Szkoła Główna Gospodarstwa Wiejskiego
w Warszawie
Instytut Informatyki Technicznej

Mgr Sylwia Stachowiak

SAT-kryptoanaliza wybranych algorytmów kryptografii symetrycznej

SAT-cryptanalysis of selected symmetric cryptography algorithms

Rozprawa doktorska
Doctoral thesis

Promotor:
dr hab. Mirosław Kurkowski, prof. ucz.
Uniwersytet Kardynała St. Wyszyńskiego
Wyższa Szkoła Policji w Szczytnie
Promotor pomocniczy:
dr hab. Konrad Furmańczyk, prof. ucz.
Szkoła Główna Gospodarstwa Wiejskiego

Warszawa, 2022

Oświadczenie promotora pracy

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia jej w postępowaniu o nadanie stopnia naukowego.

Data ..... Podpis promotora pracy .....

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca doktorska została napisana przez mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem stopnia naukowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data ..... Podpis autora pracy .....

Streszczenie

W dzisiejszych sieciach i systemach komputerowych, z oczywistych względów, wymaga się zapewnienia odpowiedniej ochrony przesyłanych lub gromadzonych danych. Do celów tych wykorzystuje się metody i algorytmy kryptograficzne, w tym szyfry symetryczne i asymetryczne.

W rozprawie zawarto wyniki badań dotyczące SAT-kryptoanalizy wybranych szyfrów symetrycznych. Metoda polega na translacji problemu bezpieczeństwa algorytmu kryptograficznego do problemu SAT, z wykorzystaniem bezpośredniego kodowania do formuły boolowskiej. W ramach prowadzonych prac opracowano i opisano formuły kodujące dla trzech różnych szyfrów symetrycznych: Salsa20, AES oraz wybranych modyfikacji DES. Ponadto przeprowadzono serię badań eksperymentalnych łamania brutalnego szyfrów metodą SAT-kryptoanalizy z wybranym tekstem jawnym. Otrzymane wyniki pozwoliły wyznaczyć granice możliwości przeprowadzenia na ww. szyfry ataku brutalnego z wykorzystaniem technik SAT.

Słowa kluczowe: DES, Salsa20, AES, SAT-kryptoanaliza, problem spełnialności formuł logicznych.

Abstract

In today's computer networks and systems, for obvious reasons, it is required to ensure adequate protection of transmitted or collected data. For these purposes, cryptographic methods and algorithms, including symmetric and asymmetric ciphers, are used.

The dissertation contains the results of research on SAT-cryptanalysis of selected symmetric ciphers. The method translates the security problem of a cryptographic algorithm into one of the instances of the SAT problem, using direct encoding into a propositional boolean formula. As part of the work, the encoding formulas for three different symmetric ciphers: Salsa20, AES, and selected DES modifications were developed and described. Moreover, a series of experimental results of ciphers' brutal breaking using the SAT-cryptanalysis method with selected plaintext were carried out. The obtained results allowed us to identify the limits of the possibility of carrying out the above-mentioned brutal attack ciphers using SAT techniques.

Keywords: DES, Salsa20, AES, SAT-cryptoanalysis, Boolean satisfiability problem.

Dziękuję wszystkim życzliwym ludziom, których spotkałam na swojej życiowej drodze...

Spis treści

Rozdział 1.

Wprowadzenie

W tym rozdziale przedstawiona zostanie pokrótce problematyka bezpieczeństwa danych oraz metody jego zapewniania w sieciach i systemach komputerowych oraz zasygnalizowane będą wybrane, stosowane obecnie rozwiązania. Na tym tle w podrozdziale 1.1. zostanie sformułowany cel naukowy prowadzonych i opisanych w niniejszej rozprawie badań, a w podrozdziale 1.2. zostaną przedstawione główne wyniki badawcze uzyskane przez autorkę. W ostatnim podrozdziale zostanie zaprezentowana organizacja tej pracy.

Nie bez powodu mówi się, że najcenniejszym towarem w dzisiejszym świecie jest informacja. Można śmiało stwierdzić, że w dobie społeczeństwa informacyjnego, na bilans zysków i strat we wszystkich dziedzinach życia ogromny wpływ ma posiadanie odpowiednich informacji lub ich brak. W mediach nieustannie można przeczytać o kolejnych sukcesach lub porażkach ludzi, firm, czy nawet państw, których częstym powodem jest właśnie fakt posiadania lub braku wrażliwych informacji. Obecnie w skomputeryzowanym świecie informacje wymieniane są obecnie szybko i w bardzo dużej ilości. Kilkadziesiąt lat temu było to nie tylko technicznie niewykonalne, ale można powiedzieć, że nawet niewyobrażalne.

W otaczającej nas cyfrowej, naszpikowanej informacją, rzeczywistości jednymi z najważniejszych problemów są jest nie tylko bezpieczeństwo informacji, ale i jej wiarygodność. Przy czym nie każda informacja może być jawna, dostępna dla wielu lub wszystkich. Można tutaj myśleć o indywidualnych prawach posiadaczy informacji do zachowania jej dla określonych grup odbiorców. Może to również być ściśle związane z szerzej pojętym bezpieczeństwem, począwszy od bezpieczeństwa gospodarstwa domowego aż po nawet bezpieczeństwo narodowe. W tym drugim przypadku chodzi o odpowiednią weryfikację samej informacji, tożsamości jej nadawcy, a także o to czy podczas przesyłania nie została ona zmodyfikowana.

Najczęstszym kanałem wymiany informacji jest obecnie sieć komputerowa. W Europie, w dzisiejszych czasach, trudno już prawie sobie wyobrazić gospodarstwo domowe bez komputera czy smartfona podłączonego do Internetu. Młodzi ludzie pozbawieni dostępu do sieci często czują się zagubieni. Trudno jednak byłoby zapewnić sprawne i odpowiednie przesyłanie informacji bez narzędzi mających w różny sposób zabezpieczyć komunikację odbywającą się za pomocą światłowodu czy łącza bezprzewodowego.

Kryptologia dostarcza odpowiednie algorytmy mające zapewnić poufność przesyłanych lub zgromadzonych danych. Jest to stosunkowo młoda dziedzina nauki, którą z metodologicznego punktu widzenia można umieścić na pograniczu matematyki i informatyki.

Kryptologia dzieli się na kryptografię i kryptoanalizę. Pierwsza z nich zajmuje się tworzeniem algorytmów i metod mających spełniać cele bezpieczeństwa w szeroko pojętej ochronie danych. Jej głównym zadaniem jest opracowywanie metod szyfrowania informacji, aby między innymi zapewnić poufność przesyłanych w postaci szyfrogramów danych. Ponadto, jak pokazują pochodzące z końca XX wieku rozwiązania, systemy kryptograficzne umożliwiają wykorzystanie kryptografii na przykład do potwierdzania autentyczności informacji jawnych (podpisy cyfrowe). Dużym postępem w tej dziedzinie było opracowanie tzw. kryptografii asymetrycznej (zwanej również kryptografią z kluczem publicznym).

Drugą gałęzią kryptologii jest kryptoanaliza. Jest to nauka zajmująca się analizą rozwiązań kryptograficznych pod kątem ich szeroko rozumianego bezpieczeństwa. Mowa tutaj o tzw. mocy kryptograficznej algorytmów określającej stopień zaufania do zapewnienia poufności przetwarzanych danych. Algorytmy kryptograficzne można analizować stosując wiele różnych metod. Będzie to pokrótce opisane w kolejnych częściach pracy.

1.1. Cel badań i teza rozprawy

Niniejsza rozprawa poświęcona jest właśnie wybranym, specyficznym metodom kryptoanalizy służącym do badania szyfrów symetrycznych. W rozważaniach zawartych w rozprawie zaprezentowane są wyniki analizy wybranych szyfrów symetrycznych dokonywanej metodą translacji problemu bezpieczeństwa szyfru do problemu SAT (SATisfiability Problem [17]) - spełnialności odpowiedniej formuły zapisanej w języku klasycznej logiki zdaniowej[1]. Zastosowane będzie do tego celu bezpośrednie kodowanie algorytmu kryptograficznego (szyfru) do

[1]W tej rozprawie ograniczamy się do stosowania formuł klasycznej logiki zdaniowej z przeliczalnym zbiorem zmiennych zdaniowych oraz klasycznymi spójnikami logicznymi: negacją, koniunkcją, alternatywą, implikacją, równoważnością oraz operacją XOR i stałymi *false i true*.

formuły boolowskiej tak, aby problem kryptoanalizy szyfru metodą ataku z tekstem jawnym był równoważny problemowi spełnialności formuły kodującej [55]. Wyniki eksperymentalne otrzymywane będą przy pomocy wyspecjalizowanych narzędzi rozwiązujących problem SAT dla dużych, czasem mających wiele tysięcy zmiennych formuł zdaniowych. Narzędzia te nazywane są SAT-solverami [16, 18, 19, 90, 91].

Prezentowane w rozprawie wyniki badawcze nie leżą co prawda w głównych i najpopularniejszych nurtach kryptoanalizy, stanowią jednak uzupełnienie badań prowadzonych w tym zakresie na całym świecie, pokazując tym samym szersze tło kryptoanalizy, jako dyscypliny naukowej. Tym samym uzupełniają szerszy obraz kryptoanalizy jako całości.

Teza stawiana w rozprawie brzmi: *Zastosowanie bezpośredniego kodowania boolowskiego oraz SAT-solverów jest efektywną metodą do badania własności szyfrów symetrycznych i ich modyfikacji, w tym do wyznaczania granicy możliwości przeprowadzenia na szyfr ataku brutalnego metodą SAT-kryptoanalizy z uwzględnieniem różnych parametrów i/lub modyfikacji szyfru.*

Cele szczegółowe rozprawy są następujące:

1. Opracowanie i wdrożenie dla wybranych, charakterystycznych szyfrów symetrycznych nowych metod ich bezpośredniego kodowania do formuł boolowskich.

2. Przeprowadzenie serii badań eksperymentalnych łamania brutalnego szyfrów, ich fragmentów i/lub modyfikacji metodą SAT-kryptoanalizy z tekstem jawnym i szyfrogramem.

3. Przeprowadzenie serii badań eksperymentalnych dla różnych wariantów S-boxów stosowanych w szyfrach symetrycznych i ich różnych metod kodowania.

4. Wyznaczenie dla badanych algorytmów szyfrujących i ich modyfikacji granicy możliwości przeprowadzenia na szyfr ataku brutalnego metodą SAT ze względu na różne parametry szyfru.

Wszystkie badania eksperymentalne przeprowadzone zostały przy zastosowaniu różnych, wiodących SAT-solverów [16, 18, 19, 90, 91].

1.2. Główne wyniki rozprawy

Zgodnie z sygnalizowanymi wyżej celami badawczymi udało się przeprowadzić wiele eksperymentów na powstałych w ramach prac kodowaniach rozważanych szyfrów i/lub ich modyfikacji.

Po zakodowaniu badanych szyfrów do formuł boolowskich metodą kodowania bezpośredniego we wszystkich przypadkach potwierdzono poprawność wykonanego kodowania przeprowadzając proces szyfrowania za pomocą SAT solvera dla wartości zawartych w odpowiednich normach.

Analizując otrzymane wyniki należy zauważyć, że nie można jednoznacznie określić, który z solverów SAT jest najefektywniejszy. Praca solverów zależy od wielu czynników. Pracują one deterministyczne lub niedeterministyczne. Szczegółowe wyniki zależą od losowanych wektorów bitów tekstu jawnego i klucza, choć w tym przypadku nie wykracza to nigdy poza ramy odpowiedniego rzędu. Można stwierdzić, że udało się określić granicę obliczalności SAT-kryptoanalizy w rozpatrywanych przypadkach.

Wykonane eksperymenty pokazały, że algorytm DES zakodowany do formuły boolowskiej i poddany atakowi z wybranym tekstem jawnym przy użyciu narzędzi, jakimi są SAT-solvery, może zostać złamany w rozsądnym czasie w wersji zredukowanej do pięciu rund z dodanymi 6 bitami klucza. Oczywiście biorąc pod uwagę fakt, że DES od dawna jest łamany brutalnie w rozsądnym czasie przy pomocy dedykowanych architektur lub szybkich maszyn nie jest to wynik imponujący. Jednak kodowanie szyfru DES jest dobrym przykładem w jaki sposób przebiega sam proces kodowania szyfru do formuły boolowskiej.

Z przeprowadzonych testów wynika, że szyfr strumieniowy Salsa20 ze 128-bitowym kluczem zakodowany do formuły boolowskiej i poddany atakowi z wybranym tekstem jawnym przy użyciu narzędzi, jakimi są SAT-solvery, może zostać złamany w rozsądnym czasie w wersji zredukowanej do dwóch rund. Dodatkowo przy użyciu wybranych narzędzi SAT możliwe jest obliczenie brakujących bitów klucza w zredukowanym do 4 rund wariancie algorytmu Salsa20 (Salsa20/4) z dodanymi 68 początkowymi bitami klucza w czasie krótszym niż 24 godziny.

Wykonane eksperymenty pokazały, że algorytm AES, w wariancie wykorzystującym 128-bitowy klucz, zakodowany do formuły boolowskiej i poddany atakowi z wybranym tekstem jawnym przy użyciu narzędzi, jakimi są SAT-solvery, może zostać złamany w rozsądnym czasie w wersji zredukowanej do jednej rundy z dodanymi co najmniej 16 bitami klucza.

Część opisanych w rozprawie wyników została opublikowana w pracach [125, 126]. Kolejne dwie prace omawiające niepublikowane dotąd wyniki są w przygo-

towaniu [127, 128]. W eksperymentach wykorzystano wiele solverów SAT, a wyniki zaprezentowano dla kilku wybranych dających najlepsze rezultaty [16, 18, 19]. Dobrym przykładem jest, zaprezentowany na konkursie SAT Competition w 2017 roku, SAT-solver glu_vc - rozwiązanie z rodziny CDCL (Conflict-Driven, Clause-Learning), które wykorzystuje heurystykę VSIDS, wskazując zmienne zdaniowe preferowane do dalszego rozgałęziania drzewa. Do naszych badań wykorzystano również dobrze znany i szeroko stosowany MiniSAT, a także kilka innych, w tym równoległych rozwiązań: Plingeling i Glucose-Syrup.

Pełny, własny wkład badawczy autorki opisany w niniejszej rozprawie jest następujący:

- powtórzono wyniki bezpośredniej SAT-kryptoanalizy szyfru DES dla wielu, w tym nowych, SAT-solverów,
- zbadano wpływ na stabilność pracy SAT-solverów ze względu ma dobór wartościowania bitów tekstu jawnego i klucza,
- opracowano bezpośrednie kodowanie boolowskie liniowych Sboxów,
- dokonano SAT-kryptoanalizy dla nowego kodowania liniowych Sboxów,
- wykonano obliczenia mające na celu wyznaczenie granicy praktycznej obliczalności SAT-kryptoanalizy dla szyfru DES i jego różnych modyfikacji w obecnych realiach technicznych (software i moce obliczeniowe stosowanych maszyn),
- opracowano bezpośrednie kodowanie boolowskie poszczególnych elementów składowych szyfru Salsa20, jak i całego szyfru dla każdej rundy,
- dokonano SAT-kryptoanalizy dla Salsa20/2, a także dla Salsa20/4, czyli czterech rund Salsa20 z dodanymi początkowymi, a także z dodanymi końcowymi bitami klucza,
- wykonano obliczenia mające na celu wyznaczenie granicy praktycznej obliczalności SAT-kryptoanalizy dla szyfru Salsa20 w obecnych realiach technicznych (software i moce obliczeniowe stosowanych maszyn).
- opracowano bezpośrednie kodowanie boolowskie poszczególnych elementów szyfru AES, jak i całego szyfru,
- opracowano bezpośrednie kodowanie boolowskie dla przekształcenia `MixColumns()` dla dowolnego wielomianu pierwotnego,

- dokonano SAT-kryptoanalizy dla zredukowanego do jednej rundy wariantu AES,

- wykonano obliczenia mające na celu wyznaczenie granicy praktycznej obliczalności SAT-kryptoanalizy dla szyfru AES w obecnych realiach technicznych (software i moce obliczeniowe stosowanych maszyn).

Wszystkie prace programistyczne i obliczeniowe zostały samodzielnie wykonane przez autorkę rozprawy.

Autorka rozprawy opublikowała wraz z M. Kurkowskim i A. Soboniem następujące artykuły naukowe:

- *SAT vs. Substitution Boxes of DES like Ciphers*, 2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 113-118, 2021. [125]

- *SAT-Based Cryptanalysis of Salsa20 Cipher*, in: Choraś M., Choraś R.S., Kurzyński M., Trajdos P., Pejaś J., Hyla T. (eds) Progress in Image Processing, Pattern Recognition and Communication Systems. CORES IP&C ACS 2021. Lecture Notes in Networks and Systems, vol 255. Springer, Cham. [126]

- *Complete SAT based Cryptanalysis of RC5 Cipher*, Journal of Information and Organizational Sciences 44, no. 2 (2020): 365-382, [117]

- *Towards Complete SAT-based Cryptanalysis of RC5 Cipher*, in Proc. of. 2019 IEEE 15th International Scientific Conference on Informatics, 2019, pp. 397-402, IEEE Press. [118]

Kolejny artykuł ww. zespołu jest zgłoszony na konferencję międzynarodową:

- *SAT vs. Advanced Encryption Standard*, in review process in IEEE 16th International Scientific Conference on Informatics. [127]

oraz kolejny jest w przygotowaniu:

- *New boolean encoding of some AES cryptosystem mathematical aspects.* [128]

1.3. Struktura rozprawy

Rozprawa została podzielona na sześć rozdziałów. Rozdział pierwszy zawiera wprowadzenie do tematyki rozprawy wraz z przedstawieniem problemu badawczego, celu i tezy rozprawy, jej głównych wyników oraz struktury niniejszej pracy.

Rozdział drugi poświęcony jest podstawowym aspektom bezpieczeństwa w systemach informatycznych. Opisano w nim mechanizmy ochrony poufności, w tym algorytmy symetryczne, zarówno te blokowe z ich trybami pracy, jak i strumieniowe oraz pokrótce algorytmy asymetryczne. Ponadto przedstawiono wybrane szyfry klasyczne, problem SAT oraz rozwiązujący ten problem algorytm DPLL. Dokonano też opisu kryptoanalizy szyfrów metodą SAT.

W rozdziale trzecim opisano zasadę działania poprzedniego standardu szyfrowania symetrycznego algorytm DES oraz bazującą na SAT metodę kryptoanalizy tegoż szyfru. Przedstawiono również bezpośrednie kodowanie szyfru do formuł boolowskich, następnie translację formuł do formatu DIMACS, a także uzyskane wyniki eksperymentalne.

Rozdział czwartym poświęcony jest SAT kryptoanalizie szyfru strumieniowego Salsa20. Opisano jego strukturę oraz składowe elementy algorytmu. Przedstawiono kodowanie szyfru metodą bezpośredniego kodowania boolowskiego poszczególnych elementów szyfru Salsa20 do formuł logicznych, a następnie ich konwersję do formatu DIMACS oraz wyniki eksperymentalne kryptoanalizy uzyskane z wykorzystaniem SAT-solverów.

W rozdziale piątym opisano sposób działania szyfru AES, obecnego standardu szyfrowania symetrycznego. Przedstawiono także wykorzystane w jego konstrukcji operacje matematyczne oraz przekształcenia używane przez algorytm szyfrujący. Dokonano kodowania poszczególnych elementów szyfru AES do formuł boolowskich, a następnie ich translacji do formatu DIMACS. Przedstawiono, sprowadzoną do problemu SAT, kryptoanalizę algorytmu AES z wybranym tekstem jawnym oraz jej wyniki uzyskane przy pomocy narzędzi sprawdzających spełnialność formuł SAT-solverów.

Rozdział szósty zawiera podsumowanie całej rozprawy oraz opis planowanych przyszłych badań.

Rozdział 2.

Wstęp do kryptologii oraz problemu SAT i jego zastosowań

W rozdziale tym zostaną przedstawione podstawowe pojęcia stanowiące bazę do rozważań prowadzonych w dalszej części niniejszej rozprawy. W podrozdziale 2.1. zostanie omówiona pokrótce kryptologia jako dziedzina wiedzy. Następnie w podrozdziale 2.2. będą zaprezentowane wybrane szyfry klasyczne. W dwóch kolejnych podrozdziałach zostaną opisane podstawowe rodzaje szyfrów. W podrozdziale 2.4. szerzej zostanie przedstawiona jedna ze składowych części kryptologii jaką jest kryptoanaliza. Następnie w podrozdziałach 2.6. i 2.7. zostanie przedstawiony problem SAT oraz SAT-solvery, czyli narzędzia do jego praktycznego rozwiązywania. Podrozdział 2.8. został poświęcony opisowi kryptoanalizy z tekstem jawnym i szyfrogramem metodą SAT.

2.1. Wstęp do kryptologii

Współczesna kryptologia skupia się zarówno na tworzeniu, jak i doskonaleniu metod i technik ochrony informacji. Obecnie, ochrona informacji obejmuje nie tylko uniemożliwienie osobom niepowołanym dostępu do informacji, ale też inne usługi, takie jak: uwierzytelnianie komunikujących się stron, podpisy cyfrowe, zachowanie i możliwość weryfikacji integralności danych oraz inne zagadnienia związane z bezpieczeństwem danych [83, 102, 113, 129].

Zapewnienie bezpieczeństwa danych w sieciach i systemach komputerowych wymaga spełnienia kilku kryteriów. Są nimi: tajność, wiarygodność, integralność i niezaprzeczalność [113]. *Tajność* (czyli poufność) pozwala na taki przepływ informacji między nadawcą i odbiorcą, aby jej treść pozostała niedostępna dla osób postronnych. *Wiarygodność* zapewnia odbiorcy możliwość sprawdzenia tożsamości nadawcy. *Integralność* umożliwia odbiorcy sprawdzenie, czy wiado-

mość przesyłana niezabezpieczonym kanałem nie została zmodyfikowana przez osobę postronną, co daje nam pewność, że wszystkie modyfikacje wiadomości zostaną wykryte. Natomiast *niezaprzeczalność* uniemożliwia nadawcy wiadomości twierdzenie, że danej wiadomości nie wysłał.

Jak już napisano we Wprowadzeniu, kryptografia to część składowa kryptologii. Zajmuje się ona tworzeniem algorytmów, protokołów i systemów używanych do ochrony informacji przed różnego rodzaju zagrożeniami. Kryptoanaliza jest drugim składnikiem kryptologii i koncentruje się na znalezieniu nowych lub rozwoju już istniejących metod ataków na funkcjonujące lub proponowane systemy. Celem kryptoanalizy stosowanych rozwiązań ochrony informacji jest wykazanie, że nie pozwalają one osiągnąć założonego poziomu bezpieczeństwa lub znalezienie jego granicy [24, 95, 102, 113]. Istnieje kilka głównych metod kryptoanalitycznych wykorzystujących różne typy modelowania matematycznego pracy szyfrów i/lub odpowiednie techniki obliczeniowe.

Nowoczesne **szyfrowanie** jest operacją kryptograficzną, której celem jest zapewnienie tajności danych przesyłanych kanałem transmisyjnym lub gromadzonych w odpowiednim repozytorium. Operacja ta polega na pobraniu informacji i przekształceniu jej za pomocą tzw. klucza kryptograficznego na wersję niezrozumiałą dla niepowołanej strony, nazywaną szyfrogramem. **Deszyfrowanie** jest operacją odwrotną do szyfrowania. Odbiorca będący w posiadaniu odpowiedniego klucza może odczytać wiadomość na podstawie szyfrogramu.

