczne wprowadzenie do szyfrowania.*Warszawa: Wydawnictwo Naukowe PWN SA, 2018, s. 1-59. ISBN 978-83-01-19900-5.
2. Bhat B., Ali A. W., Gupta A.: *DES and AES performance evaluation.*  
   Noida: International Conference on Computing, Communication & Automation, 2015, s. 887-890.  
   doi: 10.1109/CCAA.2015.7148500
3. Cheng H., Ding Q.: *Overview of the Block Cipher*.   
   Harbin: 2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control, 2012, s. 1628-1631. *doi: 10.1109/IMCCC.2012.379*
4. FIPS PUB 197. *The official AES standard.* NIST, 21.10.2001
5. Floissac N., L'Hyver Y.: *From AES-128 to AES-192 and AES-256, How to Adapt Differential Fault Analysis Attacks on Key Expansion   
   Nara:* 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, 2011, s. 43-53.  
   doi: 10.1109/FDTC.2011.15
6. Hoang V. T., Rogaway P. *On Generalized Feistel Networks*, w: *Advances in Cryptology -- CRYPTO 2010.*  
   Berlin: Springer Berlin Heidelberg, 2010, s 613-630. *ISBN: 978-3-642-14623-7*
7. Kalaiselvi K., Kumar A.: *Enhanced AES cryptosystem by using genetic algorithm and neural network in S‑box*.  
   Banglore: 2016 IEEE International Conference on Current Trends in Advanced Computing (ICCTAC), 2006, s. 1-6. doi: 10.1109/ICCTAC.2016.7567340
8. Liu N., et al.: *Cryptographic performance for Rijndael and RC6 block ciphers*.   
   Xiamen: 2017 11th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*,* 2017, s. 36-39. doi: 10.1109/ICASID.2017.8285739
9. Mohamed A. B., Zaibi G., Kachouri A.: *Implementation of RC5 and RC6 block ciphers on digital images.*Sousse: Eighth International Multi-Conference on Systems, Signals & Devices*,* 2011,   
   s. 1-6.  
   doi: 10.1109/SSD.2011.5767447
10. Nechvatal J., et al. *Report on the Development of The Advanced Encryption Standard (AES).* W: Journal of Research of the National Institute of Standards and Technology, Tom 103, Numer 3, s. 511-577, 2001. 106. 10.6028/jres.106.023.
11. Noura M. et al.: *S-DES: An efficient & secure DES variant.*   
    Jounieh: 2018 IEEE Middle East and North Africa Communications Conference (MENACOMM)*,* 2018, s. 1-6. doi: 10.1109/MENACOMM.2018.8371019
12. Rivest R. L. *The RC5 Encryption Algorithm.*  
    Cambridge: MIT Laboratory for Computer Science, 1997
13. Rivest R. L. et. All. *The RC6 Block Cipher.*  
    Cambridge: MIT Laboratory for Computer Science, 1998
14. Sanchez-Avila C., Sanchez-Reillol R.: *The Rijndael block cipher (AES proposal): a comparison with DES.*Londyn: Proceedings IEEE 35th Annual 2001 International Carnahan Conference on Security Technology (Cat. No.01CH37186), 2001, s. 229-234. doi: 10.1109/CCST.2001.962837
15. Schneier B.: *Kryptografia dla praktyków: protokoły, algorytmy i programy źródłowe w języku C*. Warszawa: Wydawnictwa Naukowo-Techniczne, 2002 [ISBN 83-204-2678-2](https://pl.wikipedia.org/wiki/Specjalna:Ksi%C4%85%C5%BCki/8320426782).
16. Schneiner B., Whiting D. *A Performance Comparison of the Five AES Finalists.* W: <https://www.schneier.com/academic/paperfiles/paper-aes-comparison.pdf> [dostęp 20.06.19]
17. Stinson D. R.: *Kryptografia. W teorii i praktyce.*Warszawa: Wydawnictwo Naukowo-Techniczne, 2005. ISBN 83-204-2982-X
18. Trenholme S. *AES' Galois field*. W: <https://www.samiam.org/galois.html> [dostęp 01.04.19]
19. Vu A., et al., *A homogeneous parallel brute force cracking algorithm on the GPU*.   
    Seoul: ICTC 2011*,* 2011, s. 561-564. doi: 10.1109/ICTC.2011.6082661
20. Wang D., Sun S. *Replacement and Structure of S-Boxes in Rijndael*.   
    Hubei: 2008 International Conference on Computer Science and Software Engineering*,* 2008, s. 782-784. doi: 10.1109/CSSE.2008.296