Używając zapisu matematycznego możemy ściślej opisać tę sytuację.

Dowolną piątkę (P, C, K, E, D) nazywamy systemem kryptograficznym jeśli spełnione są następujące warunki [129]:

- P jest skończonym zbiorem możliwych tekstów jawnych.
- C jest skończonym zbiorem możliwych tekstów zaszyfrowanych.
- K – przestrzeń kluczy – jest skończonym zbiorem możliwych kluczy.
- Dla każdego $k \in K$ istnieje reguła szyfrowania $e_k \in E$ i odpowiadająca jej reguła deszyfrowania $d_k \in D$. Reguły $e_k : P \to C$ i $d_k : C \to P$ są funkcjami takimi, że $d_k(e_k(p)) = p$ dla każdego tekstu jawnego p.

Najważniejszą rolę odgrywa tutaj ostatni warunek, który stwierdza, że jeśli do szyfrowania tekstu jawnego p użyto funkcji e_k, a otrzymany kryptogram deszyfrowano za pomocą funkcji d_k, wówczas wynikiem tego procesu będzie pierwotny tekst jawny p [129].

2.2. Szyfry klasyczne

Szyfry mają swoją długą historię. Klasyczne szyfry powstały dużo wcześniej niż komputery, dlatego też działały na znakach, nie na bitach, a szyfrowanie i deszyfrowanie odbywało się najczęściej przy pomocy kartki papieru i pióra. Wśród nich możemy znaleźć szyfr z przesunięciem, szyfr podstawieniowy, czy szyfr afiniczny.

Zasada działania **szyfru z przesunięciem** oparta jest na arytmetyce modularnej. Do jego opisu przyjmijmy, że nadawca i odbiorca wiadomości posługują się alfabetem angielskim, składającym się z 26 liter, choć można by użyć pierścienia $\mathbb{Z}_m$ dla dowolnego modułu m.

Niech $P = C = K = \mathbb{Z}_{26}$. Dla $0 \leqslant k \leqslant 25$ możemy zdefiniować funkcje

$$e_k(p) = p + k \bmod 26$$

oraz

$$d_k(c) = c - k \bmod 26,$$

gdzie $p, c \in \mathbb{Z}_{26}$ [129].

Można zauważyć, że tak przedstawiony szyfr z przesunięciem spełnia warunek $d_k(e_k(p)) = p$, dla każdego $p \in \mathbb{Z}_{26}$. Jest zatem, w sensie podanej wcześniej definicji, systemem kryptograficznym.

Kładąc w powyższych formułach $k = 3$ otrzymujemy tzw. szyfr Cezara, który swoją nazwę zawdzięcza rzymskiemu wodzowi i politykowi Juliuszowi Cezarowi. Jak podaje Swetoniusz w swoim dziele *Dwunastu Cezarów*, Cezar używał tego szyfru pisząc na przykład listy do Cycerona, w których niektóre fragmenty, wymagające tajemnicy, szyfrował [132].

Szyfr podstawieniowy to kolejny dobrze znany system kryptograficzny, który jest stosowany od stuleci. Jego opis znajduje się poniżej.

Niech $P = C = \mathbb{Z}_{26}$. K składa się ze wszystkich permutacji zbioru liczb $\{0, 1, \ldots, 25\}$. Dla każdej permutacji $\pi \in K$ oraz dowolnych $p, c \in Z_{26}$ możemy zdefiniować

$$e_\pi(p) = \pi(p)$$

oraz

$$d_\pi(c) = \pi^{-1}(c),$$

gdzie π^{-1} jest permutacją odwrotną do π.

Szyfr z przesunięciem jest szczególnym przypadkiem szyfru podstawieniowego, w którym korzysta się tylko z 26 spośród 26! możliwych permutacji 26 elementów.

Szyfr afiniczny jest również szczególnym przypadkiem szyfru podstawieniowego. W szyfrze tym funkcje szyfrowania są postaci

$$e(p) = ap + b \bmod 26,$$

gdzie $a, b \in \mathbb{Z}_{26}$. Takie funkcje nazywa się afinicznymi, stąd nazwa szyfr afiniczny. Łatwo zauważyć, że dla $a = 1$ mamy do czynienia z szyfrem z przesunięciem [129].

Niech $P = C = \mathbb{Z}_{26}$ i niech $K = \{(a, b) \in \mathbb{Z}_{26} \times \mathbb{Z}_{26} : NWD(a, 26) = 1)\}$. Dla $k = (a, b) \in K$ definiujemy

$$e_k(p) = ap + b \bmod 26$$

oraz

$$d_k(c) = a^{-1}(c - b) \bmod 26,$$

gdzie $p, c \in \mathbb{Z}_{26}$ [129].

W obu przedstawionych wyżej szyfrach, po ustaleniu klucza każdemu znakowi alfabetu tekstu jawnego odpowiada jednoznacznie wyznaczony znak alfabetu tekstu zaszyfrowanego. Systemy kryptograficzne mające taką własność nazywane są *monoalfabetycznymi*.

Szyfr Vigenére'a taki nie jest. Powstał on w XVI wieku, a jego nazwa pochodzi od nazwiska Francuza Blaise'a Vigenére'a, który, jak się później okazało, nie był autorem akurat tego szyfru. Twórcą był zaś Włoch Giovan Battista Bellaso, który dokładnie opisał ten szyfr w 1553 roku. Można powiedzieć, że szyfr Vigenére'a jest pewnym udoskonaleniem, czy też rozszerzeniem szyfru Cezara. Stał się on popularny i był później używany m.in przez wojsko szwajcarskie podczas I wojny światowej [75].

Działanie szyfru Vigenére'a jest podobne do działania szyfru Cezara przy czym litery nie są przesuwane o 3 miejsca w dół alfabetu, lecz o wartości zdefiniowane przez klucz. W tym przypadku klucz jest ciągiem liter i może być nazywany *słowem kluczowym*. Pozycje kolejnych liter słowa kluczowego w alfabecie wskazują na wartość przesunięcia kolejnych liter tekstu jawnego zgodnie z szyfrem przestawieniowym [10].

Niech n będzie ustaloną dodatnią liczbą całkowitą oraz niech $P = C = K = (\mathbb{Z}_{26})^n$. Wówczas dla klucza $k = (k_1, k_2, \ldots, k_n)$ funkcje szyfrująca i deszyfrująca są postaci:

$$e_k(p_1, p_2, \ldots, p_n) = (p_1 + k_1, p_2 + k_2, \ldots, p_n + k_n)$$

oraz

$$d_k(c_1, c_2, \ldots, c_n) = (y_1 - k_1, y_2 - k_2, \ldots, y_n - k_n),$$

przy czym wszystkie działania są wykonywane w pierścieniu $\mathbb{Z}_{26}$.

W szyfrze Vigenére'a, który ma ustalony klucz długości n, zakładając, że wszystkie znaki słowa kluczowego są różne, znak alfabetu może być przekształcony na jeden z n możliwych znaków. Taki system kryptograficzny nazywamy *polialfabetycznym*.

Rozwój techniki determinował kolejne rozwiązania mające zastosowania w kryptografii. Powstawały metody szyfrowania oparte na urządzeniach mechanicznych [75, 83], czy też mechaniczno-elektrycznych, jak słynna Enigma [65, 75, 83, 100, 102, 113]. Pojawienie się komputerów miało istotny wpływ na zmianę sposobów szyfrowania. Początkowo, do lat 60. XX wieku, komputery miały charakter eksperymentalny, były raczej ciekawostką naukową, wykonywały obliczenia naukowe i dla wojska. Szybko okazało się, że możliwości komputerów są ogromne, a dodatkowo ich ceny spadały i kiedy stały się wielozadaniowym urządzeniem produkowanym seryjnie, rozpoczął się proces ich adoptowania przez przemysł i inne instytucje, w tym finansowe, takie jak np. banki. W tych warunkach powstała konieczność zabezpieczenia danych wrażliwych, a kryptologia zaczęła odgrywać istotną rolę dla bezpieczeństwa transakcji w biznesie.

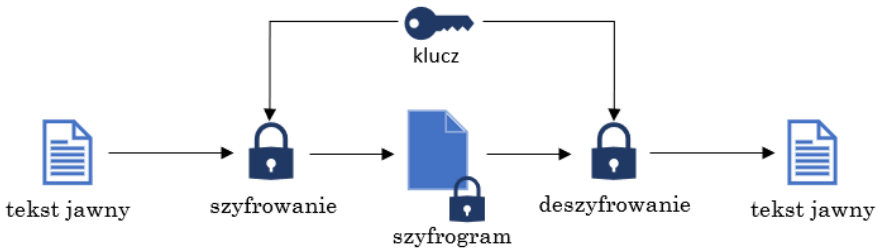
Zastosowanie komputerów spowodowało, że składowanie danych i ich transmisja wymagające szyfrowania były teraz szybsze, wydajniejsze i w tych zastosowaniach można było użyć bardziej skomplikowanych algorytmów. Miało to związek z być może najważniejszą zmianą, jaka nastąpiła dzięki użyciu komputerów, dotyczącą szyfrowania, czyli zmianą typu przetwarzanych danych. Dotychczas szyfrowanie odbywało się na poziomie liter i znaków, natomiast komputery operować zaczęły w systemie dwójkowym. Konsekwencją tego było przejście z szyfrowania liter i znaków na szyfrowanie zer i jedynek. Nietrudno zauważyć, że w takiej sytuacji konieczne stało się ustalenie reguł konwersji znaków na system binarny.

Skorzystano tutaj z tzw. kodu ASCII (ang. American Standard Code for Information Interchange), który poszczególnym znakom alfanumerycznym, znakom pisarskim oraz sterującym przyporządkowuje odpowiedni kod binarny, standardowo z zakresu 0-127. Mając takie narzędzia możliwe stało się zapisanie wiadomości tekstowej w postaci dwójkowej, a następnie jej szyfrowanie. Proces ten ogólnie nie odbiega od procesu szyfrowania z czasów przed powstaniem komputerów. Nadal jest to przetworzenie elementów wiadomości według danego klucza i algorytmu, tak by dla osoby z zewnątrz nie miały one większego sensu. Różnica wynika z faktu, iż elementem, który poddawany jest działaniu algorytmu jest

pojedynczy bit lub ich blok, a nie znak lub ich grupa, jak to miało miejsce do tej pory [77].

2.3. Szyfry symetryczne i asymetryczne

Istnieje wiele rodzajów klasyfikacji szyfrów, ale najpopularniejszym obecnie jest podział na szyfry symetryczne i asymetryczne. I tak w kryptosystemach symetrycznych, nazywanych też systemami z kluczem prywatnym, do szyfrowania i deszyfrowania danej wiadomości wykorzystywany jest ten sam tajny klucz lub klucz szyfrujący i deszyfrujący mają tę własność, że znając jeden z nich można wyznaczyć drugi. Stosowanie tego typu algorytmów szyfrujących wymaga uzgodnienia klucza między nadawcą a odbiorcą przed rozpoczęciem przesyłania wiadomości.



Rysunek 2.1: Schemat działania szyfru symetrycznego

Rysunek 2.1 przedstawia schemat działania szyfru symetrycznego. Nadawca szyfruje przygotowaną wiadomość (tekst jawny) p używając do tego celu algorytmu szyfrowania i klucza k, i wysyła szyfrogram $c = e_k(p)$ do Odbiorcy. Oboje korzystają z tego samego algorytmu szyfrującego, przy użyciu którego dane są szyfrowane i deszyfrowane. Dysponują również tym samym wspólnym, tajnym kluczem $k \in K$ wykorzystywanym przez algorytm szyfrowania i deszyfrowania. Odbiorca otrzymuje szyfrogram c i deszyfruje go wykorzystując wcześniej ustalony, wspólny klucz k, $d_k(c) = p$ i teraz może odczytać wiadomość (tekst jawny) p wysłaną przez nadawcę.

Jedną z wad kryptografii symetrycznej jest fakt, iż przed pierwszym wysłaniem tekstu zaszyfrowanego, użytkownicy muszą przekazać sobie klucz używając do tego bezpiecznego kanału, co tworzy problem dystrybucji klucza [83, 102, 113].

Wśród szyfrów symetrycznych można wyróżnić szyfry blokowe, w których stosowany jest klucz k o stałej długości – niezależnej od długości wiadomości (tekstu jawnego) p oraz szyfry strumieniowe, wykorzystujące klucz k o długości nie mniejszej od długości wiadomości p.

Szyfry blokowe (ang. block ciphers) to klasa algorytmów szyfrów symetrycznych, w których jednostką przetwarzania nie jest pojedynczy bit, a cały blok bitów. Szyfry blokowe przetwarzają iteracyjnie dane, w n-bitowych blokach z wykorzystaniem k-bitowego klucza, dlatego często nazywane są również iteracyjnymi szyframi blokowymi. Pojedyncza iteracja w szyfrze blokowym nazywana jest rundą szyfru. Szyfr blokowy jest podstawowo charakteryzowany przez dwie wartości: rozmiar bloku oraz rozmiar klucza, a jego bezpieczeństwo zależy od obu tych parametrów i oczywiście od konstrukcji algorytmu.

W przypadku zastosowania n-bitowego bloku wiadomość (tekst jawny) p jest traktowana jako konkatenacja ciągów o długości n-bitów, zatem $p = p_1 p_2 \dots p_m$, gdzie p_i jest blokiem o długości n-bitów dla każdego $i = 1, 2 \dots, m$ i każdy kolejny blok p_i jest szyfrowany tym samym kluczem k tworząc ciąg tekstu zaszyfrowanego (szyfrogram) c, taki, że $c = e_k(p_1)e_k(p_2) \dots e_k(p_m)$.

Najczęściej stosowane są bloki 64- lub 128-bitowe. Na przykład bloki DES mają 64 (2^6) bitów, a bloki AES mają 128 (2^7) bitów. Ważne jest, aby bloki nie były zbyt dużych rozmiarów, w celu minimalizacji długości przetwarzanych części tekstu jawnego, a także zużycia pamięci.

Zwykle szyfry blokowe podczas swojej pracy wykonują wiele iteracji (rund) podstawowej wersji zestawu operacji przetwarzania danych. W każdej takiej iteracji stosuje się zmodyfikowany klucz główny szyfru, nazywany często kluczem rundowym.

W szyfrach blokowych do wygenerowania z klucza głównego, będącego przeważnie ciągiem o długości pomiędzy 64 a 256 bitów, zbioru kilkunastu lub kilkudziesięciu kluczy rundowych (ang. round keys, subkeys) stosuje się algorytmy generowania kluczy rundowych (ang. key schedule, key expansion algorithm). Klucze rundowe wykorzystywane są w rundach (iteracjach) właściwego procesu szyfrowania lub deszyfrowania. Zależnie od konstrukcji szyfru, w ramach jednej iteracji wykorzystuje się jeden lub kilka kluczy rundowych.

Generowanie kluczy rundowych jest zazwyczaj jednorazową, czasochłonną operacją odbywającą się przed właściwym szyfrowaniem bloków. Ważne jest by wygenerowane przez algorytm klucze rundowe były niezależne, czyli o właściwościach statystycznych zbliżonych do ciągów losowych.

Biham i Shamir w pracach [24, 25] oraz inni w późniejszych publikacjach, m.in. [79] wykazali, że niezależność kluczy rundowych ma wpływ na proces kryptoanalizy szyfru. Dodatkowo Knudsen i Mathiassen w pracy [82] przedstawili eksperyment pokazujący, że szyfry blokowe z kluczami rundowymi wygenerowanymi przez bardziej skomplikowany algorytm są odporniejsze na kryptoanalizę, a szyfry z kluczami wytworzonymi z wykorzystaniem prostych przekształceń lub posiadającymi defekty statystyczne, są łatwiejsze do złamania.

Ponieważ wiadomości, jakie trzeba zaszyfrować są zwykle znacznie większe od rozmiaru bloku, należy stosować odpowiednie schematy pracy szyfru determinujące metodę szyfrowania kolejnych, składających się na całą wiadomość, bloków. Metody te nazywane są **trybami szyfrowania**. Najbardziej naiwnym podejściem jest podzielenie wiadomości na bloki odpowiednich rozmiarów i zakodowanie osobno każdego z nich prze zastosowaniu tych samych parametrów szyfrowania (kluczy).

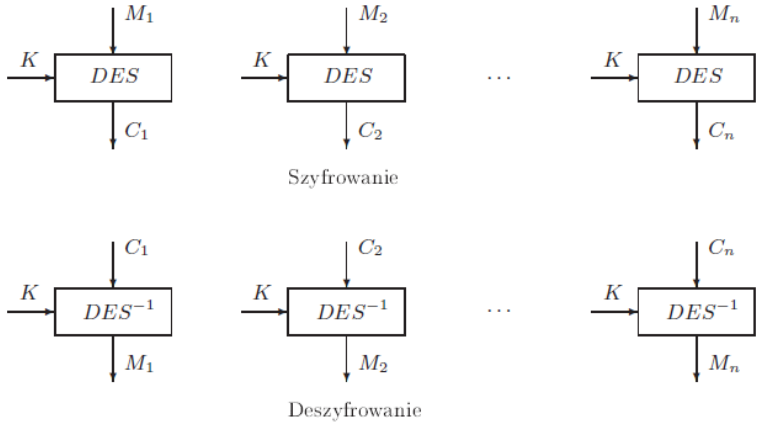
Podstawowymi trybami działania stosowanymi w szyfrach blokowych są:

- tryb ECB (ang. Electronic Codebook mode),
- tryb CBC (ang. Cipher Block Chaining mode),
- tryb CFB (ang. Cipher Feedback mode),
- tryb OFB (ang. Output Feedback mode).

Tryb *elektronicznej książki kodowej* ECB odpowiada zwykłemu szyfrowi blokowemu: przyjmijmy, że $p_1p_2...p_m$ jest ciągiem $n-bitowych$ bloków tekstu jawnego. Wtedy szyfruje się każdy blok p_i, dla $i = 1, 2 \ldots, m$ wykorzystując ten sam klucz k, otrzymując ciąg bloków tekstu zaszyfrowanego $c_1c_2 \ldots c_n$ taki, że $c_i = e_k(p_i)$ dla wszystkich $i = 1, 2 \ldots, m$.

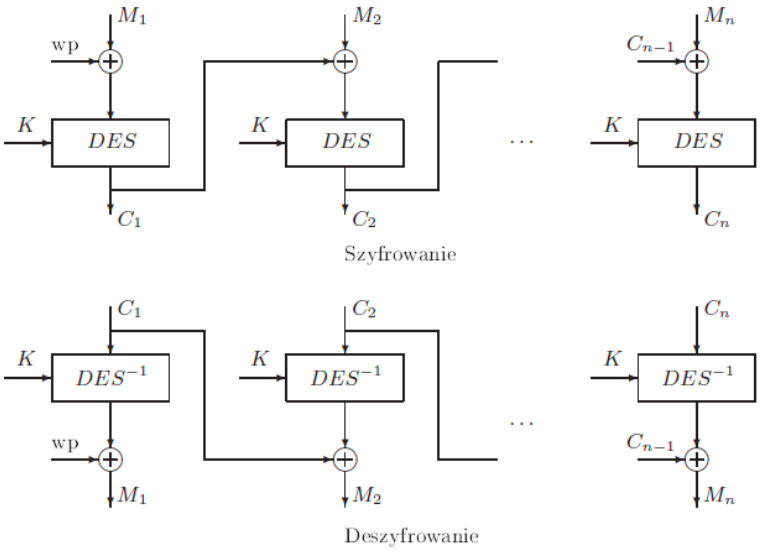
Tryb ten nie jest uważany za bezpieczny, dostarcza bowiem stronie atakującej wiele par postaci tekst jawny – szyfrogram, szyfrowanych tym samym kluczem co ułatwia kryptoanalizę. Inne tryby pracy eliminują to zagrożenie.

Tryb *wiązania bloków szyfrogramu* CBC różni się od trybu ECB tym, że zamiast szyfrowania i-tego bloku tekstu jawnego jako $e_k(p_i)$, szyfruje go zgodnie z regułą $e_k(p_i \oplus c_{i-1})$, dla $i = 1, 2, \ldots, m$, gdzie $\oplus$ to funkcja XOR, a to oznacza, że przed zaszyfrowaniem kolejnego bloku tekstu jawnego wykonuje na nim i na otrzymanym w poprzednim kroku bloku tekstu zaszyfrowanego kluczem k operację XOR. Podczas szyfrowania pierwszego bloku p_1, nie można skorzystać z poprzedniego bloku szyfrogramu (bo on nie istnieje), dlatego CBC bierze wektor początkowy wp (ang. initialization vector), który jest pseudolosowym ciągiem bitów, stąd $c_0 = wp$. Deszyfrowanie w trybie CBC wymaga znajomości



Rysunek 2.2: Stosowanie algorytmu DES w trybie ECB

wartości początkowej wp, dlatego jest ona wysyłana z szyfrogramem w postaci jawnej, a odszyfrowywanie odbywa się za pomocą operacji $d_k(c_i) \oplus c_{i-1} = p_i$.

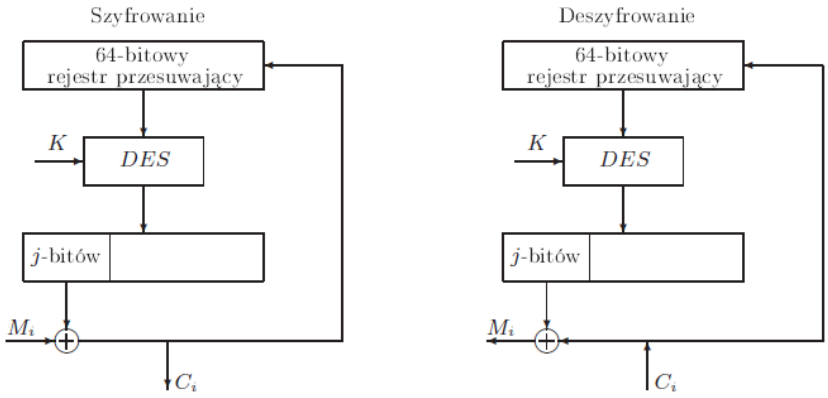


Rysunek 2.3: Stosowanie algorytmu DES w trybie wiązania bloków szyfrowych

Tryb *szyfrowego sprzężenia zwrotnego* CFB generuje klucz strumieniowy w ten sposób, że kolejny element klucza z_i jest tworzony według reguły $z_i = e_k(c_{i-1})$, $i = 1, 2, \ldots, m$, przy czym ponownie $c_0 = wp$. Kolejnym krokiem do uzyskania tekstu zaszyfrowanego jest obliczenie różnicy symetrycznej klucza i tekstu jawnego, a zatem $c_i = p_i \oplus z_i$ dla $i = 1, 2, \ldots, m$.

Tryb *wyjściowego sprzężenia zwrotnego* OFB, podobnie jak tryb CFB, generuje klucz strumieniowy, ale powstaje on w wyniku kolejnych szyfrowań wektora początkowego wp. Stąd $z_0 = wp$ po czym kolejne elementy klucza strumieniowego są obliczane według wzoru $z_i = e_k(z_{i-1})$, $i = 1, 2, \ldots, m$, natomiast ciąg tekstu jawnego jest szyfrowany następująco $c_i = p_i \oplus z_i$ dla $i = 1, 2, \ldots, m$ [129, 10].

Przejdźmy teraz do **szyfrów strumieniowych** (ang. stream ciphers). Istota szyfrów strumieniowych polega na tym, że tworzy się klucz strumieniowy $z = z_1 z_2 \ldots z_m$ (przyjmijmy, że m oznacza długość strumienia), następnie wykorzystuje się go do szyfrowania strumienia tekstu jawnego $p = p_1 p_2 \ldots p_m$ otrzymując w ten sposób strumień tekstu zaszyfrowanego (szyfrogram) $c = e_{z_1}(p_1) e_{z_2}(p_2) \ldots e_{z_m}(p_m)$, a do generowania kolejnych elementów z_i w strumieniu klucza stosuje się funkcję f_i, której wartość jest zależna od klucza k oraz od pierwszych $i-1$ znaków tekstu jawnego p: $z_i = f_i(k, p_1, p_2, \ldots, p_{i-1})$. Wśród szyfrów strumieniowych, są też takie, w których strumień kluczy nie zależy od tekstu jawnego i określa się je mianem *synchronicznych* szyfrów strumieniowych [129].



Rysunek 2.4: Stosowanie algorytmu DES w trybie sprzężenia zwrotnego

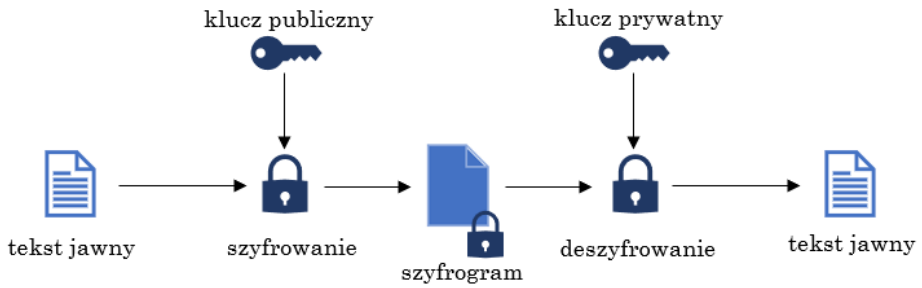
Działanie szyfrów strumieniowych operujących na bitach polega na tworzeniu pseudolosowego strumienia bitów określanego mianem *strumienia klucza* (ang. keystream), a operacja szyfrowania tekstu polega na użyciu funkcji XOR, której argumentami będą tekst jawny i strumień klucza. Aby dokonać operacji deszyfrowania, wystarczy na szyfrogramie i strumieniu klucza wykonać operację XOR. Szyfry strumieniowe jako dane wejściowe przyjmują dwie wartości: klucz i wartość jednorazową (ang. nonce). Klucz pozostaje tajny, a wartość jednorazowa może być jawna, ale powinna być niepowtarzalna dla każdego klucza [10].

Zaletą algorytmów symetrycznych jest najczęściej ich szybkość. Natomiast jedną z wad kryptografii symetrycznej jest, jak już wcześniej wspomniano, tzw. *problem dystrybucji kluczy*, czyli konieczność uzgodnienia klucza pomiędzy nadawcą a odbiorcą, zanim rozpoczną oni wymianę informacji. Konieczne jest zawsze w takich przypadkach opracowanie bezpiecznej metody przekazania tajnego klucza. Chcąc stosować zatem tylko kryptografię symetryczną dochodzimy tu do paradoksu: *aby stworzyć bezpieczny kanał łączności potrzebujemy mieć wcześniej bezpieczny kanał łączności*. W tradycyjnych zastosowaniach nie stanowi to większego problemu, gdyż odpowiednie osoby mogą się ze sobą spotkać i przekazać sobie klucz. Jeżeli jednak komunikacja ma następować za pośrednictwem sieci komputerowych, to istnieje poważna możliwość przechwycenia klucza przez niepowołaną do tego stronę. Ponieważ bezpieczeństwo algorytmu opiera się na bezpieczeństwie klucza, musi on być utrzymany w tajemnicy. Ujawnienie klucza oznaczałoby, że w takim systemie każdy , kto wszedł w posiadanie szyfrogramu, jest w stanie deszyfrować przesyłane wiadomości, tak długo jak, komunikacja będzie wymagała utajnienia, tak długo klucz powinien być utajniony [83, 102, 113].

Inną istotną wadą tego rodzaju kryptografii jest tzw. problem *zaufania do klucza*, a właściwie do kontrahenta. Aby bowiem szyfrować symetrycznie, obie strony komunikacji muszą mieć ten sam klucz. Żadna z nich nie ma pełnej kontroli nad bezpieczeństwem klucza, a jeżeli szyfrować tym samym kluczem ma większa grupa stron, zaufanie do bezpieczeństwa klucza drastycznie spada.

Z wymienionych wyżej powodów opracowano nie tylko bezpieczne metody dystrybucji kluczy symetrycznych, ale również zupełnie inną koncepcję szyfrowania [83, 102, 113].

W kryptosystemach z kluczem publicznym (**systemach asymetrycznych**), których schemat działania przedstawia rysunek 2.5, do szyfrowania i deszyfrowania używa się dwóch różnych kluczy. Każdy użytkownik tego typu algorytmu ma przypisaną unikalną parę kluczy: klucz publiczny k_e i klucz prywatny k_d. Pierwszy z nich jest ogólnodostępny i jeśli wiadomość p zostanie zaszyfrowana przy pomocy klucza k_e, co możemy opisać następującym równaniem $e_{k_e}(p) = c$, wówczas odszyfrowanie otrzymanego w ten sposób szyfrogramu c jest możliwe tylko kluczem prywatnym k_d, odpowiadającym użytemu podczas szyfrowania kluczowi publicznemu, co pisuje równanie $d_{k_d}(c) = p$. Klucz publiczny może być w niektórych kryptosystemach wyznaczony z klucza prywatnego (np. w systemie ElGamal), ale klucza prywatnego nigdy nie można uzyskać znając klucz publiczny, gdyż kryptografia klucza publicznego używa funkcji, dla których łatwo jest obliczyć wartość, ale trudno je odwrócić, czyli znając ich wynik określić wartości wejściowe. Mowa tutaj o tak zwanych funkcjach obliczalnie jednokierunkowych (nieodwracalnych obliczeniowo).



Rysunek 2.5: Schemat działania szyfru asymetrycznego

Dla tych funkcji możemy szybko obliczyć wartość dla zadanego argumentu, natomiast znając wartość funkcji jednokierunkowej, obliczenie jej argumentu jest bardzo trudne, wręcz, niewykonalne w rozsądnym czasie (praktycznie niewykonalne).

Współczesne asymetryczne systemy kryptograficzne oparte są na jednym z jednostronnych w sensie efektywności obliczeniowej przekształceń matematycznych:

- rozkład dużych liczb na czynniki pierwsze (faktoryzacja),
- obliczanie logarytmu dyskretnego

i mogą być stosowane jako samodzielny sposób zabezpieczania danych, dystrybucji kluczy i uwierzytelniania użytkowników [115].

Algorytmy kryptografii asymetrycznej najczęściej wykorzystuje się podczas uwierzytelniania przy pomocy podpisów cyfrowych oraz do zarządzania kluczami. Najpopularniejszymi szyframi asymetrycznymi są: algorytm RSA, wspomniany wcześniej algorytm ElGamala oraz algorytmy wykorzystujące krzywe eliptyczne [83, 102, 113].

Szyfrowanie symetryczne i asymetryczne to dwa główne rodzaje szyfrowania, które są zwykle łączone po to, by tworzyć bezpieczne systemy komunikacji. Zastosowanie połączenia algorytmów kryptografii symetrycznej i asymetrycznej rozwiązuje problem dystrybucji klucza w systemach kryptografii symetrycznej w ten sposób, że szyfrowanie asymetryczne jest zastosowane do przekazania wspólnego klucza, a następnie dane są szyfrowane algorytmami symetrycznymi.

2.4. Wprowadzenie do kryptoanalizy

Od wielu już lat przy projektowaniu systemu kryptograficznego stosuje się tzw. *zasadę Kerckhoffsa*, która mówi, że bezpieczeństwo szyfru powinno opierać się na tajności klucza, a nie na tajności algorytmu szyfrującego. Choć dziś oczekiwanie jawności algorytmu szyfrującego może się wydawać oczywiste, gdy szyfry i protokoły są podawane do publicznej wiadomości i mogą być używane przez każdego, to jednak nie zawsze tak było. W 1883 roku Auguste Kerckhoffs w swoim artykule „La Cryptographie Militaire" [78] odwołując się do wojskowych maszyn szyfrujących, które przeznaczone były dla armii, sformułował sześć wymagań dotyczących wojskowego systemu szyfrowania. Wśród nich tę wspomnianą powyżej.

Nowoczesna kryptografia posługuje się zatem algorytmami podawanymi do publicznej wiadomości. Przed ich zastosowaniem są one badane przez ekspertów. Nie algorytmy, tylko klucze prywatne stanowią tajemnice, których zachowanie determinuje bezpieczeństwo stosowania algorytmu. Bez znajomości tych kluczy nie można szyfrować/odszyfrowywać dokumentów w dobrym kryptosystemie.

Funkcjonujące dzisiaj najważniejsze zasady konstruowania trudnych do złamania szyfrów zostały sformułowane przez Claude'a Elwooda Shannona w 1949 roku. Złamanie szyfru zdefiniował on jako znalezienie metody na odtworzenie klucza i/lub tekstu jawnego na podstawie jego szyfrogramu. Zgodnie z zasadami Shannona algorytmy szyfrowania nie tylko nie mogą dostarczać statystycznych informacji o tekście jawnym, ale też muszą być zaprojektowane w taki sposób, aby ich złożoność wykluczała możliwość złamania szyfru. Istotną częścią analizy algorytmów kryptograficznych jest oszacowanie złożoności obliczeniowej ich samych oraz/lub metod ich złamania. Więcej informacji na ten temat znajdzie się w kolejnych podrozdziałach.

Projekty kryptograficzne mogą być, w ogólności, bezpieczne albo bezwarunkowo, albo warunkowo. Pierwszy rodzaj bezpieczeństwa dotyczy projektów odpornych na ataki przeprowadzane przez osoby mające do dyspozycji nieograniczoną moc obliczeniową. W drugim przypadku bezpieczeństwo projektu jest zależne od trudności odwrócenia operacji kryptograficznej, na której opiera się ów projekt [77].

Projekty kryptograficzne są poddawane atakom, czyli metodom umożliwiającym wyliczenie chronionych elementów projektu „znacząco" szybciej niż zakładał to twórca projektu. Algorytm szyfrowania, który wykorzystuje tajne klucze może być poddany atakowi **siłowemu** (*ang. brute-force attack*). U podstaw tego ataku leży wypróbowanie kolejnych kluczy z całej ich przestrzeni do momentu natrafienia na sensowne rozwiązanie. Powodzenie ataku zależy w głównej mie-

rze od mocy obliczeniowej komputera, ponieważ to od sprzętu, zależy ile kluczy jesteśmy w stanie sprawdzić w określonym czasie. W niektórych przypadkach liczba kluczy dostępnych w danej przestrzeni jest tak duża, że metoda pełnego przeszukiwania nie ma szans powodzenia [77].

Ze względu na dostęp dla kryptoanalityka do odpowiednich danych i\lub parametrów szyfru algorytmy szyfrujące mogą być poddawane następującym typom ataków [10, 77, 113]:

- **Atak z samym szyfrogramem** (*ang. ciphertext-only attack*) - mamy tutaj do czynienia z sytuacją, w której kryptoanalityk zna jedynie szyfrogramy, a jego zadanie polega na odnalezieniu zastosowanego klucza kryptograficznego lub odszyfrowanie jednego lub więcej szyfrogramów. W tym modelu napastnicy są pasywni - nie wykonują zapytań szyfrowania i deszyfrowania.

- **Atak ze znanym tekstem jawnym** (*ang. known-plaintext attack*) - podczas tego ataku przeciwnik pozostaje w posiadaniu zestawu par, tekstów jawnych i odpowiadających im szyfrogramów i chce znaleźć klucz lub odszyfrować nowe szyfrogramy. Ten model jest również modelem ataku pasywnego.

- **Atak z wybranym tekstem jawnym** (*ang. chosen-plaintext attack*) - polega na wybraniu tekstów jawnych i wykonaniu zapytania szyfrowania dla tych tekstów jawnych, a następnie badaniu otrzymanych szyfrogramów. Celem ataku jest uzyskanie klucza lub algorytmu deszyfrującego. Ten typ ataku jest atakiem aktywnym.

- **Atak z wybranym szyfrogramem** (*ang. chosen-ciphertext attack*) - atakujący sam wybiera szyfrogramy i analizuje odpowiadające im teksty jawne, a jego celem jest opracowanie metody automatycznego dekryptażu, aby następnie określić klucz kryptograficzny, mając możliwość wykonania zapytań szyfrowania i odszyfrowywania.

W ciągu kilku ostatnich dekad opracowano wiele różnych rodzajów ataków kryptoanalitycznych. Wśród najważniejszych z nich są: kryptoanaliza liniowa (ang. linear cryptanalysis), opracowana w 1993 roku przez Mitsuru Matsui [96, 99] i kryptoanaliza różnicowa (ang. differential cryptanalysis), opublikowana w 1991 roku, której autorami są Eli Biham i Adi Shamir [24] oraz ich hybrydy lub odmiany. Są to m.in. kryptoanaliza z powiązanymi kluczami (ang. related-key cryptanalysis) [26, 34, 35, 144, 145], atak z poślizgiem (ang. slide attack) [32, 33], atak bumerangowy (ang. boomerang attack) [64, 80, 136], atak prostokątny (ang. rectangle attack) [28, 30, 71, 80] czy kryptoanaliza z niemożliwymi

różnicami (ang. impossible differential cryptanalysis) [27, 31, 37], a także kryptoanaliza różnicowo-liniowa (ang. differential-linear cryptanalysis) [11, 38, 68], która jest połączeniem obu technik, zaprezentowana po raz pierwszy przez Susan Langford i Martina Hellmana w 1994 roku w pracy [92].

Szczegółowa charakterystyka powyższych metod i rodzajów kryptoanalizy wykracza poza ramy niniejszej rozprawy.

2.5. Złożoność obliczeniowa algorytmów

Jak już wspomniano w poprzednim podrozdziale zgodnie z zasadami Shannona algorytmy kryptograficzne muszą być zaprojektowane w taki sposób, aby złożoność problemu ich złamania była zbyt wielka do jego praktycznej realizacji i to z dużym marginesem.

W tym podrozdziale wprowadzone zostaną podstawowe pojęcia w zakresie złożoności obliczeniowej niezbędne do rozumienia problemów opisywanych w dalszych rozdziałach rozprawy. Więcej o złożoności obliczeniowej algorytmów, w tym algorytmów kryptograficznych, można znaleźć m. in. w [102], czy [113] lub [83].

Dobrym przykładem jest, gwarantujący bezpieczeństwo szyfru asymetrycznego RSA, problem rozkładu dużych liczb naturalnych na czynniki pierwsze. Im większa liczba, tym więcej zasobów potrzeba do rozłożenia jej na czynniki. Ogólnie, im większy rozmiar danych wejściowych, tym więcej zasobów (czasu, pamięci, procesorów) potrzeba do ich przetwarzania w celu rozwiązania problemu. Złożoność algorytmu w ogóle, w tym na przykład algorytmu kryptograficznego, jest funkcją rozmiaru danych wejściowych.

Złożoność obliczeniową algorytmu można definiować na różne sposoby, ogólnie jednak pod tym pojęciem rozumie się ilość zasobów, których potrzebuje maszyna (komputer) do wykonywania algorytmu. Zasoby te to najczęściej czas i pamięć, teraz często też liczba procesorów. Ze zrozumiałych względów rozmiary tych zasobów mogą znacznie różnić się od siebie w zależności od kilku parametrów.

Czasowa złożoność obliczeniowa algorytmu to funkcja wskazująca, ile czasu potrzeba na jego wykonanie. Czas mierzymy tu nie w sekundach czy minutach, tylko w liczbie odpowiednio wyselekcjonowanych kroków obliczeniowych, czy czasem operacji na bitach. Ile sekund to trwa, zależy w znacznym stopniu od sprzętu, na którym obliczenia są wykonywane. Niezależnie od sprzętu ani od burzliwego rozwoju komputerów złożoność algorytmu definiuje się jako funkcję wyrażającą ile elementarnych operacji na pojedynczych bitach trzeba wykonać

realizując algorytm, w zależności od rozmiaru danych wejściowych. Złożoność czasowa problemu obliczeniowego jest funkcją rozmiaru danych wyrażająca złożoność najlepszego algorytmu rozwiązania tego problemu.

Jako elementarne przyjmuje się na ogół najprostsze, nierozkładalne już, instrukcje w pewnym języku programowania czy w abstrakcyjnym modelu obliczeń, jakim jest na przykład maszyna Turinga albo jednoprocesorowa maszyna RAM (od ang. Random Access Machine). Nie jest przy tym istotne, jaki to konkretnie język, ponieważ interesuje nas tylko proporcjonalny rząd wielkości liczby operacji, z dokładnością do ewentualnego przemnożenia przez (skończoną) stałą. Dla ustalenia uwagi może to być język Java. Można tez pojedyncze instrukcje (linijki) pseudokodu algorytmu traktować jako operacje elementarne. W tej rozprawie nie odwołujemy się w ogóle do tego ani do abstrakcyjnych maszyn, a jako elementarne traktujemy operacje arytmetyczne na pojedynczych bitach wykonywane w szkolnym dodawaniu i odejmowaniu liczb naturalnych reprezentowanych w układzie dwójkowym.

Pamięciowa złożoność określa jak wiele miejsca obliczenie to zajmuje w pamięci komputera. Złożoność pamięciowa algorytmu (programu) jest miarą ilości wykorzystywanej pamięci. Na przykład, liczba wykorzystywanych komórek taśmy maszyny Turinga.

Przyjmuje się, że algorytmy o efektywności ponadwielomianowej, tzn. wyrażającej się funkcją rosnącą asymptotycznie szybciej niż wielomiany o współczynnikach całkowitych, nie są możliwe do realizacji w praktyce na danych odpowiednio dużych. Nazywa się je niemożliwymi do wykonania w praktyce (ang. infeasible, intractable).

Na przykład, kiedy mówimy, że dla problemu rozkładu liczb naturalnych na czynniki pierwsze nie jest znany algorytm działający w czasie możliwym do realizacji w praktyce, oznacza to na ogół: nie jest znany algorytm działający w czasie wielomianowym.

Złożoność wielomianowa nie jest jednak wystarczającym kryterium praktyczności. Na przykład, dla słynnego problemu sprawdzania, czy liczba naturalna jest pierwsza (tzn. podzielna tylko przez 1 i siebie samą) znany jest już od niemal 20 lat algorytm działający w czasie wielomianowym [2]. Jednak stopień tego wielomianu jest zbyt duży, by dla liczb wielkości interesującej obecnie w praktycznych obliczeniach algorytm ten dawał odpowiedź w czasie praktycznie potrzebnym. Są to bowiem liczby około 1000-bitowe.

Eksploatowany dalej problem SAT jest jednym z problemów z klasy NP (ang. nondeterministic polynomial, niedeterministycznie wielomianowych), czyli problemów, które mogą być rozwiązane w wielomianowym czasie na niedeterministycznej maszynie Turinga. Równoważna definicja mówi, że problem NP jest

to problem, dla którego sprawdzenie podanego z zewnątrz rozwiązania ma złożoność wielomianową. Różnica pomiędzy problemami P i NP polega na tym, że w przypadku P znalezienie rozwiązania ma mieć złożoność wielomianową, podczas gdy dla NP wielomianowe ma być tylko sprawdzenie (zweryfikowanie) rozwiązania. Problem NP-zupełny (NPC, ang. NP-Complete), to problem zupełny w klasie NP, ze względu na tzw. redukcje wielomianowe. Dowolny, inny problem należący do NP może być do niego zredukowany w czasie wielomianowym. Pierwszym historycznie problemem, o którym udowodniono, że jest NP-zupełny jest SAT (patrz [41]). Problem SAT, jako kluczowy dla opisywanych badań zostanie dokładnie opisany w następnym podrozdziale.

Pytanie, czy problemy NP-zupełne można rozwiązywać w czasie wielomianowym, jest obecnie największą zagadką informatyki teoretycznej. Ciągle nie udowodniono tego, że $P \neq NP$. Nie udowodniono także, że $P = NP$. Rozwiązanie tego problemu znalazło się na liście problemów milenijnych. Mimo ufundowania nagrody miliona dolarów za rozwiązanie tak postawionego problemu nikomu do tej pory nie udało się tego zrobić.

2.6. Problem SAT

Jak wspomniano wyżej, problem spełnialności formuł boolowskich nazywany *problemem SAT* jest pierwszym zidentyfikowanym problemem NP-zupełnym [41]. Zagadnienie to zajmuje się sprawdzaniem, czy dana formuła zdaniowa jest spełnialna [85]. Problem ten polega więc na stwierdzeniu, czy istnieje takie wartościowanie zmiennych występujących w zadanej formule logicznej, aby wartość tej formuły wynosiła 1. Jeżeli istnieje przynajmniej jedno takie wartościowanie, to formułę nazywamy spełnialną (SAT), a wartościowanie spełniającym. W przeciwnym wypadku formuła nosi miano niespełnialnej (UNSAT).

Problem spełnialności formuł logicznych jest rozstrzygalny i najprostszą metodą rozwiązującą ten problem jest metoda polegająca na rozważeniu wszystkich możliwych wartościowań zmiennych zdaniowych występujących w badanej formule. Wszystkich, takich podstawień jest 2^n, gdzie n oznacza liczbę zmiennych w formule. Zatem metoda ta charakteryzuje się wykładniczą złożonością obliczeniową [44].

Podstawowym algorytmem służącym do rozwiązywania problemów SAT jest procedura DPLL, która została zaproponowana w 1962 roku, a jej nazwa pochodzi od nazwisk autorów pracy, która ją opisuje: Davisa, Putnama, Logemanna oraz Lovelanda [48]. Metoda DPLL wymaga stosowania koniunkcyjnej postaci normalnej, której struktura zostanie przedstawiona niżej. Schemat działania algorytmu DPLL jest następujący:

- podjęcie decyzji o wyborze zmiennej i jej wartościowaniu,
- dedukcja możliwych wartości pozostałych zmiennych, w przypadkach, gdy to jest możliwe,
- diagnoza, czy zaistniał konflikt, który uniemożliwia spełnienie formuły.

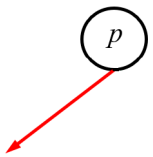
Jeżeli konflikt wystąpił, należy zastosować nawrót. W przypadku braku możliwości nawrotu, oznacza to, że nie istnieje wartościowanie, które spełnia tę formułę i należy zwrócić komunikat UNSAT. Jeżeli formuła jest spełniona należy zwrócić komunikat SAT. W przeciwnym razie algorytm jest kontynuowany poprzez powrót do punktu pierwszego i podjęcia decyzji o wartościowaniu kolejnej zmiennej.

Omówiony schemat można przedstawić za pomocą następującego przykładu. Rozważmy formułę

$$(p \vee q \vee r) \wedge (p \vee q \vee s) \wedge (q \vee \neg r \vee \neg s) \wedge (p \vee \neg q \vee r) \wedge (p \vee \neg q \vee s) \wedge (p \vee \neg r \vee \neg s).$$

Aby dla podanej formuły znaleźć wartościowanie spełniające, należy wykonywać kolejno kroki zgodnie z przedstawionym algorytmem. Dla każdej fazy pracy algorytmu na rysunkach 2.6 - 2.11 przedstawione jest odpowiednie drzewo decyzyjne wraz z formułą.

Pierwszym krokiem jest podjęcie decyzji o wyborze zmiennej i jej wartościowaniu. Zatem weźmy na przykład zmienną p i przypiszmy jej wartość 0.



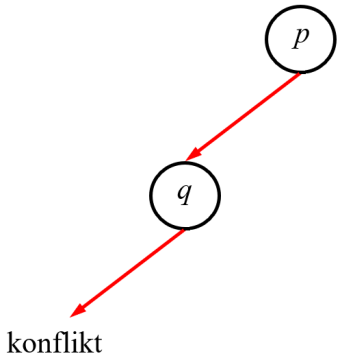
Rysunek 2.6: Wybór wartościowania pierwszej zmiennej

$$(p \vee q \vee r) \wedge (p \vee q \vee s) \wedge (q \vee \neg r \vee \neg s) \wedge (p \vee \neg q \vee r) \wedge (p \vee \neg q \vee s) \wedge (p \vee \neg r \vee \neg s)$$

Wydedukowanie wartości kolejnych zmiennych jest niemożliwe, nie pojawił się też konflikt. Formuła nie jest zatem spełniona, dlatego należy kontynuować nadawanie wartości.

Następnie zmiennej q przypiszmy wartość 0.

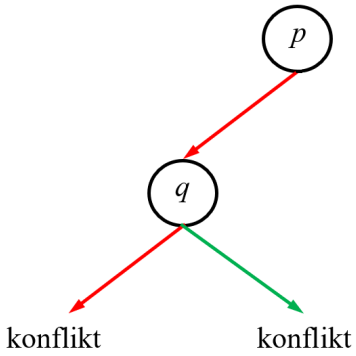
$$(p \vee q \vee r) \wedge (p \vee q \vee s) \wedge (q \vee \neg r \vee \neg s) \wedge (p \vee \neg q \vee r) \wedge (p \vee \neg q \vee s) \wedge (p \vee \neg r \vee \neg s)$$



Rysunek 2.7: Powodujący konflikt wybór wartościowania drugiej zmiennej

Aby formuła była spełniona, każdy czynnik koniunkcji powinien mieć wartość 1. Stąd z klauzuli pierwszej i drugiej wynika, że zmienne r i s przyjmują wartość 1, co prowadzi do tego, że klauzula trzecia i szósta mają wartość 0. Zatem przy takim wartościowaniu formuła nie jest spełniona.

W kolejnym kroku należy zastosować nawrót, czyli zmienić ostatnio podjętą decyzję tzn. zmiennej q przypisać wartość 1.

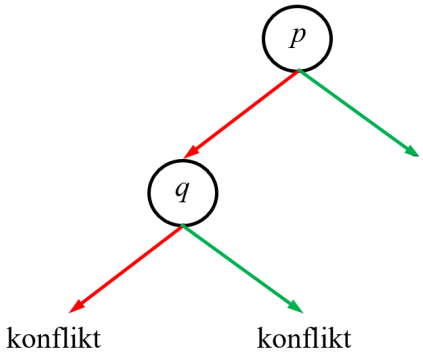


Rysunek 2.8: Kolejny konflikt, konieczność zmiany poprzednich wyborów

$$(p \vee q \vee r) \wedge (p \vee q \vee s) \wedge (q \vee \neg r \vee \neg s) \wedge (p \vee \neg q \vee r) \wedge (p \vee \neg q \vee s) \wedge (p \vee \neg r \vee \neg s)$$

Podobnie jak poprzednio, z klauzuli czwartej i piątej, wynika, że zmienne r i s przyjmują wartość 1, ale skutkuje to tym, że wartość klauzuli szóstej wynosi 0. Wystąpił konflikt. Stąd wnioskujemy, że po przypisaniu zmiennej p wartości 0 formuła ta nie jest spełniona przy obu wartościowaniach zmiennej q. Następstwem tego jest redukcja przeszukiwanego drzewa o połowę. Sytuację tę ilustruje rysunek 2.8.

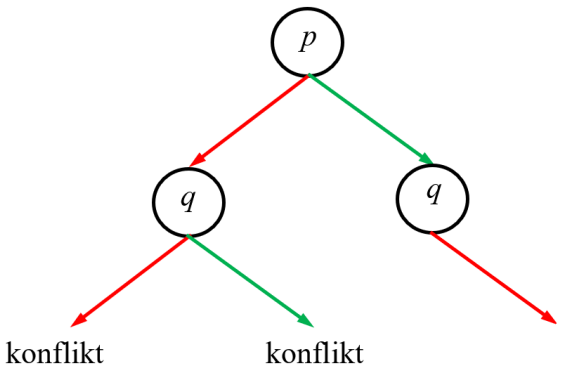
Jeśli założymy, że formuła składała się z ponad 5 tys. zmiennych, podobnie jak w przypadku 16 rund szyfru DES, skala uproszczenia jest jeszcze bardziej widoczna. Efektem jest znaczne skrócenie przeszukiwania drzewa, dlatego SAT solvery mogą znaleźć rozwiązanie spełniające dla bardzo złożonych formuł, gdzie nie jest możliwe sprawdzenie wszystkich wartościowań.



Rysunek 2.9: Kolejny wybór wartościowania

$$(p \vee q \vee r) \wedge (p \vee q \vee s) \wedge (q \vee \neg r \vee \neg s) \wedge (p \vee \neg q \vee r) \wedge (p \vee \neg q \vee s) \wedge (p \vee \neg r \vee \neg s)$$

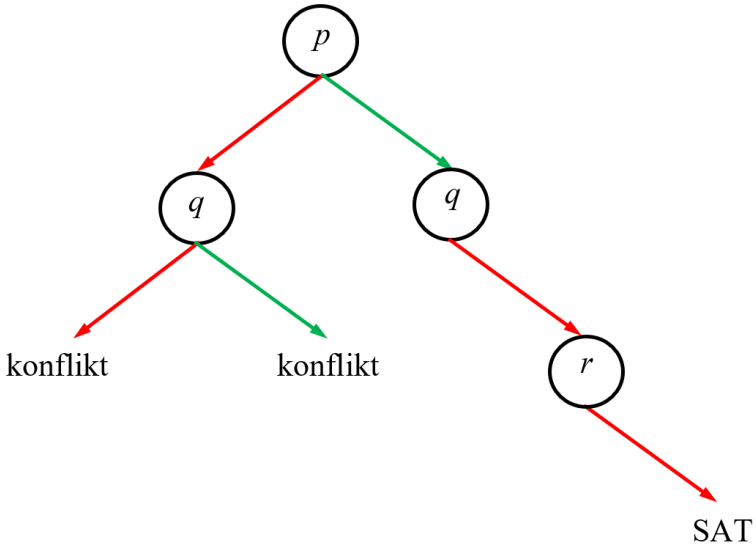
W kolejnym kroku następuje powrót do pierwszej decyzji i tym razem zmiennej p przypisujemy wartość 1. Nie można określić wartościowania pozostałych zmiennych, nie ma też konfliktu, a zatem w dalszym ciągu formuła nie jest spełniona.Tę fazę pracy algorytmu przedstawia rysunek 2.9.
Zatem przechodzimy do wyboru kolejnej zmiennej i przypisania jej wartościowania. Przyjmijmy, że wartość q wynosi 0.



Rysunek 2.10: ...i kolejny...

$$(p \vee q \vee r) \wedge (p \vee q \vee s) \wedge (q \vee \neg r \vee \neg s) \wedge (p \vee \neg q \vee r) \wedge (p \vee \neg q \vee s) \wedge (p \vee \neg r \vee \neg s)$$

Nie można określić wartościowania pozostałych zmiennych, nie ma konfliktu, a zatem w dalszym ciągu formuła nie jest spełniona. Przechodzimy do wyboru kolejnej zmiennej i nadaniu jej wartościowania. Ustalmy zmienną r i przypiszmy jej wartość 0.



Rysunek 2.11: Uzyskanie wyniku SAT

$$(p \vee q \vee r) \wedge (p \vee q \vee s) \wedge (q \vee \neg r \vee \neg s) \wedge (p \vee \neg q \vee r) \wedge (p \vee \neg q \vee s) \wedge (p \vee \neg r \vee \neg s)$$

Skutkuje to spełnieniem zadanej formuły, zwrócony jest zatem komunikat SAT.

Koniunkcyjna postać normalna (Conjunctive Normal Form – CNF), stosowana m.in. w przedstawionym powyżej algorymie DPLL, danej formuły to równoważna jej formuła logiczna mająca postać koniunkcji klauzul, z których każda jest alternatywą literałów (zmiennych zdaniowych lub ich negacji) [66].

$$(p_{11} \vee p_{12} \vee \cdots \vee p_{1k_1}) \wedge (p_{21} \vee p_{22} \vee \cdots \vee p_{2k_2}) \wedge \cdots \wedge (p_{n1} \vee p_{n2} \vee \cdots \vee p_{nk_n})$$

gdzie każde p_{ij} jest literałem.

Rozważmy następujący przykład:

$$((s \oplus r) \Rightarrow (a \wedge b)) \vee q.$$

Korzystając z odpowiednich praw logicznych przekształcimy powyższą formułę do koniunkcyjnej postaci normalnej [66]. Stosując prawo definicji implikacji otrzymujemy:

$$(\neg(s \oplus r) \vee (a \wedge b)) \vee q.$$

Następnie wykorzystując prawo definicji operacji xor i prawa łączności alternatywy mamy:

$$(s \Leftrightarrow r) \vee ((a \wedge b) \vee q).$$

W kolejnych krokach dwukrotnie korzystamy z prawa rozdzielności alternatywy względem koniunkcji.

$$(s \Leftrightarrow r) \vee ((a \vee q) \wedge (b \vee q)),$$

$$[(s \Leftrightarrow r) \vee (a \vee q)] \wedge [(s \Leftrightarrow r) \vee (b \vee q)].$$

Teraz zastosujemy prawo definicji równoważności, następnie ponownie prawo rozdzielności alternatywy względem koniunkcji i otrzymamy:

$$[((s \Rightarrow r) \wedge (r \Rightarrow s)) \vee (a \vee q)] \wedge [((s \Rightarrow r) \wedge (r \Rightarrow s)) \vee (b \vee q)],$$

$$[(s \Rightarrow r) \vee (a \vee q)] \wedge [(r \Rightarrow s) \vee (a \vee q)] \wedge [(s \Rightarrow r) \vee (b \vee q)] \wedge [(r \Rightarrow s) \vee (b \vee q)].$$

Aby doprowadzić powyższą formułę do koniunkcyjnej postaci normalnej korzystamy z prawa definicji implikacji i prawa łączności alternatywy:

$$(\neg s \vee r \vee a \vee q) \wedge (\neg r \vee s \vee a \vee q) \wedge (\neg s \vee r \vee b \vee q) \wedge (\neg r \vee s \vee b \vee q).$$

Stosując odpowiednie prawa logiczne dokonamy teraz konwersji poniższych równoważności do koniunkcyjnej postaci normalnej [66]. Przekształcenie pierwszej z nich jest wykorzystywane w podczas konwersji formuł boolowskich kodujących szyfry DES i Salsa20, natomiast druga równoważność występuje w kodowaniu boolowskim wszystkich prezentowanych w niniejszej rozprawie szyfrów i jej przekształcenie do CNF zostało wykorzystane do otrzymania formuł modelujących działanie szyfrów, w postaci pozwalającej na zastosowanie SAT-solverów.

$$q \Leftrightarrow (p \oplus r)$$

$$(q \Rightarrow (p \oplus r)) \wedge (q \Leftarrow (p \oplus r))$$
$$(\neg q \vee (p \oplus r)) \wedge (\neg(p \oplus r) \vee q)$$
$$(\neg q \vee \neg(p \Leftrightarrow r)) \wedge (\neg\neg(p \Leftrightarrow r) \vee q)$$
$$[\neg q \vee \neg((\neg p \vee r) \wedge (p \vee \neg q))] \wedge [((\neg p \vee r) \wedge (p \vee \neg r)) \vee q]$$
$$[\neg q \vee (\neg(\neg p \vee r) \vee \neg(p \vee \neg q))] \wedge [(\neg p \vee r \vee q) \wedge (p \vee \neg r \vee q)]$$
$$[\neg q \vee ((p \wedge \neg r) \vee (\neg p \wedge q))] \wedge [(\neg p \vee r \vee q) \wedge (p \vee \neg r \vee q)]$$
$$[\neg q \vee ((p \vee (\neg p \wedge r)) \wedge (\neg r \vee (\neg p \wedge q))] \wedge [(\neg p \vee r \vee q) \wedge (p \vee \neg r \vee q)]$$

$$[\neg q \vee ((p \vee \neg p) \wedge (p \vee r)) \wedge ((\neg r \vee \neg p) \wedge (\neg r \vee q))] \wedge [(\neg p \vee r \vee q) \wedge (p \vee \neg r \vee q)]$$
$$[\neg q \vee ((p \vee r) \wedge (\neg r \vee \neg p) \wedge (\neg r \vee q))] \wedge [(\neg p \vee r \vee q) \wedge (p \vee \neg r \vee q)]$$
$$[(\neg q \vee p \vee r) \wedge (\neg q \vee \neg r \vee \neg p) \wedge (\neg q \vee \neg r \vee q)] \wedge [(\neg p \vee r \vee q) \wedge (p \vee \neg r \vee q)]$$
$$(\neg p \vee r \vee q) \wedge (\neg q \vee p \vee r) \wedge (p \vee \neg r \vee q) \wedge (\neg q \vee \neg r \vee \neg p)$$

$$p \Leftrightarrow (q \oplus r) \oplus s$$
$$[p \Rightarrow (q \oplus r) \oplus s] \wedge [(q \oplus r) \oplus s \Rightarrow p]$$
$$\neg p \vee ((q \oplus r) \oplus s)$$
$$\neg p \vee \neg((q \oplus r) \Leftrightarrow s)$$
$$\neg p \vee \neg\,[((q \oplus r) \Rightarrow s) \wedge (s \Rightarrow (q \oplus r))]$$
$$\neg p \vee \neg((q \oplus r) \Rightarrow s) \vee \neg(s \Rightarrow (q \oplus r))$$
$$\neg p \vee \neg(\neg(q \oplus r) \vee s) \vee \neg(\neg s \vee (q \oplus r))$$
$$\neg p \vee ((q \oplus r) \wedge \neg s) \vee (s \wedge \neg(q \oplus r))$$
$$[(\neg p \vee (q \oplus r)) \wedge (\neg p \vee \neg s)] \vee (s \wedge \neg(q \oplus r))$$
$$(\neg p \vee (q \oplus r) \vee s) \wedge (\neg p \vee (q \oplus r) \vee \neg(q \oplus r)) \wedge (\neg p \vee \neg s \vee s) \wedge (\neg p \vee \neg s \vee \neg(q \oplus r))$$
$$(\neg p \vee (q \oplus r) \vee s) \wedge (\neg p \vee \neg s \vee \neg(q \oplus r))$$
$$[\neg p \vee \neg(q \Rightarrow r) \vee \neg(r \Rightarrow q) \vee s] \wedge [\neg p \vee \neg s \vee ((q \Rightarrow r) \wedge (r \Rightarrow q))]$$
$$[\neg p \vee (q \wedge \neg r) \vee (r \wedge \neg q) \vee s] \wedge [\neg p \vee \neg s \vee ((\neg q \vee r) \wedge (\neg r \vee q))]$$
$$[((\neg p \vee s \vee q) \wedge (\neg p \vee s \vee \neg r)) \vee (r \wedge \neg q)] \wedge [(\neg p \vee \neg s \vee \neg q \vee r) \wedge (\neg p \vee \neg s \vee \neg r \vee q)]$$
$$(\neg p \vee s \vee q \vee r) \wedge (\neg p \vee s \vee \neg r \vee \neg q) \wedge (\neg p \vee \neg s \vee \neg q \vee r) \wedge (\neg p \vee \neg s \vee \neg r \vee q)$$

oraz

$$\neg((q \oplus r) \oplus s) \vee p$$
$$((q \oplus r) \Leftrightarrow s)] \vee p$$
$$[((q \oplus r) \Rightarrow s) \wedge (s \Rightarrow (q \oplus r))] \vee p$$
$$[(\neg(q \oplus r) \vee s) \wedge (\neg s \vee (q \oplus r))] \vee p$$
$$[p \vee (\neg(q \oplus r) \vee s)] \wedge [(p \vee \neg s \vee (q \oplus r))]$$
$$[p \vee ((\neg q \vee r) \wedge (q \vee \neg r)) \vee s)] \wedge [(p \vee \neg s \vee ((q \wedge \neg r) \vee (\neg q \wedge r))]$$
$$(p \vee \neg q \vee r \vee s) \wedge (p \vee q \vee \neg r \vee s) \wedge [((p \vee \neg s \vee q) \wedge (p \vee \neg s \vee \neg r)) \vee (\neg q \wedge r))]$$
$$(p \vee \neg q \vee r \vee s) \wedge (p \vee q \vee \neg r \vee s) \wedge (p \vee \neg s \vee q \vee r) \wedge (p \vee \neg s \vee \neg r \vee \neg q).$$

W ten sposób, wykorzystując odpowiednie prawa klasycznej logiki zdaniowej [66], można dokonać przekształcenia formuł kodujących elementy opisywanych

dalej w niniejszej rozprawie szyfrów symetrycznych (DES, Salsa20 i AES) lub ich modyfikacji do koniunkcyjnej postaci normalnej (CNF). Postać ta jest wymagana do zastosowania do dalszych badań kryptoanalitycznych narzędzi rozwiązujących problem SAT, czyli SAT-solverów.

2.7. SAT solvery

Do praktycznego rozwiązywania problemów SAT dla formuł o dużych rozmiarach używa się programów zwanych SAT-solverami (ang. SAT-solvers) [8, 16, 18, 19, 21, 7, 119, 120, 121, 123]. Współcześnie stosowane SAT-solvery mogą czasem skutecznie rozwiązywać, na komputerach klasy PC, problemy zakodowane przez formuły mające setki tysięcy klauzul czy zmiennych zdaniowych. Istnieje wiele problemów, które można przekształcić do postaci problemów SAT, następnie rozwiązać je, a uzyskany model odwzorować na rozwiązanie problemu źródłowego; proces konstruowania tych odwzorowań nosi nazwę odpowiednio kodowania SAT (ang. SAT encoding) albo dekodowania SAT (ang. SAT decoding), gdzie zbiór klauzul odpowiadający danemu problemowi jest deklaratywną reprezentacją tego problemu, co oznacza, że związek między danymi wejściowymi, a wyjściowymi występującymi w problemie jest relacyjny.

Rozwiązywanie problemów przez kodowanie SAT stosuje się m.in. w takich dziedzinach jak: szeregowanie zadań, planowanie działań, konstruowanie i weryfikowanie układów cyfrowych, bioinformatyka i szeroko pojęta ochrona danych [101].

Pierwsze algorytmy dla SAT wprowadzono w latach 60-tych ubiegłego wieku, tworząc programy komputerowe, których zadaniem jest rozwiązanie problemu spełnialności formuł boolowskich, czyli tzw. SAT solverów. SAT solvery sprawdzają, czy wprowadzona formuła jest spełnialna, zwracając komunikat *SATISFIABLE* lub *UNSATISFIABLE*. Ponadto w przypadku, gdy formuła jest spełnialna znajdują tzw. wartościowanie spełniające, a więc wartościowanie zmiennych zdaniowych, które ją spełniają. I chociaż od opublikowania algorytmu DPLL minęło ponad pół wieku, jego główna idea pozostaje podstawą nowoczesnych SAT-solverów.

Oczywiście nie oznacza to, że postęp poczyniony w ciągu ostatnich kilku dekad był nieistotny. Wręcz przeciwnie, problem spełnialności formuł boolowskich został efektywnie wykorzystany i znaleziono jego zastosowanie m.in. w projektowaniu i formalnej weryfikacji układów scalonych (w tym procesorów Intel Core [76]), zautomatyzowanej weryfikacji oprogramowania (używane m.in. przez Microsoft [106]), czy zarządzaniu zależnościami między opcjonalnymi składnikami oprogramowania (tj. wtyczki do środowiska programistycznego Java Eclipse

[15]). W ciągu ostatnich lat nastąpił ogromny postęp w tworzeniu i doskonaleniu SAT solverów. Wiele problemów (np. tych dotyczących weryfikacji sprzętu i oprogramowania), które wydawały się być poza zasięgiem jeszcze kilka lat temu, obecnie można rozwiązywać rutynowo.

Przyczyniło się do tego kilka czynników. Po pierwsze powstanie nowych algorytmów, po drugie lepsze heurystyki, ale największe znaczenie miały dopracowane techniki implementacji. Zainteresowanie tym tematem jest dość duże, dlatego każdego roku, już od dwudziestu lat, organizowany jest konkurs *SAT Competition* [134], którego celem jest utrzymanie siły napędowej w doskonaleniu SAT-solverów, prezentacja nowych rozwiązań i porównywanie SAT solverów. SAT solvery są testowane pod kątem uzyskiwanych czasów potrzebnych do znalezienia wartościowania spełniającego (SAT) i*backslash*lub potwierdzenia, że wprowadzona formuła jest niespełnialna (UNSAT). Narzędzia SAT startują w różnych kategoriach, a w 2021 roku pierwszy raz utworzono ścieżkę kryptograficzną *CRYPTO TRACK*. Warto zauważyć, że projekty biorące udział w konkursie są publicznie dostępne z pełnym kodem źródłowym i należą obecnie do wiodących rozwiązań SAT.

W ciągu ostatnich kilku dekad zastosowano z powodzeniem techniki SAT do badania różnego rodzaju własności licznych systemów informatycznych, w tym systemów współbieżnych. Liczne prace dowodzą skuteczność i efektywność proponowanych rozwiązań [50, 108, 109, 110, 111, 112, 139, 141, 142]. Znakomitym przykładem są badania systemów współbieżnych związanych z bezpieczeństwem sieci i systemów komputerowych. Sztandarowym przykładem są tutaj zabezpieczające protokoły kryptograficzne. Protokoły te stanowią podklasę klasy protokołów komunikacyjnych ogólnego przeznaczenia [107, 85].

Metody SAT zastosowane do badania ww. protokołów dały wiele interesujących i ważnych z punktu widzenia bezpieczeństwa systemów wyników naukowych. Pozwoliły one na opracowanie i wdrożenie do praktycznego zastosowania narzędzi automatycznej weryfikacji własności bezpieczeństwa protokołów. Stosowane tutaj techniki SAT okazały się wysoce efektywne mimo wysokich parametrów badanych systemów i modelujących ich pracę formuł [3, 4, 5, 6, 86, 87, 89, 133]. Ciekawym aspektem jest tutaj włączenie do rozważań czasu i modelowanie oraz weryfikacja również czasowych własności mających wpływ na bezpieczeństwo protokołów [72, 72, 87, 89, 140, 143].

Przechodząc do technicznym aspektów związanych z zastosowaniem technik SAT wspomnieć należy o wymaganym formacie zapisu formuł akceptowanego przez SAT solvery. Jest to tzw. format DIMACS. Format ten jest powszechnie stosowany jako standardowy format formuł logicznych zapisanych w koniunkcyjnej postaci normalnej (CNF) i wykorzystywany do wprowadzania danych

wejściowych dla SAT solverów.

Dane te są zapisane w pliku tekstowym według poniższego schematu:

```
c <Komentarz>
...
c <Komentarz>
p cnf <LiczbaZmiennych> <LiczbaKlauzul>
<ReprezentacjaKlauzuli>
...
<ReprezentacjaKlauzuli>
```

Wiersze rozpoczynające się literą `c` zawierają komentarze i nie są obligatoryjne. Kolejny wiersz pliku w formacie DIMACS zaczyna się od wiersza nagłówka w postaci `p cnf <LiczbaZmiennych> <LiczbaKlauzul>`. Gdzie `<LiczbaZmiennych>` i `<LiczbaKlauzul>` są zastępowane liczbami dziesiętnymi wskazującymi liczbę zmiennych i klauzul w formule. Klauzule są zakodowane jako sekwencja liczb całkowitych oddzielonych spacjami i znakami nowej linii, przy czym literał pozytywny jest reprezentowany przez dodatnią liczbę całkowitą, a do oznaczenia literału negatywnego stosowane są liczby całkowite ujemne. Zwykle każdą klauzulę zapisuje się w osobnym wierszu, dopisując na jej końcu spację i cyfrę 0.

Ze względu na zastosowany algorytm współczesne SAT-solvery można podzielić na dwie główne grupy. Pierwsza z nich reprezentuje podejście stanowiące ewolucję oryginalnego algorytmu DPLL (stosowanego na przykład w SAT-solverze Minisat [123]) i jest określane jako CDCL (ang. Conflict-Driven Clause Learning). Rozszerza klasyczny schemat algorytmu DPLL między innymi o mechanizmy takie jak: uczenie się klauzul, okresowe restarty i różne heurystyki, zarówno deterministyczne, jak i niedeterministyczne. Obecnie algorytmy CDCL są reprezentowane w kilku najnowocześniejszych SAT-solverach, w tym Lingeling [22], Glucose [8].

Obok dalszych ulepszeń w podejmowaniu decyzji po napotkaniu konfliktów i ogólnie sprawniejsze wyszukiwanie, kolejnym ważnym krokiem w kierunku rozwoju SAT-solverów jest przetwarzanie równoległe. Pomimo, że ich rozwój jest jeszcze na wczesnym etapie, to jest już wspierany przez dwa z wyżej wymienionych projektów, a mianowicie Lingeling i Glucose. Mowa tutaj o pLingeling oraz Glucose-Syrup. Oczekuje się, że liczba SAT-solverów stosujących przetwarzanie równoległe będzie wzrastać.

2.8. SAT kryptoanaliza

Wykorzystywanie SAT solverów do analizy algorytmów kryptograficznych stosowane jest już od wielu lat. W 2006 roku Courtois i Bard kontynuowali prace przedstawione w [43] i dokonali ataków algebraicznych na szyfr DES przy użyciu SAT solverów wykorzystując fakt równoważności wielomianowej dwóch problemów NP-trudnych [44]. Rozwiązania SAT są również stosowane do badania własności funkcji skrótu [74, 69, 105], a także do kryptoanalizy klasy szyfrów bazujących na konstrukcji zwanej funkcją gąbkową [51]. Ciekawym podejściem było również zastosowanie technik SAT do problemu faktoryzacji, na którym bazuje słynny kryptosystem RSA [57, 124].

Wspomniana tutaj wcześniej kryptoanaliza szyfru DES z użyciem SAT solverów wykorzystywała techniki tzw. kryptoanalizy algebraicznej. Stosowano również jej modyfikacje, takie jak np. kombinacja technik algebraicznych i różnicowych [62].

W niniejszej pracy zaprezentowane zostanie inne podejście do kryptoanalizy algorytmów szyfrujących wykorzystującej techniki SAT. Algorytm nie będzie opisywany za pomocą równań algebraicznych, które następnie były kodowane do postaci formuł boolowskich, natomiast zastosowane będzie kodowanie bezpośrednie algorytmu szyfrującego, tak jak miało to miejsce w pracach [40, 55, 117]. Zaletą naszej metody jest kodowanie każdego bitu bezpośrednio podczas pracy algorytmu, bez nadmiarowości pod względem rozmiaru formuły kodowania. W tej procedurze używamy jednej zmiennej zdaniowej do opisania każdego bitu podczas procesu szyfrowania. Działanie każdego elementu szyfru w poszczególnych rundach zostało zakodowane do formuły boolowskiej. Liczby całkowite etykietujące uzyskane literały dobrane są w taki sposób, aby otrzymany zbiór klauzul modelował działanie całego szyfru.

Aby móc rozpocząć kryptoanalizę z wybranym tekstem jawnym należy wygenerować ciągi bitów reprezentujących tekst jawny oraz klucz. W tym celu dla każdego bitu klucza i tekstu jawnego tworzymy klauzulę, która składa się jedynie z literału reprezentującego dany bit. Jeżeli wartość bitu wynosi 1, to zmienna zdaniowa jest reprezentowana przez liczbę całkowitą dodatnią, w przeciwnym razie zmienna zdaniowa jest oznaczana przez liczbę całkowitą ujemną. Tak zamodelowane klauzule dołączamy do formuły kodującej działanie szyfru.

Następny etap to poszukiwanie wartościowania spełniającego przygotowaną formułę. Przy użyciu SAT solvera otrzymujemy wartościowanie zmiennych zdaniowych, które tworzą szyfrogram. Biorąc pod uwagę fakt, że praca solvera w tym przypadku modeluje szyfrowanie danych, nie jest zaskoczeniem, że wartościowanie takie uzyskiwane jest błyskawicznie. Ostatecznie, wykorzystując

formułę kodującą algorytm szyfrowania, zbiór klauzul opisujących tekst jawny oraz zbiór klauzul reprezentujących szyfrogram przystępujemy do poszukiwania wartościowania bitów klucza a tym samym do ataku kryptoanalitycznego.

Oczywiście należy rozważyć wpływ doboru bitów klucza i tekstu jawnego na uzyskiwane czasy weryfikacji (stabilność działania solverów ze względu na wybór wartości bitów). Zostanie to skomentowane w następnym rozdziale.

Rozdział 3.

SAT-kryptoanaliza DES

W tym rozdziale zostanie przedstawiona SAT-kryptoanaliza szyfru DES wykonana przy pomocy bezpośredniego kodowania boolowskiego. Zostaną również zaprezentowane wyniki eksperymentalne uzyskane przy pomocy wybranych narzędzi SAT oraz ich analiza. Większość z prezentowanych tutaj treści omawia rozwiązania już wcześniej zaproponowane przez Kurkowskiego, głównie w pracach [40, 55]. W tym rozdziale autorka szczegółowo wprowadza w problematykę kodowania bezpośredniego szyfrów do formuł boolowskich. Powtórzenie badań eksperymentalnych, zwłaszcza dla nowo opracowanych SAT solverów ma już nową wartość badawczą. Dodatkowo autorka przedstawia nowe kodowanie liniowych S-boxów oraz wykonane eksperymenty pokazujące jak różne metody kodowania liniowych S-boxów mają wpływ na efektywność SAT kryptoanalizy.

3.1. Podstawy szyfru DES

Początki powstania szyfru DES sięgają roku 1971, kiedy to Horst Feistel, pracownik IBM w pracy *Block Cipher Cryptographic System* z 1971 zaproponował nowy rodzaj szyfru, tzw. sieć Feistela (FN - Feistel Network), odmianę sieci podstawieniowo-permutacyjnej (substitution–permutation network (SPN)), w której proces szyfrowania i deszyfrowania może być realizowany przy użyciu jednego schematu. Wkrótce Feistel opublikował opartą na idei sieci FN pierwszą wersję szyfru Lucyfer (ang. Lucipher), który był szyfrem blokowym, a podstawową jednostką przetwarzania nie był pojedynczy bit czy bajt, a całe bloki danych. Feistel stworzył kilka wersji tego szyfru, a Lucyfer w wersji oznaczonej jako DTD-1 stał się pierwszym powszechnie stosowanym szyfrem używanym w bankowości elektronicznej [77].

W międzyczasie Narodowe Biuro Standardów USA (obecnie NIST) rozpoczęło tworzenie programu zabezpieczania informacji zdigitalizowanych, w ramach

którego miał powstać algorytm kryptograficzny do szyfrowania danych, dobry zarówno do ich przechowywania, jak i przesyłania. W 1973 roku w Rejestrze Federalnym przedstawiono wymagania, jakie powinien spełniać potencjalny algorytm: miał zapewnić wysoki stopień bezpieczeństwa, być łatwy do zaimplementowania sprzętowego, a także być kompletny i nietrudny do zrozumienia. Ponadto jego algorytm miał być jawny, a bezpieczeństwo miało zależeć od klucza. Żaden spośród zgłoszonych w pierwszej turze algorytmów nie był zadawalający.

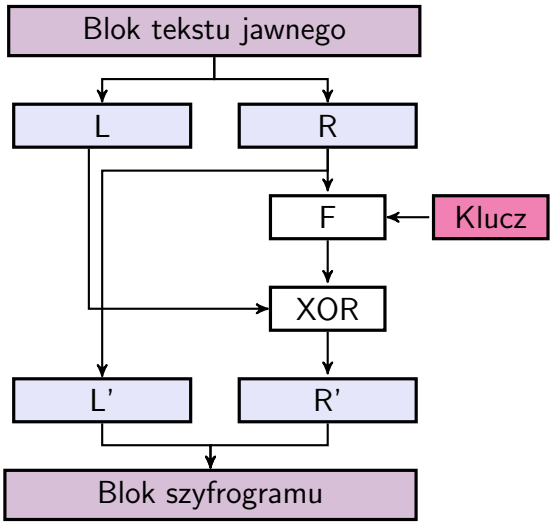
W związku z tym w 1974 NIST ponownie opublikował informacje o zapotrzebowaniu na algorytm kryptograficzny. Wtedy do konkursu swój algorytm zgłosił IBM. Algorytm ten był oparty na szyfrze Lucifer i spełniał podane przez NIST kryteria. Nad rozwojem zgłoszonego szyfru pracowali Roy Adler, Don Coppersmith, Horst Feistel, Edna Grossman, Alan Konheim, Carl Mayer, Bill Notz, Lynn Smith, Walt Tuchman oraz Bryant Tuckerman. Po wprowadzeniu modyfikacji przez amerykańską National Security Agency w 1976 algorytm został zaakceptowany jako standard federalny – można było go używać do szyfrowania nietajnych danych rządowych USA.

Algorytm został nazwany Data Encryption Standard. Publikacja pełnego opisu algorytmu odbyła się w 1977 roku w dokumencie FIPS PUB 46 – Data Encryption Standard. Algorytm kilkukrotnie (1983, 1988 – FIPS-46-1, 1993 – FIPS-46-2 oraz 1999 – FIPS-46-3) zdobywał certyfikat przedłużający jego ważność jako standardu federalnego. Ostatecznie został wycofany w 2001 roku i zastąpiony przez Advanced Encryption Standard, który od tego momentu stał się nowym standardem federalnym. W 1981 roku American National Standards Institute uznało DES za standard sektora prywatnego, jednocześnie zmieniając jego nazwę na Data Encryption Algorithm.

3.2. Sieć Feistela

Sieć Feistela to struktura często wykorzystywana do budowy symetrycznych szyfrów blokowych, a jej działanie ma charakter iteracyjny, gdzie podstawowym przekształceniem jest runda, a szyfrowany blok (w ogólności) może być dowolnej parzystej długości.

W pierwszej rundzie szyfrowany jest blok tekstu jawnego, co obrazuje rysunek 3.1, a w rundzie $n + 1$ w miejsce bloku tekstu jawnego wstawiany jest szyfrogram n - tej rundy. Podczas jednej rundy dany blok danych wejściowych jest dzielony na dwie części równej długości. Prawa połowa tego bloku staje się lewą stroną szyfrogramu. Aby otrzymać prawą stronę szyfrogramu, algorytm łączy za pomocą nieodwracalnej funkcji F prawą stronę bloku zawierającego dane



Rysunek 3.1: Schemat działania sieci Feistela dla pierwszej rundy

wejściowe z kluczem, a następnie wynik tej operacji oraz lewa strona bloku danych wejściowych zostają poddane działaniu funkcji XOR. Warto zauważyć, że funkcja F jest dowolną funkcją, która wejściowy ciąg bitów przekształca na ciąg równoliczny z wejściowym [83].

Na rysunku 3.2 przedstawiony jest schemat szyfrowania i deszyfrowania algorytmem Feistela. Przyjmijmy, że $i = 1, \ldots, n$ oznacza numer rundy, L_i i R_i odpowiednio lewą i prawą stronę bloku wejściowego, a F jest funkcją zależną od klucza szyfrującego K przekształcającą wzajemnie jednoznacznie połówki bloku. Dodatkowo niech K_i oznacza klucz i-tej rundy oraz L_{i+1}, R_{i+1} oznaczają lewą i prawą stronę bloku po i-tej rundzie. Zauważmy też, że blok L_{n+1} i R_{n+1} zawiera lewą i prawą stronę szyfrogramu.

Przekształcenie danych wejściowych poprzez sieć Feistela odbywa się zgodnie z następującymi równaniami:

$$\begin{aligned} L_{i+1} &= R_i, \\ R_{i+1} &= L_i \oplus F(R_i, K_i). \end{aligned}$$

Sieć Feistela używa jednego algorytmu zarówno do szyfrowania, jak i deszyfrowania. Deszyfrowanie polega na zastosowaniu odwrotnych przekształceń Feistela w odwrotnej kolejności niż przy szyfrowaniu. Wówczas mamy:

$$\begin{aligned} R_i &= L_{i+1} \\ L_i &= R_{i+1} \oplus F\left(L_{i+1}, K_i\right). \end{aligned}$$

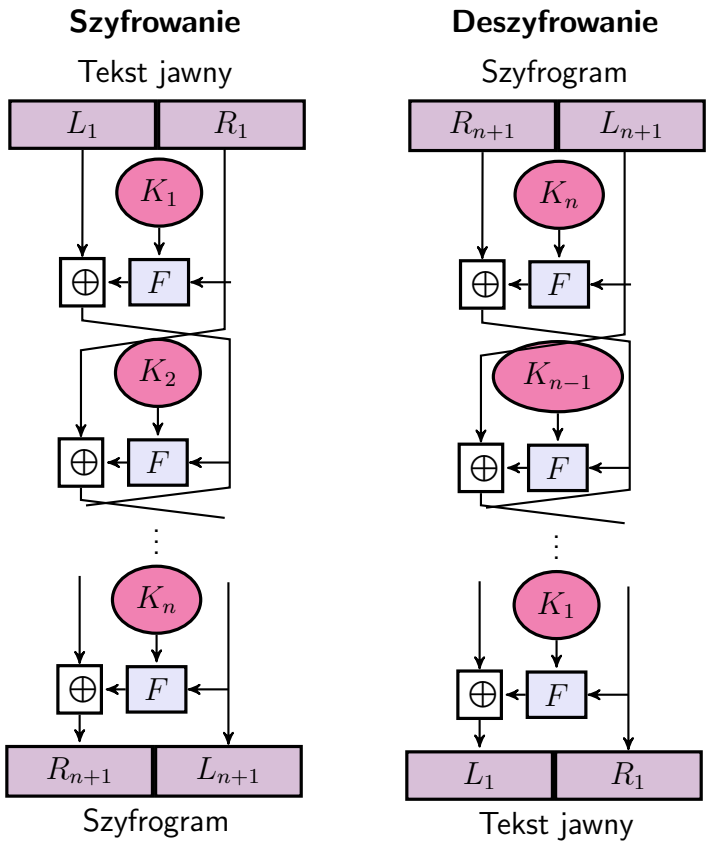
Wykorzystując poniższe własności funkcji XOR:

$$x \oplus x = 0,$$
$$x \oplus 0 = x,$$
$$x \oplus (y \oplus z) = (x \oplus y) \oplus z.$$

otrzymujemy następujące równości na podstawie których można zauważyć, że (L_1, R_1) jest ponownie tekstem jawnym:

$$\begin{aligned} R_{i+1} \oplus F\left(L_{i+1}, K_i\right) &= \left(L_i \oplus F\left(R_i, K_i\right)\right) \oplus F\left(L_i, K_i\right) = \\ &= L_i \oplus \left(F\left(R_i, K_i\right) \oplus F\left(L_i, K_i\right)\right) = L_i \oplus 0 = L_i. \end{aligned}$$

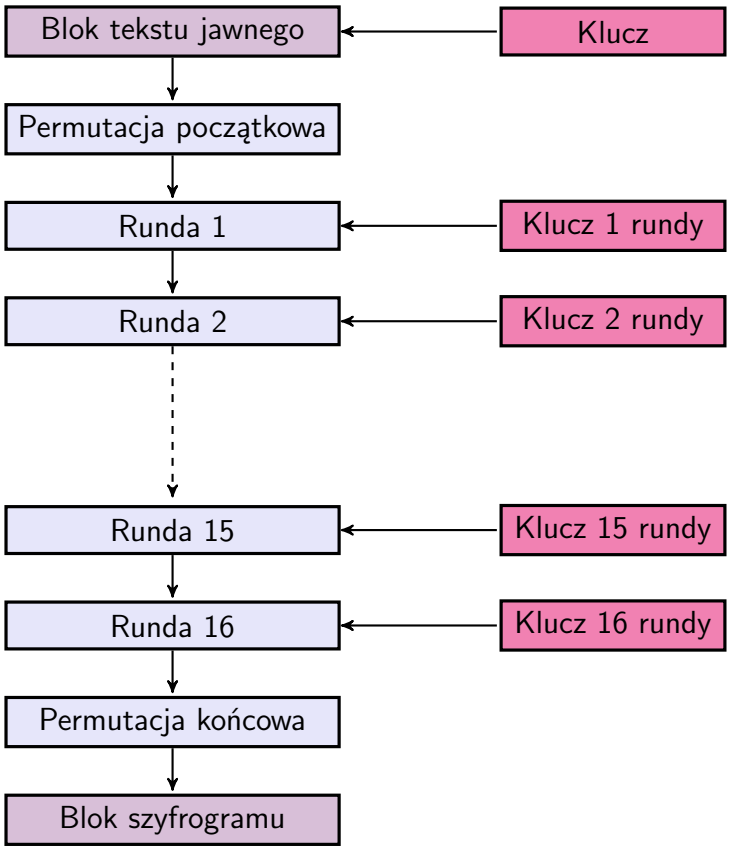
Na rysunku 3.2 przedstawiony jest schemat n-iteracji szyfrowania i deszyfrowania siecią Feistela.



Rysunek 3.2: Schemat szyfrowania i deszyfrowania przy użyciu sieci Feistela

3.3. Struktura szyfru DES

Konstrukcja algorytmu DES opiera się na ogólnej strukturze algorytmu Lucifer. DES przetwarza 64-bitowe bloki danych stosując 64-bitowe klucze w 16 rundach (w praktyce wykorzystywanych jest 56 bitów klucza, co ósmy bit jest pomijany). Algorytm ten może być używany zarówno do szyfrowania, jak i deszyfrowania. W pierwszej rundzie blok danych wejściowych jest blokiem tekstu jawnego, a w kolejnych rundach jest szyfrogramem uzyskanym w rundzie poprzedniej. Na wejściu blok tekstu jawnego jest poddawany działaniu permutacji początkowej, która jako dane wejściowe przyjmuje ciąg 64-bitowy i zwraca 64-bitowy wynik. Następnie sekwencja ta zostaje poddana 16 przekształceniom Feistela, z których każde wykorzystuje inny klucz rundowy. Do otrzymanego w ten sposób ciągu bitów algorytm stosuje permutację końcową. Blok danych wyjściowych jest blokiem szyfrogramu. Ogólny schemat działania algorytmu przedstawia rysunek 3.3.



Rysunek 3.3: Schemat działania algorytmu DES

W dalszej części tego podrozdziału zostaną przedstawione przekształcenia, które są wykorzystywane w szyfrze DES.

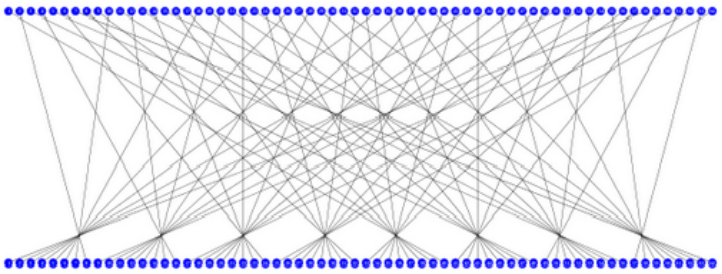
3.3.1. Permutacja początkowa

Permutacja początkowa IP jako dane wejściowe przyjmuje ciąg 64-bitów i zwraca sekwencję 64 bitów. Jeśli przez y oznaczymy blok danych wejściowych i $y = (y_1, y_2, ..., y_{64})$, to permutacja IP przekształci go na blok postaci $IP(y) = (y_{58}, y_{50}, \ldots, y_4, y_{62}, \ldots, y_7)$ zgodnie z poniższą tabelą:

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

Tabela 3.1: Permutacja początkowa IP

Rysunek 3.4 przedstawia schemat przemieszania bitów stosowany przez permutację początkową [137].



Rysunek 3.4: Permutacja bitów podczas IP

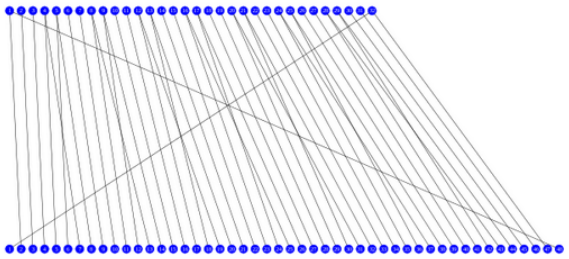
3.3.2. Funkcja rozszerzenia

Funkcja rozszerzenia E jako dane wejściowe przyjmuje blok 32 bitów, zmienia ich uporządkowanie i powtarza niektóre z nich, aby w wyniku swego działania otrzymać blok wyjściowy o długości 48 bitów. Jeśli przez z oznaczymy ciąg 32-bitowy oraz $z = (z_1, z_2, \ldots, z_{32})$, to $E(z)$ jest sekwencją o długości 48 bitów i przekształca blok wejściowy na ciąg postaci $E(z) = (z_{32}, z_1, \ldots, z_9, z_8, \ldots, z_1)$ według poniższej tabeli:

32	1	2	3	4	5	4	5	6	7	8	9
8	9	10	11	12	13	12	13	14	15	16	17
16	17	18	19	20	21	20	21	22	23	24	25
24	25	26	27	28	29	28	29	30	31	32	1

Tabela 3.2: Tabela wyboru bitów funkcji E

Na rysunku 3.5 przedstawia sposób działania funkcji E, której argumentami są kolejne bity danych wejściowych [137].



Rysunek 3.5: Działanie funkcji rozszerzenia E

3.3.3. Funkcja podstawieniowa S-box

Algorytm DES stosuje osiem bloków podstawień, czyli tak zwanych S-boxów $S_1, \ldots, S_8$, które przedstawia zamieszczona poniżej tabela 3.3. Każdy S-box S_j jest macierzą o 4 wierszach i 16 kolumnach, której elementami są liczby ze zbioru $\{0, 1, \ldots, 15\}$. Funkcja podstawieniowa S_j przekształca 6-bitowy ciąg $y = (y_1, y_2, y_3, y_4, y_5, y_6)$ na 4-bitowy ciąg $S_j(y)$ w ten sposób, że liczba całkowita odpowiadająca dwóm krańcowym bitom danych wejściowych y_1y_6 wyznacza numer wiersza macierzy S_j, natomiast liczba odpowiadająca czterem bitom $y_2y_3y_4y_5$ określa, którą kolumnę należy wybrać.

Warto zwrócić uwagę, że zarówno wiersze, jak i kolumny numerowane są od zera. Wartość $S_j(y)$ jest 4-bitową reprezentacją liczby całkowitej znajdującej się w macierzy S_j na przecięciu się danego wiersza i kolumny. Jeśli na przykład $y = (0, 1, 0, 0, 1, 1)$ to S_1 zwróci wartość znajdującą się w wierszu o numerze 1, kolumnie 9, czyli 6 zapisane w systemie binarnym jako 0110. S-boxy istotnie wpływają na bezpieczeństwo algorytmu DES. Można wyróżnić niektóre właściwości projektowe S-boxów [113]:

- Żaden S-box nie jest funkcją liniową ani afiniczną (S-boxy są nieliniowe).

- Funkcja z każdego wiersza S-boxu jest permutacją (S-boxy dają ciągi ze zrównoważoną liczbą zer i jedynek).
- Zmiana pojedynczego bitu danych wejściowych S-boxu zmienia przynajmniej dwa bity wyjściowe (S-boxy mają cechę lawinowości zmian).
- Dla każdego S-boxu S, $S(y)$ oraz $S(y \oplus 001100)$ musi różnić się przynajmniej na dwóch bitach.
- $S(y) \neq S(y \oplus 11ef00)$ dla dowolnie wybranych bitów e i f.
- S-boxy minimalizują różnicę między ilością jedynek i zer na wyjściu z S-boxów, kiedy tylko jakiś pojedynczy bit jest stały.

S-box S_1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S-box S_2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S-box S_3															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S-box S_4															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S-box S_5															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S-box S_6															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S-box S_7															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S-box S_8															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Tabela 3.3: Osiem S-boxów zastosowanych w algorytmie DES

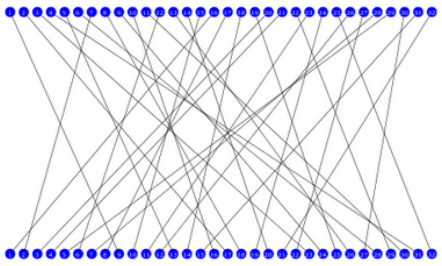
3.3.4. Permutacja P-box

Funkcja permutacji P-box daje 32-bitowe wyjście z 32-bitowego wejścia poprzez permutację bitów bloku wejściowego. Niech teraz y oznacza 32- bitowy blok i $y = (y_1, y_2, \ldots, y_{32})$. Wówczas funkcja P-box przekształci go na blok taki, że P-box$(y) = (y_{16}, y_7, \ldots, y_{10}, y_2, \ldots, y_{25})$ zgodnie z poniższą tabelą:

16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

Tabela 3.4: Permutacja P-box

Wpływ permutacji P-box na bity wejścia prezentuje rysunek 3.6 [137].



Rysunek 3.6: Permutacja bitów poprzez funkcję P-box

3.3.5. Permutacja końcowa

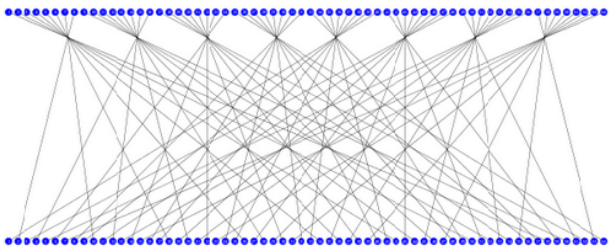
Algorytm DES wywołuje permutację końcową IP^{-1} po wykonaniu 16 iteracji sieci Feistela, przy czym po ostatniej rundzie lewa i prawa połówka nie są zamieniane miejscami, natomiast połączone stają się blokiem danych wejściowych do permutacji końcowej. Zabieg ten jest niezbędny do tego, by algorytm mógł być użyty zarówno do szyfrowania, jak i deszyfrowania.

Niech y oznacza ciąg 64 bitów. Wtedy IP^{-1} jest również ciągiem złożonym z 64 bitów. Załóżmy dodatkowo, że $y = (y_1, y_2, \ldots, y_{64})$, wówczas $IP^{-1}(y) = (y_{40}, y_8, \ldots, y_{31}, y_{38}, \ldots, y_{25})$ zgodnie z tabelą 3.5.

40	8	48	16	56	24	64	32	39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30	37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28	35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26	33	1	41	9	49	17	57	25

Tabela 3.5: Permutacja końcowa IP^{-1}

Na rysunku 3.6 przedstawione jest działanie permutacji końcowej na bity danych wejścia [137].



Rysunek 3.7: Przemieszanie bitów podczas permutacji końcowej IP^{-1}

3.3.6. Generowanie kluczy rundowych

W każdej rundzie szyfru DES algorytm używa innego 48-bitowego klucza wygenerowanego z klucza początkowego. Klucz inicjujący jest blokiem 64-bitowym z 8 bitami parzystości na pozycjach 8, 16, ..., 64, które są określone w ten sposób, aby każdy bajt składał się z nieparzystej liczby jedynek. Podczas generowania zestawu podkluczy bity parzystości są pomijane, a pozostałe 56 bitów jest poddawane działaniu permutacji $PC1$.

Przyjmijmy, że y jest ciągiem 56-bitów i $y = (y_1, \ldots, y_7, y_9, \ldots, y_{15}, y_{17}, \ldots, y_{23}, y_{25}, \ldots, y_{31}, y_{33}, \ldots, y_{39}, y_{41}, \ldots, y_{43}, \ldots, y_{55}, y_{57}, \ldots, y_{63})$.

Wówczas $PC1(y)$ jest ciągiem takim, że $PC1(y) = (y_{57}, y_{49}, \ldots, y_1, y_{58}, \ldots, y_4)$ zgodnie z tabelą 3.6.

57	49	41	33	25	17	9	1
58	50	42	34	26	18	10	2
59	51	43	35	27	19	11	3
60	52	44	36	63	55	47	39
31	23	15	7	62	54	46	38
30	22	14	6	61	53	45	37
29	21	13	5	28	20	12	4

Tabela 3.6: Permutacja początkowa klucza $PC1$

W kolejnym kroku blok wyjściowy permutacji początkowej klucza $PC1$ jest dzielony na dwie połówki, które są używane do utworzenia wszystkich podkluczy. Następnie ciąg bitów wchodzący w skład każdej z nich zostaje poddany cyklicznemu przesunięciu o jedną lub dwie pozycje w lewo, w zależności od numeru rundy zgodnie z tabelą:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Tabela 3.7: Przesunięcie klucza

W ostatnim kroku otrzymana sekwencja 56 bitów zostaje przekształcona przez permutację kompresującą $PC2$. Jeśli y oznacza sekwencję 56 bitów i $y = (y_1, \ldots, y_{56})$, to wynikiem funkcji permutacji $PC2$ jest 48-bitowy blok oraz $PC2(y) = (y_{14}, y_{17}, \ldots, y_{10}, y_{23}, \ldots, y_{32})$ zgodnie z tabelą:

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

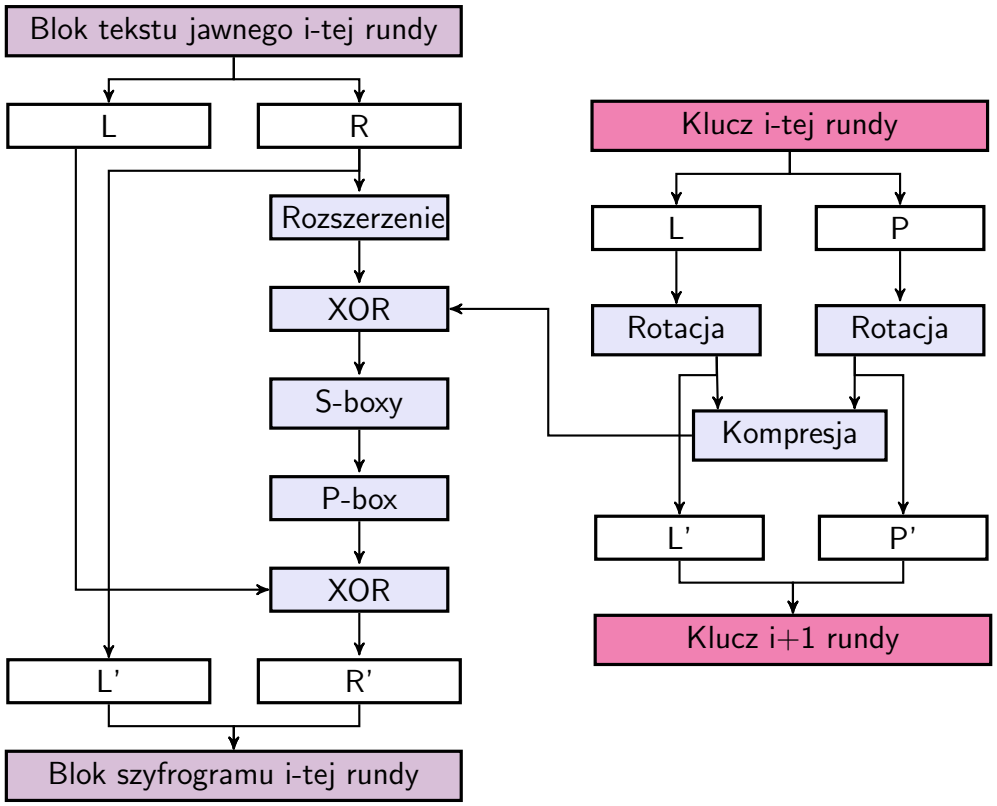
Tabela 3.8: Permutacja kompresująca $PC2$

3.3.7. Algorytm szyfrowania

Algorytm DES jest realizowany w trzech etapach. Na początku 64-bitowy blok tekstu jawnego jest poddawany działaniu permutacji początkowej IP. W drugim etapie wykonywanych jest 16 iteracji zmodyfikowanej sieci Feistela, której

schemat przedstawiony jest na rysunku 3.8. Działanie sieci podstawieniowo - permutacyjnej w jednej rundzie polega na tym, że blok wejściowy jest dzielony na dwie równe części o długości 32 bitów każdy. Prawa strona zostaje zapisana jako lewa strona szyfrogramu i - tej rundy.

Aby otrzymać prawą stronę szyfrogramu danej rundy, algorytm przekształca prawą stronę przez funkcję F szyfru, co oznacza, że 32-bity prawej strony danych wejściowych do sieci Feistela zostaje poddanych operacji rozszerzenia E, której wynikiem jest sekwencja 48 bitów, które następnie są poddawane operacji XOR, z wygenerowanym kluczem i - tej rundy. Tak otrzymany ciąg bitów jest dzielony na 8 bloków po 6-bitów. Każdy z tych bloków trafia do odpowiedniego S-boxa.



Rysunek 3.8: Schemat sieci Feistela użytej w algorytmie DES

Zwrócone przez funkcję podstawieniową ciągi 4-bitowe zostają połączone i przekształcone przez permutację P-box, a otrzymany blok danych zostaje poddany operacji XOR z lewą częścią bloku tekstu jawnego i - tej rundy. Tak otrzymana prawa strona szyfrogramu i - tej rundy połączona z lewą częścią szyfrogramu i - tej rundy jest jednocześnie blokiem tekstu jawnego $i+1$ - szej rundy. W ostatnim etapie, po wykonaniu 16 rund sieci Feistela, otrzymany z funkcji szy-

fru F ciąg bitów jest przekształcany przez permutację końcową IP^{-1}. 64-bitowy ciąg wyjściowy stanowi ostateczny szyfrogram.

3.4. Kodowanie elementów szyfru

Poniżej przedstawiono metodę bezpośredniego kodowania boolowskiego dwóch wcześniej opisanych algorytmów: sieci Feistela i DES. W pierwszej kolejności opisano procedurą kodowania przekształcenia Feistela, a następnie metodę kodowania głównych kroków w DES, w tym permutacji i bloków podstawieniowych. Idea bezpośredniego kodowania boolowskiego szyfrów symetrycznych została zaproponowana przez Kurkowskiego w pracach [55, 40] i rozwijana dalej także przez autorkę niniejszej rozprawy w [117, 118, 125, 126, 127, 128]. W porównaniu z innymi badaniami kryptoanalitycznymi korzystającymi z technik SAT metoda ta charakteryzuje się brakiem nadmiarowości liczby stosowanych do kodowania szyfru zmiennych zdaniowych.

3.4.1. Kodowanie boolowskie sieci Feistela

Dla tych rozważań przyjęto, że blok tekstu jawnego przetwarzanego przez sieć Feistela FN będzie miał długość 64 bitów, a klucz szyfrujący 32 bity. W miejsce funkcji F wstawiono funkcję XOR. Niech $p_1, \ldots, p_{64}$ będą zmiennymi zdaniowymi reprezentującymi tekst jawny, $k_1, \ldots, k_{32}$ klucz, a $c_1, \ldots, c_{64}$ szyfrogram.

Zgodnie z algorytmem Feistela 3.2 dla lewej części szyfrogramu otrzymano następującą formułę kodującą:

$$\bigwedge_{i=1}^{32} (c_i \Leftrightarrow p_{i+32}) .$$

Warto zauważyć, że $\oplus$ też jest funkcją boolowską. Stąd druga część bloku wyjściowego ma postać:

$$\bigwedge_{i=33}^{64} (c_i \Leftrightarrow (p_i \oplus k_{i-32} \oplus p_{i-32})) .$$

Cała formuła kodująca dla pojedynczej rundy algorytmu Feistela ma postać:

$$\Phi^1_{FN} : \bigwedge_{i=1}^{32} (c_i \Leftrightarrow p_{i+32}) \wedge \bigwedge_{i=33}^{64} (c_i \Leftrightarrow (p_i \oplus k_{i-32} \oplus p_{i-32})) .$$

Rozważając przypadek j- iteracji FN niech $(p^1_1, \ldots, p^1_{64}), (k_1, \ldots, k_{32})$ będą wektorami zmiennych reprezentującymi odpowiednio: tekst jawny i klucz. Przez

$(p_1^t, \ldots, p_{64}^t)$ oraz $(c_1^i, \ldots, c_{64}^i)$ oznaczmy wektory zmiennych opisujących dane wejściowe t - tej rundy dla $t = 2, \ldots, j$ i danych wyjściowych i-tej rundy dla $i = 1, \ldots, j-1$. Przyjmijmy również, że $\left(c_1^j, \ldots, c_{64}^j\right)$ jest wektorem zmiennych szyfrogramu po j - tej rundzie.

Formuła, która koduje wszystkie j rund FN jest postaci:

$$\Phi_{FN}^j = \bigwedge_{i=1}^{32} \bigwedge_{s=1}^{j} \left(c_i^s \Leftrightarrow p_{i+32}^s\right) \wedge$$
$$\bigwedge_{i=33}^{64} \bigwedge_{s=1}^{j} \left[c_i^s \Leftrightarrow \left(p_{i-32}^s \oplus k_{i-32} \oplus p_i^s\right)\right] \wedge \bigwedge_{i=1}^{64} \bigwedge_{s=1}^{j-1} (p_i^{s+1} \Leftrightarrow c_i^s).$$

Warto tu zwrócić uwagę, że ostatnia część tej formuły stwierdza, że bity wyjścia rundy s są bitami wejścia rundy $(s+1)$.

Patrząc na powyższą formułę, łatwo zauważyć, że jest ona koniunkcją odpowiednich równoważności. Z punktu widzenia konieczności zmiany postaci formuły na koniunkcyjną postać normalną jest to niezwykle ważne. Ten zapis ma jeszcze jedną zaletę: umożliwia automatyczne generowanie formuły dla wielu testowanych rund szyfru.

3.4.2. Kodowanie permutacji, rotacji, kompresji i rozszerzeń

Struktura formuły kodowania algorytmu DES jest podobna do formuły kodowania FN. Można traktować DES jako sekwencję permutacji, redukcji, rozszerzeń, XOR, obliczeń S-box i rotacji bitów klucza. Każda z tych operacji może być zakodowana jako koniunkcja odpowiednich równoważności lub implikacji.

Oznaczając przez $(p_1, \ldots, p_{64})$ sekwencję zmiennych reprezentujących bity bloku tekstu jawnego i przez $(q_1, \ldots, q_{64})$ sekwencję zmiennych reprezentujących dane wyjściowe, formuła kodująca permutacji początkowej P przyjmuje postać:

$$\bigwedge_{i=1}^{64} \left(q_i \Leftrightarrow p_{P(i)}\right).$$

W podobny sposób można zakodować wszystkie permutacje, rozszerzenia, redukcje i rotacje DES.

3.4.3. Kodowanie S-boxów

Przechodząc do kodowania S-boxów, warto przypomnieć, że każdy blok podstawieniowy użyty w algorytmie DES jest macierzą z czterema wierszami i szesnastoma kolumnami, przy czym każdy wiersz jest inną permutacją elementów

zbioru Z_{16}. Liczby te są reprezentowane w postaci binarnej jako czteroskładnikowe sekwencje bitów. Każdy S-box może być traktować jako funkcja typu $S_{box} : \{0,1\}^6 \rightarrow \{0,1\}^4$.

Dla uproszczenia przez $\bar{x}$ oznaczono ciąg $(x_1, \ldots, x_6)$, a przez $S^k_{box}(\bar{x})$ k - tą współrzędną wartości $S_{box}(\bar{x})$ dla $k = 1, 2, 3, 4$.

Każdy S-box można zakodować jako następującą formułę boolowską:

$$\Phi_{S_{box}} : \bigwedge_{\bar{x} \in \{0,1\}^6} \left(\bigwedge_{i=1}^{6} (\neg)^{1-x_i} p_i \Rightarrow \bigwedge_{j=1}^{4} (\neg)^{1-S^j_{box}(\bar{x})} q_j \right),$$

gdzie $(p_1, \ldots, p_6)$ są wektorami wejściowymi do S-boxa, a $(q_1, \ldots, q_4)$ są wektorami wyjściowymi. Dodatkowo, przez $(\neg)^0 p$ i $(\neg)^1 p$ rozumiemy odpowiednio p i $\neg p$. Korzystając z powyższego wzoru, można zakodować każdy z S-boxów użytych w rozważanych rundach DES jako 256 implikacji.

Poniżej przedstawiono przykład S-boxa i schemat jego kodowania do zdaniowych formuł logicznych. Stosując S-box S_1, który jako pierwszy został użyty w algorytmie DES oraz zasadę działania omawianych tutaj bloków podstawieniowych do $S_1(010011) = 0110$ otrzymujemy formułę postaci:

$$\wedge \begin{cases} \neg p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge p_5 \wedge p_6 \Rightarrow \neg q_1 \\ \neg p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge p_5 \wedge p_6 \Rightarrow q_2 \\ \neg p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge p_5 \wedge p_6 \Rightarrow q_3 \\ \neg p_1 \wedge p_2 \wedge \neg p_3 \wedge \neg p_4 \wedge p_5 \wedge p_6 \Rightarrow \neg q_4. \end{cases}$$

Jak można zauważyć, jeśli jakiś bit jest równy 1, to jest on reprezentowany przez zmienną zdaniową, a jeśli jakiś bit jest równy 0, to reprezentuje go zmienna zdaniowa poprzedzona negacją.

Dysponując takimi procedurami, można zakodować dowolną liczbę rund algorytmu DES jako formułę logiczną.

3.5. Translacje formuł do formatu DIMACS

Po zakodowaniu każdego elementu algorytmu DES, przechodząc do kolejnego etapu przygotowania do SAT-kryptoanalizy, otrzymane formuły zdaniowe należało zapisać w koniunkcyjnej postaci normalnej (CNF), a następnie dokonać ich konwersji do formatu akceptowanego przez SAT solvery, czyli do formatu DIMACS. Poniżej przedstawiono formuły kodujące elementy szyfru DES zapisane w CNF i fragmenty plików tekstowych zawierających formuły kodujące działanie szyfru DES w formacie DIMACS.

3.5.1. Przekształcenie Feistela

Stosując metody konwersji formuł logicznych do CNF opisane w rozdziale 2.6. na stronie 33 otrzymano następującą koniunkcję alternatyw, która jest równoważna formule przedstawiającej przekształcenie przez sieć Feistela prawej części danych wejściowych.

$$
\bigwedge \left\{ \begin{array}{c}
\neg c_{33} \vee p_1 \vee q_1 \\
c_{33} \vee \neg p_1 \vee q_1 \\
c_{33} \vee p_1 \vee \neg q_1 \\
\neg c_{33} \vee \neg p_1 \vee \neg q_1 \\
\neg c_{34} \vee p_2 \vee q_2 \\
c_{34} \vee \neg p_2 \vee q_2 \\
c_{34} \vee p_2 \vee \neg q_2 \\
\neg c_{34} \vee \neg p_2 \vee \neg q_2 \\
\vdots \\
\neg c_{63} \vee p_{31} \vee q_{31} \\
c_{63} \vee \neg p_{31} \vee q_{31} \\
c_{63} \vee p_{31} \vee \neg q_{31} \\
\neg c_{63} \vee \neg p_{31} \vee \neg q_{31} \\
\neg c_{64} \vee p_{32} \vee q_{32} \\
c_{64} \vee \neg p_{32} \vee q_{32} \\
c_{64} \vee p_{32} \vee \neg q_{32} \\
\neg c_{64} \vee \neg p_{32} \vee \neg q_{32}
\end{array} \right.
$$

Lewa część danych wyjściowych z sieci Feistela jest kodowana jako odpowiednia równoważność, a jej przekształcenie do CNF opisano w podrozdziale 3.5.2.

Kolejnym krokiem jaki został wykonany, było przekształcenie formuły boolowskiej kodującej działanie sieci Feistela do postaci DIMACS. Na rysunku 3.9 przedstawiono fragment pliku tekstowego zawierającego kodowanie (zapisane w formacie DIMACS) działania sieci Feistela w pierwszej rundzie szyfrowania algorytmem DES.

3.5.2. Permutacje, rotacje, rozszerzenia i kompresje

Podczas bezpośredniego kodowania do formuł boolowskich występujących w algorytmie DES przekształceń typu: permutacje, rotacje, rozszerzenia i kompresje wykorzystuje się równoważność, której przekształcenia do CNF dokonano

```
529 369 481 0
529 -369 481 0
529 369 -481 0
-529 -369 -481 0
-530 370 482 0
530 -370 482 0
530 370 -482 0
-530 -370 -482 0
-531 371 483 0
531 -371 483 0
531 371 -483 0
-531 -371 -483 0
      ⋮
-575 415 527 0
575 -415 527 0
575 415 -527 0
-575 -415 -527 0
-576 416 528 0
576 -416 528 0
576 416 -528 0
-576 -416 -528 0
```

Rysunek 3.9: Fragment pliku zawierającego formułę kodującą FN

zgodnie z opisem w podrozdziale 2.6.:

$$\bigwedge \begin{cases} \neg p_1 \vee q_1 \\ p_1 \vee \neg q_1 \\ \neg p_2 \vee q_2 \\ p_2 \vee \neg q_2 \\ \quad \vdots \\ \neg p_{64} \vee q_{64} \\ p_{64} \vee \neg q_{64} \end{cases}$$

Tak przedstawione formuły zapisano w formacie DIMACS. Poniżej przedstawiono fragment pliku, w którym zapisane jest przekształcenie tekstu jawnego przez permutację początkową IP.

3.5.3. S-box

Głównym spójnikiem występującym w formule boolowskiej reprezentującej bezpośrednie kodowanie S-boxów jest implikacja. Korzystając z odpowiednich

```
p cnf 128 128
  -193 58 0
  193 -58 0
  -194 50 0
  194 -50 0
  -195 42 0
  195 -42 0
  -196 34 0
  196 -34 0
  -197 26 0
  197 -26 0
  -198 18 0
  198 -18 0
  -199 10 0
  199 -10 0
   -200 2 0
   200 -2 0
  -201 60 0
  201 -60 0
       ⋮
   -256 7 0
   256 -7 0
```

Rysunek 3.10: Fragment pliku przechowującego formułę kodującą IP

praw logicznych poniższą formułę zdaniową zapisano w koniunkcyjnej postaci normalnej.

$$(p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6) \Rightarrow (q_1 \wedge q_2 \wedge q_3 \wedge q_4)$$
$$\neg(p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6) \vee (q_1 \wedge q_2 \wedge q_3 \wedge q_4)$$
$$(\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6) \vee (q_1 \wedge q_2 \wedge q_3 \wedge q_4)$$

$$(\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee q_1) \wedge$$
$$\wedge(\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee q_2) \wedge$$
$$\wedge(\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee q_3) \wedge$$
$$\wedge(\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee q_4).$$

Otrzymana formuła została zapisana w postaci DIMACS, a fragment otrzymanego pliku ilustruje rysunek 3.11.

Zgodnie z opisem metody bezpośredniego kodowania zaprezentowanej w podrozdziale 2.8. podczas kodowania szyfru DES do opisania pojedynczego bitu użyto jednej zmiennej zdaniowej, a liczby całkowite odpowiadające literałom

```
 529 530 531 532 533 534 577 0
 529 530 531 532 533 534 578 0
 529 530 531 532 533 534 579 0
529 530 531 532 533 534 -580 0
529 530 531 532 533 -534 -577 0
529 530 531 532 533 -534 -578 0
529 530 531 532 533 -534 -579 0
529 530 531 532 533 -534 -580 0
529 530 531 532 -533 534 -577 0
 529 530 531 532 -533 534 578 0
529 530 531 532 -533 534 -579 0
529 530 531 532 -533 534 -580 0
529 530 531 532 -533 -534 577 0
529 530 531 532 -533 -534 578 0
529 530 531 532 -533 -534 579 0
529 530 531 532 -533 -534 580 0
 529 530 531 -532 533 534 577 0
 529 530 531 -532 533 534 578 0
529 530 531 -532 533 534 -579 0
 529 530 531 -532 533 534 580 0
                ⋮
-529 -530 -531 -532 -533 534 -577 0
-529 -530 -531 -532 -533 534 -578 0
-529 -530 -531 -532 -533 534 -579 0
-529 -530 -531 -532 -533 534 -580 0
-529 -530 -531 -532 -533 -534 577 0
-529 -530 -531 -532 -533 -534 578 0
-529 -530 -531 -532 -533 -534 -579 0
-529 -530 -531 -532 -533 -534 580 0
```

Rysunek 3.11: Fragment pliku zawierającego formułę kodującą S-box

dobrano tak, aby otrzymany zbiór klauzul modelował działanie całego algorytmu DES. W tabeli 3.9 przedstawiono wykaz liczb całkowitych reprezentujących bity występujące w pierwszej rundzie algorytmu DES oraz sumę użytych do opisu bitów liczb całkowitych. Tabela 3.10 przedstawia liczbę użytych zmiennych zdaniowych i wygenerowanych klauzul użytych do zakodowania kolejnych rund szyfru DES.

Opracowana i przedstawiona w niniejszym rozdziale formuła kodująca działanie szyfru DES została przetestowana i jest ona równoważna działaniu algorytmu DES. Testy wykonywano na kilku etapach kodowania poszczególnych przekształceń użytych w DES, jak i po zakodowaniu całego szyfru. Na potrzeby tej weryfikacji, wartości wejściowe i wyjściowe zostały określone przy użyciu wektorów testowych podanych w pracy [129].

Input	$1, \ldots, 64$	64
Key	$65, \ldots, 128$	64
Ciphertext	$129, \ldots, 192$	64
IP	$193, \ldots, 256$	64
PC1	$257, \ldots, 284 \mid 285, \ldots, 312$	56
Key shift	$313, \ldots, 368$	56
PC2	$369, \ldots, 416$	48
Szyfrogram I rundy	$417, \ldots, 448 \mid 449, \ldots, 480$	64
E	$481, \ldots, 528$	48
XOR	$529, \ldots, 576$	48
S-box	$577, \ldots, 608$	32
P-box	$609, \ldots, 640$	32

Tabela 3.9: Rejestr zmiennych zdaniowych dla formuły kodującej pierwszą rundę DES

Liczba rund	Liczba użytych zmiennych	Liczba wygenerowanych klauzul
1	704	3424
2	1032	6224
3	1360	9024
4	1688	11824
5	2016	14624
6	2344	17424
7	2672	20224
8	3000	23024
9	3328	25824
10	3656	28624
11	3984	31424
12	4312	34224
13	4640	37024
14	4968	39824
15	5296	42624
16	5624	45424

Tabela 3.10: Dane dotyczące kodowania poszczególnych rund DES

3.6. Badania eksperymentalne

W 1994 roku Mitsuru Matsui podjął swoją pierwszą próbę kryptoanalizy DES, wykorzystując własną metodę zwaną kryptoanalizą liniową [96]. W roku 1998 *distributed.net* używając metody brute force przywraca klucz w ciągu 41 dni. I wreszcie, w styczniu 1999 roku, Electronic Frontier Foundation wraz z *distributed.net*, łamie klucz algorytmu w ciągu 22 godzin i 15 minut. Od tej pory uważa się za bezpieczne używanie tylko modyfikacji algorytmu DES o nazwie Triple DES. Obecnie można również zastosować inne, mocne modyfikacje DES.

Rozpoczynając autorskie prace wykorzystujące metodę bezpośredniego kodowania boolowskiego i techniki SAT do kryptoanalizy szyfru DES przyjęto następujące ścieżki badawcze:

1. wyznaczanie marginesu bezpieczeństwa szyfru DES w wersji oryginalnej - 16 rund lub mniejszej;

2. badanie różnych modyfikacji DES w tym:

   (a) wersje z 8 identycznymi S-boxami, wykorzystując wybrane S-boxy używane w DES tj.: $8 \times S_1$, $8 \times S_2$, $8 \times S_3$, $8 \times S_4$, $8 \times S_5$,

   (b) wersję z S-boxami zadanymi przez funkcje liniowe,

   (c) wersję z S-boxami, w których każdy bit wyjściowy jest funkcją liniową 6 bitów wejściowych kodowanych do formuły boolowskiej na dwa sposoby.

Do wykonania tych badań opracowano narzędzia generujące formułę boolowską modelującą działanie szyfru DES w każdej z 16 rund. Co więcej, formuły te są już generowane w formacie DIMACS i zapisywane do plików tekstowych w formie akceptowanej przez SAT-solvery. Dodatkowo wykonano następujące skrypty pomocnicze:

- generujące ciągi losowych bitów klucza i tekstu jawnego, które następnie są zapisywane formacie DIMACS w pliku tekstowym,

- porównujące ciągi zmiennych zdaniowych reprezentujących klucz inicjujący z ciągiem, który obliczył SAT-solver, a który odpowiada zmiennym reprezentującym klucz (porównuje klucz inicjujący z kluczem obliczonym przez SAT-solver),

- zapisujące w pliku tekstowym dane uzyskane przez SAT-solvery w formacie DIMACS.

Skrypty zostały napisane w języku Python, z którego korzystano w różnych środowiskach zaczynając od IDLE, przechodząc do Spyder, następnie do PyCharm, by znów wrócić do środowiska Spyder, napisanego w Pythonie dla Pythona. Spyder został zaprojektowany przez naukowców, inżynierów i analityków danych oraz dla nich, a wyróżnia się eksploratorem zmiennych pozwalającym na interaktywne przeglądanie i zarządzanie obiektami wygenerowanymi przez tworzony autorsko kod.

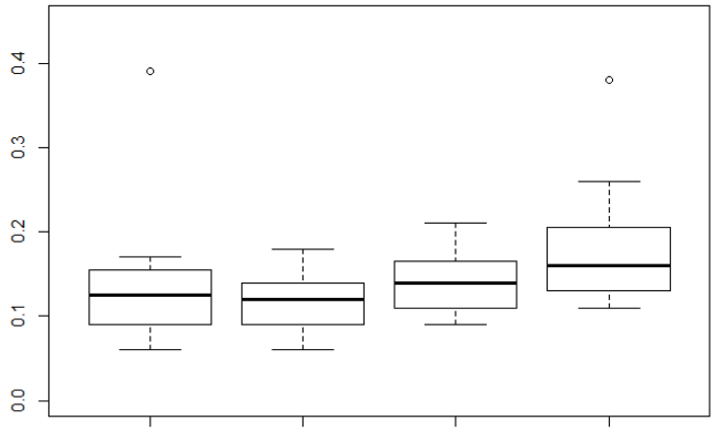
Rozpoczynając opis badań eksperymentalnych należy zastanowić się, czy i jaki wpływ na stabilność pracy SAT-solverów ma dobór wartościowania bitów tekstu jawnego i klucza (bity szyfrogramu są determinowane przez dwa ostatnie parametry).

W celu zbadania stabilności działania poszczególnych SAT-solverów przeprowadzono następujący eksperyment: dla każdego SAT-solvera wybrano próbkę losową składającą się z 64 bitów tekstu jawnego i próbkę 64 bitowego tekstu reprezentującego klucz. Dla każdego rozważanego SAT-solvera (MiniSat, Glu_vc, Glucose_syrup, Plingeling) pobrano 20 różnych tak powstałych próbek losowych a następnie każdą próbkę replikowano 20 razy dla 3 i 4 rund solvera.

W przypadku 3 rund średni czas złamania klucza dla poszczególnych 20 losowych próbek (każda zawierająca 20 elementów powtórzeń eksperymentu) dla MiniSat wynosił od 0.05 do 0.08 sekundy, gdzie mediana współczynnika zmienności (procentowy stosunek odchylenia do średniej) wynosiła 12.5%, w przypadku Glu_vc średni czas złamania klucza wynosił od 0.05 do 0.09 sekundy przy medianie współczynnika zmienności wynoszącej 12%. Dla SAT-solvera Glucose_Syrup średni czas złamania klucza wynosił od 0.04 od 0.07 sekundy przy medianie współczynnika zmienności wynoszącej 14%. W przypadku SAT-solvera Plingeling średni czas złamania klucza wynosił od 0.04 od 0.12 sekundy przy medianie współczynnika zmienności wynoszącej 16%.

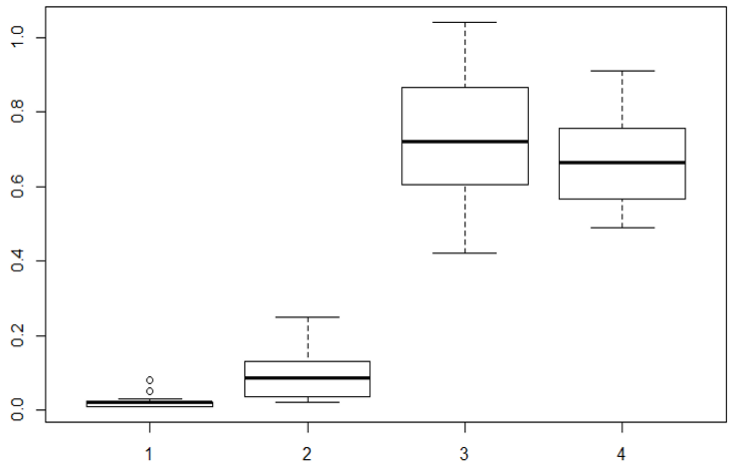
Podsumowując, można uznać, że przy 3 rundach średni czas wahał się od 0.04 sekundy do 0.12 sekundy z przeciętnym odchyleniem wynoszącym od 12% do 16% wartości średniej. Na rysunku 3.12 widzimy rozkłady współczynników zmienności dla poszczególnych czterech SAT-solverów. Więcej szczegółów podano w **Dodatku A**.

W przypadku 4 rund średni czas złamania klucza dla MiniSat wynosił od 20 sekund do 40 sekund, gdzie mediana współczynnika zmienności wynosiła 2%, w przypadku Glu_vc średni czas złamania klucza wynosił od 20 do 50 sekund przy medianie współczynnika zmienności wynoszącej 8.5%. Dla SAT-solvera Glucose_Syrup średni czas złamania klucza wynosił od 30 sekund od 60 sekund przy medianie współczynnika zmienności wynoszącej 72%. W przypadku SAT-solvera Plingeling średni czas złamania klucza wynosił od 20 od 25



Rysunek 3.12: Boxploty rozkładów współczynników zmienności dla poszczególnych SAT-solverów: MiniSat, Glu_vc, Glucose_syrup, Plingeling

sekund przy medianie współczynnika zmienności wynoszącej 66.5%. Na rysunku 3.13 widzimy rozkłady wsp. zmienności dla poszczególnych SAT-solverów. Więcej szczegółów podano w załączniku A2. Wyniki są bardzo stabilne w przypadku SAT-solverów MiniSat i Glu_vc, natomiast w przypadku SAT-solverów Glucose_Syrup i Plingeling wyniki są już mniej stabilne (odchylenie może stanowić przeciętnie około 70% średniego czasu).



Rysunek 3.13: Boxploty rozkładów współczynników zmienności dla poszczególnych SAT-solverów: MiniSat, Glu_vc, Glucose_Syrup, Plingeling

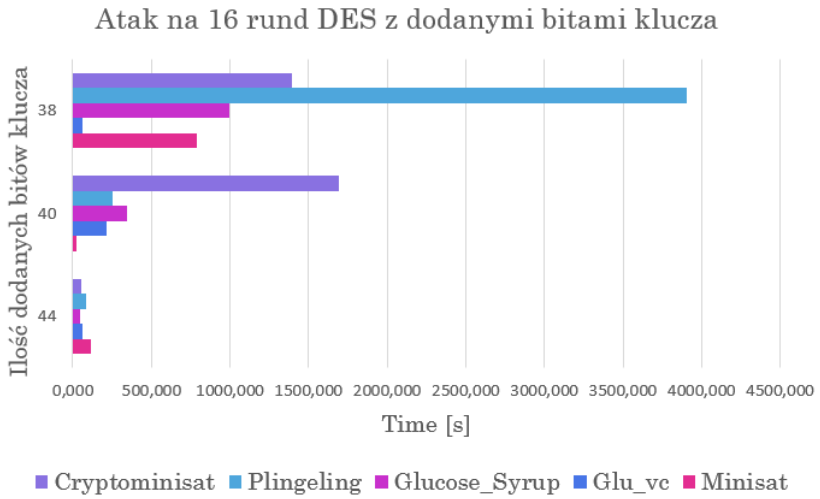
Podsumowując tę część wykonanych analiz stwierdzić można, że dobór wartości bitów tekstu jawnego i klucza nie wpływa znacząco na wyniki eksperymen-

talne w sposób mogący podważyć lub uważać za dyskusyjne wyciągane wnioski. Różnice w otrzymywanych czasach nie dotyczą rzędu lub rzędów wielkości, a jedynie są ze zrozumiałych względów odchylone od reprezentatywnej wartości przeciętnej. Szczegółowe dane dotyczące tych analiz znajdują się w **Dodatku A**. Zatem w kolejnych eksperymentach nie przeprowadzano podobnych analiz.

Pierwsze badanie kryptoanalityczne, które zostało wykonane, polegało na próbie złamania 16 rund algorytmu DES przy zastosowaniu ataku z wybranym tekstem jawnym. Jednak nawet czas jednego tygodnia nie był wystarczający, by odzyskać klucz. W związku z tym zmodyfikowano badanie i wykonano atak na 16 rund szyfru DES z dodanymi początkowymi bitami klucza. W przypadku dodania 44, 40 i 38 początkowych bitów klucza wybrane SAT-solvery, oprócz SAT-solvera Plingeling, obliczyły pozostałe bity klucza w czasie dużo krótszym niż 60 minut. Uzyskane w tym etapie badań wyniki przedstawia tabela 3.11 oraz wykres na rysunku 3.14.

		44 bity	40 bitów	38 bitów
Minisat	Time [s]	113	22,8	789
Glu_vc	Time [s]	59,7	213	63,1
Glucose_Syrup	Time [s]	47,3	344	999
Plingeling	Time [s]	83,4	253	3910
Cryptominisat	Time [s]	53,6	1690	1390

Tabela 3.11: Atak na 16 rund DES z dodanymi bitami klucza



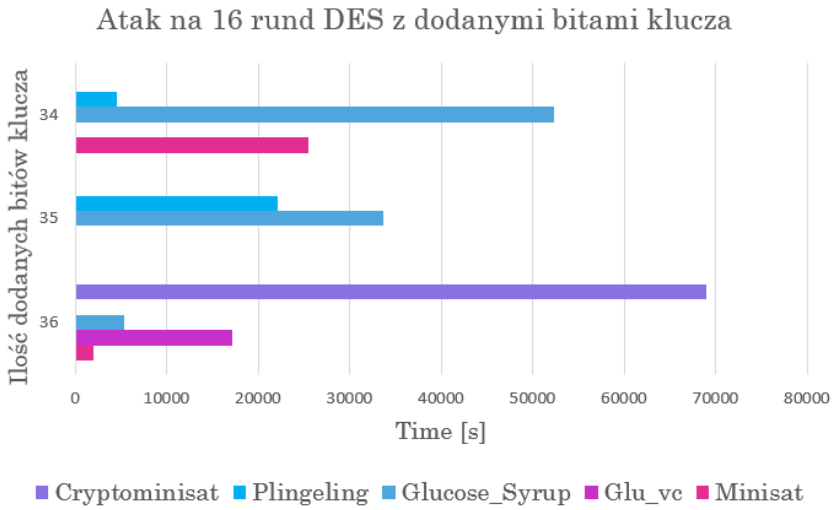
Rysunek 3.14: Atak na 16 rund DES z dodanymi bitami klucza cz. 1

Kontynuując rozpoczęte badanie poszukiwano marginesu bezpieczeństwa 16 rund algorytmu DES zmniejszając liczbę dodanych bitów klucza. Kolejno zadawano wybranym SAT-solverom obliczanie coraz to większej liczby bitów klucza. Kolejno SAT-solvery miały obliczyć pozostałe 28, 29 i i 30 bitów klucza. Różnice w czasie potrzebnym do tych obliczeń przez poszczególne SAT-solvery są duże. W przypadku obliczania 28 pozostałych bitów klucza najlepszy czas osiągnął Plingeling - 65 sekund, natomiast najwięcej czasu potrzebował Cryptominisat, bo aż ponad 19 godzin.

Taka różnica w czasach obliczeń dla różnych SAT-solverów nie jest zaskoczeniem, gdyż ich wydajność jest różna dla różnych problemów, co pokazano na przykład w pracy [109]. Wyniki tego eksperymentu obrazuje tabela 3.12 oraz wykres na rysunku 3.15.

	36 bitów	35 bitów	34 bity
Minisat Time [s]	1970	–	25500
Glu_vc Time [s]	17200	–	–
Glucose_Syrup Time [s]	5380	33700	52300
Plingeling Time [s]	64,9	22200	4620
Cryptominisat Time [s]	69100	–	–

Tabela 3.12: Atak na 16 rund DES z dodanymi bitami klucza cz. 2



Rysunek 3.15: Atak na 16 rund DES z dodanymi bitami klucza cz. 2

Kolejnym krokiem, który podjęto, aby wyznaczać margines bezpieczeństwa algorytmu DES było rozpoczęcie prób jego łamania od pierwszej rundy. Wyni-

ki uzyskane przez poszczególne SAT-solvery jednowątkowe przedstawia tabela 3.13, a przez SAT-solvery wielowątkowe tabela 3.14. Spośród SAT-solverów jednowątkowych w ataku na 4 rundy DES najlepszy czas uzyskał PicoSAT, a wśród SAT-solverów wielowątkowych PaInleSS.

	Minisat	Sat4J	PicoSAT	Glu_vc
	Time [s]	Time [s]	Time [s]	Time [s]
1 runda	0,013	0,242	0,005	0,011
2 rundy	0,019	0,288	0,016	0,026
3 rundy	0,051	0,874	0,053	0,075
4 rundy	62,2	1460	37,2	54,8

Tabela 3.13: Wyniki dla pierwszych czterech rund DES

	PaInleSS	Glucose-Syrup	PLingeling
	Time [s]	Time [s]	Time [s]
1 runda	1,08	0,012	0,016
2 rundy	1,27	0,020	0,022
3 rundy	1,18	0,079	0,141
4 rundy	18,2	89,1	25,6

Tabela 3.14: Wyniki dla pierwszych czterech rund DES uzyskane przez SAT-solvery wielowątkowe

Uzyskano wyniki, które pozwalałyby przypuszczać, iż atak na 5 rund DES będzie pomyślny. Jednak wyniki badań zaprezentowane w tabelach 3.15 oraz 3.16 pokazują, że próba odzyskania klucza w czasie krótszym niż 48 godzin nie powiodła się i badania przerwano.

Liczba dodanych bitów klucza	Minisat	Sat4J	PicoSAT	Glu_vc
	Time [s]	Time [s]	Time [s]	Time [s]
20	11,5	2010	1030	27,7
10	219	1080	–	4080
8	793	-	-	2880
6	2920	-	-	3410

Tabela 3.15: Wyniki uzyskane przez SAT-solvery jednowątkowe dla 5 rund DES

Badając atak na 5 rund DES dodano odpowiednio sformatowane bity klucza do formuły kodującej 5 rund algorytmu DES. I tak, wielowątkowy SAT-solver

PaInleSS komunikował błędy lub przepełnienie pamięci. Minisat, Glucose-Syrop, Glu_vc i Plingeling działały stabilnie. Dla 8 i 6 dodanych wartości bitów klucza Sat4J i PicoSAT nie dały wyników poniżej 24 godzin (badania przerwano).

Warto zauważyć, że najlepsze wyniki czasowe wśród jednowątkowych SAT-solverów uzyskał Minisat i wielowątkowy Plingeling.

Liczba dodanych bitów klucza	PaInleSS Time [s]	Glucose-Syrup Time [s]	PLingeling Time [s]
20	6,46	10,3	6,2
10	ERROR	7724	957
8	ERROR	2,46	520
6	ERROR	4830	1580

Tabela 3.16: Wyniki uzyskane przez SAT-solvery wielowątkowe dla 5 rund DES

Realizując kolejne zadania eksperymentalne, testowano jaki wpływ ma jakość S-boxów na problem łamania szyfrów przy użyciu techniki SAT. W tym celu badano czas potrzebny do rozwiązania problemu SAT dla 4 rund DES przy użyciu kilku wariantów S-boxów. Wyniki przeprowadzonych badań dla SAT-solverów jednowątkowych i wielowątkowych przedstawiają odpowiednio tabele 3.17 oraz 3.18.

	Minisat Time [s]	Sat4J Time [s]	PicoSAT Time [s]	Glu_vc Time [s]
DES	62,2	1455	37,2	54,8
8 x S-box_1	67,5	48,7	40,4	12,7
8 x S-box_2	43,4	1100	209	10,1
8 x S-box_3	6,06	407	38,4	63,7
8 x S-box_4	96,4	98,7	58,1	20,4
8 x S-box_5	54	606	22,8	36,6
8 x S-box_linear	5,79	434	4,63	6,02
8 x S-box_M	6,23	1,88	1,73	49,6
S-box formula	39,1	251	24,4	11,3

Tabela 3.17: Wyniki SAT-solverów jednowątkowych dla 4 rund DES i jego modyfikacji

W pierwszym przypadku sprawdzono formułę z oryginalnymi S-boxami użytymi w algorytmie DES (S-box DES). Drugi wariant analizuje formułę zawierającą osiem identycznych S-boxów, które zostały użyte w algorytmie DES jako S_1 (8 x S-box_1). W kolejnych przypadkach testowano formuły z identycznymi

	PaInleSS	Glucose-Syrup	PLingeling
	Time [s]	Time [s]	Time [s]
DES	18,2	89,1	25,6
8 x S-box_1	69,4	80,2	37,9
8 x S-box_2	30,4	74,2	13,4
8 x S-box_3	265	35,8	8,1
8 x S-box_4	140	40,2	21,6
8 x S-box_5	4,21	19,6	6,3
8 x S-box_linear	2,21	24,5	3,1
8 x S-box_M	8,22	41,0	30,7
S-box formula	33,23	10,2	1,4

Tabela 3.18: Wyniki dla 4 rund DES i jego modyfikacji

S-boxami, które zostały użyte w DES jako odpowiednio drugi, trzeci, czwarty i piąty S-box (8 x S-box_2, 8 x S-box_3, 8 x S-box_4, 8 x S-box_5).

W przypadku eksperymentu oznaczonego jako $8x$S-box_linear, rozważono S-boxy, które mogą być reprezentowane przez funkcje liniowe, takie jak: $f : \{0,..,15\} \rightarrow \{0,..,15\}$, gdzie $f(x) = (a_1 x + a_0)\,(mod 16)$, przy czym $a_i = 0,\ldots,15$ dla $i = 0,1$. Taki S-box jest zaprezentowany w tabeli 3.19. Procedura kodowania do formuły boolowskiej jest podobna do tej kodującej oryginalne S-boxy użyte w algorytmie DES.

W tabeli 3.20 przedstawiono nowo utworzony S-box, w którym każdy bit wyjściowy jest liniową funkcją sześciu bitów wejściowych.

Takie S-boxy można zakodować za pomocą metody, którą stosowaliśmy do tej pory, ale biorąc pod uwagę wspomniane wcześniej liniowe zależności między bitami wejściowymi i wyjściowymi, możemy napisać następujące cztery formuły:

$$q_1 \Leftrightarrow p_1 \oplus p_3 \oplus p_4 \oplus true,$$
$$q_2 \Leftrightarrow p_1 \oplus p_2 \oplus p_4 \oplus p_6,$$
$$q_3 \Leftrightarrow p_1 \oplus p_2 \oplus p_3 \oplus p_4 \oplus p_6 \oplus true,$$
$$q_4 \Leftrightarrow p_3 \oplus p_4 \oplus p_5 \oplus p_6 \oplus true,$$

gdzie *true* oznacza bit o wartości 1.

Te formuły kodują cały S-box. Dodatkowo wiemy, że takie formuły można przekształcić w format CNF. Rozważmy pierwszą z nich:

$$q_1 \Leftrightarrow p_1 \oplus p_3 \oplus p_4 \oplus true.$$

	x0000x	x0001x	x0010x	x0011x
0yyyy0	5	8	11	14
0yyyy1	4	9	14	3
1yyyy0	11	4	13	6
1yyyy1	13	12	11	10

	x0100x	x0101x	x0110x	x0111x
0yyyy0	1	4	7	10
0yyyy1	8	13	2	7
1yyyy0	15	8	1	10
1yyyy1	9	8	7	6

	x0000x	x0001x	x0010x	x0011x
0yyyy0	13	0	3	6
0yyyy1	12	1	6	11
1yyyy0	3	12	5	14
1yyyy1	5	4	3	2

	x0100x	x0101x	x0110x	x0111x
0yyyy0	9	12	15	2
0yyyy1	0	5	10	15
1yyyy0	7	0	9	2
1yyyy1	1	0	15	14

Tabela 3.19: Liniowy S-box

Korzystając z następującej sekwencji przekształceń:

$$\neg q_1 \Leftrightarrow p_1 \oplus p_3 \oplus p_4,$$
$$[\neg q_1 \Rightarrow p_1 \oplus p_3 \oplus p_4] \wedge [p_1 \oplus p_3 \oplus p_4 \Rightarrow \neg q_1],$$
$$[q_1 \vee (p_1 \oplus p_3 \oplus p_4)] \wedge [\neg (p_1 \oplus p_3 \oplus p_4) \vee \neg q_1],$$

i innych znanych praw logicznych możemy w końcu otrzymać rozważaną formułę w formacie CNF:

$$(q_1 \vee p_1 \vee p_3 \vee p_4) \wedge (q_1 \vee \neg p_1 \vee \neg p_3 \vee p_4) \wedge$$
$$\wedge (q_1 \vee \neg p_1 \vee p_3 \vee \neg p_4) \wedge (q_1 \vee \ p_1 \vee \neg p_3 \vee \neg p_4) \wedge$$
$$\wedge (\neg q_1 \vee \neg p_1 \vee p_3 \vee p_4) \wedge (\neg q_1 \vee \ p_1 \vee \neg p_3 \vee p_4) \wedge$$
$$\wedge (\neg q_1 \vee \ p_1 \vee p_3 \vee \neg p_4) \wedge (\neg q_1 \vee \neg p_1 \vee \neg p_3 \vee \neg p_4).$$

Ta formuła koduje pierwszą równoważność, która opisuje pierwszy bit wyjściowy w rozważanym S-Box.

	x0000x	x0001x	x0010x	x0011x
0yyyy0	11	10	4	5
0yyyy1	12	13	3	2
1yyyy0	5	4	10	11
1yyyy1	2	3	13	12

	x0100x	x0101x	x0110x	x0111x
0yyyy0	0	1	15	14
0yyyy1	7	6	8	9
1yyyy0	14	15	1	0
1yyyy1	9	8	6	7

	x0000x	x0001x	x0010x	x0011x
0yyyy0	13	12	2	3
0yyyy1	10	11	5	4
1yyyy0	3	2	12	13
1yyyy1	4	5	11	10

	x0000x	x0101x	x0110x	x0111x
0yyyy0	6	7	9	8
0yyyy1	1	0	14	15
1yyyy0	8	9	7	6
1yyyy1	15	14	0	1

Tabela 3.20: S-box_M

Uzyskane wyniki zostały opublikowane w pracy [125].

Wszystkie wyniki eksperymentalne wykonano na maszynie o następujących parametrach: Architecture: x86_64, Little Endian, 4xCore - Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, Linux VirtualBox 4.15.0-88-generic Ubuntu GNU.

3.7. Podsumowanie prac

Z przeprowadzonych testów wynika, że wykonane bezpośrednie kodowanie boolowskie szyfru blokowego DES jest równoważne jego działaniu. Wykonane eksperymenty pokazały, że algorytm DES zakodowany do formuły boolowskiej i poddany atakowi z wybranym tekstem jawnym przy użyciu narzędzi, jakimi są SAT-solvery, może zostać złamany w rozsądnym czasie w wersji zredukowanej do pięciu rund z dodanymi 6 bitami klucza. Uzyskane wyniki zostały opublikowane w pracy [125].

Analizując otrzymane wyniki należy zauważyć, że nie można jednoznacznie

określić, który z solverów SAT jest najlepszy lub najszybszy. Porównując uzyskane czasy pierwszych sześciu testów, można zauważyć, że solvery wielowątkowe SAT najszybciej poradziły sobie z atakiem na 4 rundy DES z ośmioma S-boxami nr 5, czego nie można wywnioskować z czasów uzyskanych przez solvery jednowątkowe SAT.

W eksperymentach przeprowadzonych przez autorkę rozprawy, wykorzystano wybrane SAT-solverów, a wyniki zaprezentowano dla kilku wybranych dających najlepsze rezultaty [16, 18, 19]. Dobrym przykładem jest Glu_vc - rozwiązanie z rodziny CDCL (Conflict-Driven, Clause-Learning), które wykorzystuje heurystykę VSIDS, wskazując zmienne zdaniowe preferowane do dalszego rozgałęziania drzewa. Ponadto autorzy zoptymalizowali rozwiązanie pod kątem spotykanych formuł. Glu_vc został zaprezentowany na konkursie SAT Competition w 2017 roku. Do naszych badań wykorzystano również dobrze znany i szeroko stosowany MiniSAT, a także kilka innych, w tym równoległych rozwiązań: Plingeling i Glucose-Syrup.

Badawczy wkład własny autorki rozprawy opisany w tym rozdziale jest następujący:

- powtórzono wyniki SAT-kryptoanalizy szyfru DES dla wielu, w tym nowych, SAT-solverów,
- zbadano wpływ na stabilność pracy SAT-solverów ze względu ma dobór wartościowania bitów tekstu jawnego i klucza,
- opracowano bezpośrednie kodowanie boolowskie liniowych S-boxów,
- dokonano SAT-kryptoanalizy dla nowego kodowania liniowych S-boxów,
- przeprowadzono testy, które potwierdziły, że przedstawiona translacja działania algorytmu DES do formuły boolowskiej jest równoważna jego działaniu,
- wykonano obliczenia mające na celu wyznaczenie granicy praktycznej obliczalności SAT-kryptoanalizy dla szyfru DES i jego różnych modyfikacji w obecnych realiach technicznych (software i moce obliczeniowe stosowanych maszyn).

Wszystkie prace programistyczne i obliczeniowe zostały samodzielnie wykonane przez autorkę rozprawy.

Rozdział 4.

SAT-kryptoanaliza Salsa20

W tym rozdziale zostanie przedstawiony szyfr Salsa20, wraz z jego funkcjami składowymi. W dalszej części rozdziału zostanie zaproponowane bezpośrednie kodowanie boolowskie poszczególnych elementów szyfru, a następnie jego konwersja do formatu DIMACS. W końcowej części rozdziału zostaną zaprezentowane i omówione wyniki SAT kryptoanalizy dla Salsa20 uzyskane przez wybrane SAT-solvery.

4.1. Salsa20

Salsa20 to strumieniowy szyfr symetryczny, który został opracowany w 2005 roku przez Bernsteina [12, 13]. Szyfr ten został zgłoszony do projektu eSTREAM [56] organizowanego przez sieć ECRYPT UE mającego miejsce w latach 2004-2008, którego celem było wyłonienie i zbadanie nowych szyfrów strumieniowych nadających się do powszechnego użycia.

Salsa20 jest szyfrem zorientowanym na implementacje programowe, zoptymalizowany pod kątem nowoczesnych procesorów. Wraz ze swoją odmianą ChaCha [14] został zaimplementowany w wielu protokołach i bibliotekach. Zarówno prostota, jak i szybkość algorytmu Salsa20 przyczyniły się do jego popularności wśród twórców oprogramowania.

4.2. Struktura algorytmu

Szyfr Salsa20 jest zbudowany na funkcji pseudolosowej opartej na operacjach: add-rottate-xor (ARX) – dodawaniu słów, operacji XOR i operacji rotacji. Wewnętrzny stan składa się z szesnastu 4-bajtowych słów umieszczonych w macierzy kwadratowej.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Rysunek 4.1: Stan wewnętrzny szyfru Salsa20

Stan początkowy szyfru Salsa20 złożony jest z ośmiu słów klucza, dwóch słów pozycji strumienia, dwóch słów nonce (dodatkowe bity pozycji strumienia) oraz czterech słów stałych, które zależą od długości klucza – dla klucza 32-bitowego słowa stałe mają brzmienie „expand 32-byte k", natomiast dla klucza długości 16-bajtów słowa stałe oznaczają „expand 16-byte k"[12].

„expa"	key	key	key
key	„nd 3"	nonce	nonce
pos.	pos.	„2-by"	key
key	key	key	„te k"

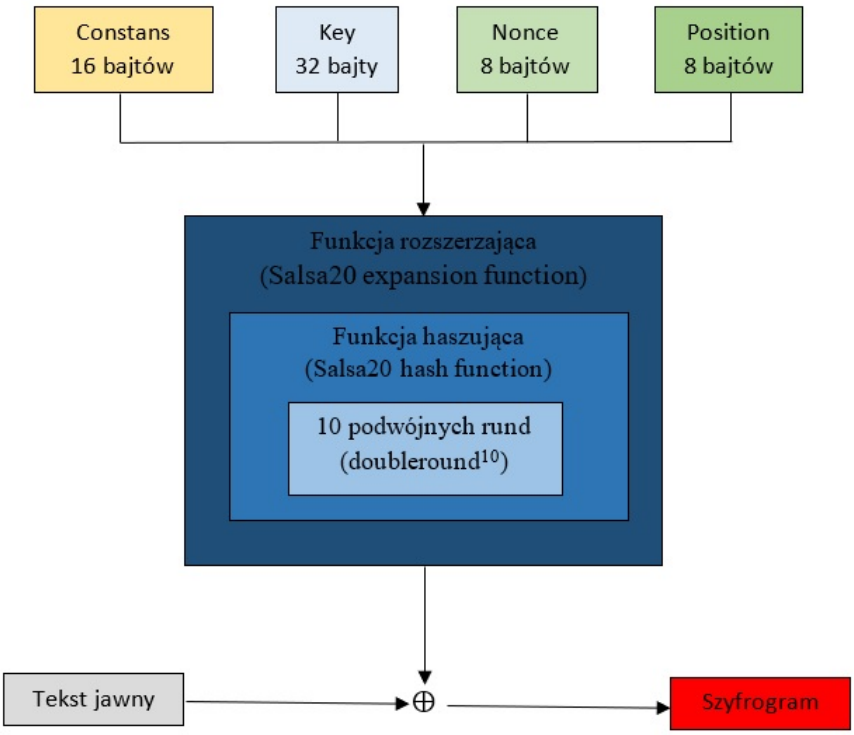
Rysunek 4.2: Stan początkowy szyfru Salsa20

Mając tak zbudowany 64-bajtowy ciąg danych wejściowych algorytm wywołuje funkcję rozszerzającą Salsa20 (expansion function Salsa20), odpowiednią w zależności od długości klucza, która wykorzystując funkcję haszującą (hash function Salsa20) zwraca 64-bajtową sekwencję danych wyjściowych. W ostatnim kroku dane zwrócone przez funkcję rozszerzającą są xorowane z 64-bajtowym ciągiem tekstu jawnego. W ten sposób otrzymujemy szyfrogram o długości 64-bajtów.

Działanie funkcji haszującej polega na podzieleniu otrzymanego z funkcji rozszerzającej 64-bajtowego ciągu na 4-bajtowe słowa, następnie poddaniu każdego słowa działaniu funkcji litleendian, mającej za zadanie zmianę kolejności bajtów w słowie. W kolejnym kroku algorytm tak otrzymaną sekwencję słów przekształca przy pomocy 10 rund funkcji doubleround, która składa się z funkcji rundy kolumn i rundy wierszy. Funkcje te poddają modyfikacjom odpowiednio kolumny i wiersze macierzy słów używając do tego funkcji quarterrouround, która przekształca słowa przy pomocy funkcji dodawania oraz rotacji.

4.2.1. Podstawowe operacje

Poniżej zostaną przedstawione elementarne operacje na 4-bajtowych słowach, które są wykorzystywane w funkcjach użytych do budowy algorytmu



Rysunek 4.3: Schemat działania szyfru Salsa20

Salsa20. Przez bajt rozumiemy element ze zbioru $\{0, 1, \ldots, 255\}$, natomiast słowo jest elementem zbioru $\{0, 1, \ldots, 2^{32} - 1\}$. Słowa można zapisać w systemie szesnastkowym, używamy wtedy oznaczenia $0x$, na przykład $0xa0b78e07 = 10 \cdot 2^{28} + 0 \cdot 2^{24} + 11 \cdot 2^{20} + 7 \cdot 2^{16} + 8 \cdot 2^{12} + 14 \cdot 2^8 + 0 \cdot 2^4 + 7 \cdot 2^0 = 2696384007$.

Suma dwóch słów. Dodawanie dwóch 4-bajtowych słów wykonujemy według wzoru:

$$(u + v) \ (mod \ 2^{32}).$$

Wynik tego działania ma 4-bajty długości. Na przykład

$$0x1234abcd + 0x56789ef0 = 0x68ad4abd.$$

XOR dwóch słów. Operację XOR dwóch słów u i v, zapiszemy jako $u \oplus v$, jest sumą u i v bez przeniesienia. Innymi słowy jeśli $u = \sum_i 2^i u_i$ oraz $v = \sum_i 2^i v_i$, to $u \oplus v = \sum_i 2^i \left(u_i + v_i - 2u_i v_i\right)$. Na przykład

$$0x1234abcd \oplus 0x56789ef0 = 0x444c353d.$$

Rotacja słowa u o w bitów w lewo. Dla $w\epsilon\{0,1,2,3,\dots\}$ rotację słowa u o w bitów w lewo oznaczamy $u \lll w$ i jej wynikiem jest unikalne, niezerowe słowo, równe $2^w u \ (mod\ 2^{32}-1)$, z wyjątkiem $0 \lll w = 0$. Innymi słowy, jeśli $u = \sum_i 2^i u_i$, to $u \lll w = \sum_i 2^{i+wmod32} u_i$. Na przykład

$$0x1234abcd \lll 4 = 0x234abcd1.$$

4.2.2. Funkcja ćwiartki rundy

W głównej permutacji w szyfrze Salsa20 zastosowana jest funkcja ćwiartki rundy (ang. quarterround function), która jako dane wejściowe przyjmuje 4 słowa i zwraca ciąg o długości 4 słów. Jeśli przez y oznaczymy sekwencję 4 słów i $y = (y_0, y_1, y_2, y_3)$, to funkcja ćwiartki rundy jest zdefiniowana następująco: $quarterround(y) = (z_0, z_1, z_2, z_3)$, gdzie

$$z_1 = y_1 \oplus ((y_0 + y_3) \lll 7),$$

$$z_2 = y_2 \oplus ((z_1 + y_0) \lll 9),$$

$$z_3 = y_3 \oplus ((z_2 + z_1) \lll 13),$$

$$z_0 = y_0 \oplus ((z_3 + z_2) \lll 18).$$

Funkcja quarterround wykonuje się w miejscu, to znaczy bez potrzeby przydzielania dodatkowej pamięci: najpierw y_1 zmienia się w z_1, następnie y_2 zmienia się w z_2, następnie y_3 przekształca się w z_3, a na końcu y_0 zmienia się w z_0. Ponieważ wszystkie operacje użyte podczas wykonywania funkcji ćwiartki rundy są odwracalne to funkcja quarterround jest odwracalna [12].

4.2.3. Funkcja rundy wierszy

Kolejną funkcją użytą w algorytmie Salsa20 jest funkcja rundy wierszy (ang. rowround fuction) [12]. Niech y oznacza ciąg 16 słów. Wtedy $rowround(y)$ jest także ciągiem złożonym z 16 słów. Jeśli $y = (y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9, y_{10}, y_{11}, y_{12}, y_{13}, y_{14}, y_{15})$, to

$$rowround\,(y) = (z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, z_9, z_{10}, z_{11}, z_{12}, z_{13}, z_{14}, z_{15}),$$

przy czym

$$(z_0, z_1, z_2, z_3) = quarterround(y_0, y_1, y_2, y_3),$$
$$(z_5, z_6, z_7, z_4) = quarterround(y_5, y_6, y_7, y_4),$$
$$(z_{10}, z_{11}, z_8, z_9) = quarterround\,(y_{10}, y_{11}, y_8, y_9)\,,$$
$$(z_{15}, z_{12}, z_{13}, z_{14}) = quarterround\,(y_{15}, y_{12}, y_{13}, y_{14})\,.$$

Przykładowo, dla drugiej równości, tzn.

$$(z_5, z_6, z_7, z_4) = quarterround\,(y_5, y_6, y_7, y_4)\,,$$

zgodnie z definicją funkcji quarterround otrzymujemy:

$$z_6 = y_6 \oplus ((y_5 + y_4) \lll 7)\,,$$
$$z_7 = y_7 \oplus ((z_6 + y_5) \lll 9)\,,$$
$$z_4 = y_4 \oplus ((z_7 + z_6) \lll 13)\,,$$
$$z_5 = y_5 \oplus ((z_4 + z_7) \lll 18)\,.$$

A zatem funkcję rundy wierszy można zapisać w postaci 16 równań tej postaci.

Dane wejściowe do funkcji rundy wierszy można przedstawić w postaci macierzy 4×4:

$$\begin{bmatrix} y_0 & y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 & y_7 \\ y_8 & y_9 & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{bmatrix}$$

Funkcja rundy wierszy poddaje modyfikacjom wiersze tej macierzy używając do tego funkcji quarterrouround. W pierwszym wierszu, funkcja rowround poddaje modyfikacji najpierw y_1, potem y_2, następnie y_3, a na końcu y_0; w drugim wierszu funkcja rowround modyfikuje najpierw y_6, potem y_7, następnie y_4, a na końcu y_5; w trzecim wierszu funkcja rowround modyfikuje najpierw y_{11}, potem y_8, następnie y_9, a na końcu y_{10}; w czwartym wierszu funkcja rowround przekształca najpierw y_{12}, potem y_{13}, następnie y_{14}, a na końcu y_{15}.

4.2.4. Funkcja rundy kolumn

Działanie funkcji rundy kolumn (ang. columnround function) jest analogicznie, jak działanie funkcji rundy wierszy. Jeśli przez x oznaczymy sekwencję 16 słów, wtedy $columnround(x)$ jest także sekwencją złożoną z 16 słów. Jeżeli $x = (x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15})$, to $columnround\,(x) = y = (y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9, y_{10}, y_{11}, y_{12}, y_{13}, y_{14}, y_{15})$, gdzie

$$
\begin{aligned}
(y_0, y_4, y_8, y_{12}) &= quarterround(x_0, x_4, x_8, x_{12}),\\
(y_5, y_9, y_{13}, y_1) &= quarterround\,(x_5, x_9, x_{13}, x_1)\,,\\
(y_{10}, y_{14}, y_2, y_6) &= quarterround\,(x_{10}, x_{14}, x_2, x_6)\,,\\
(y_{15}, y_3, y_7, y_{11}) &= quarterround\,(x_{15}, x_3, x_7, x_{11})\,.
\end{aligned}
$$

Można zapisać formułę równoważną do niej i ma ona postać:
$(y_0, y_4, y_8, y_{12}, y_1, y_5, y_9, y_{13}, y_2, y_6, y_{10}, y_{14}, y_3, y_7, y_{11}, y_{15}) = rowround(x_0, x_4, x_8, x_{12}, x_1, x_5, x_9, x_{13}, x_2, x_6, x_{10}, x_{14}, x_3, x_7, x_{11}, x_{15})$.

Podobnie, jak w przypadku funkcji rundy wierszy dane wejściowe do funkcji rundy kolumn $(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15})$ można przedstawić jako macierz kwadratową:

$$
\begin{bmatrix}
x_0 & x_1 & x_2 & x_3\\
x_4 & x_5 & x_6 & x_7\\
x_8 & x_9 & x_{10} & x_{11}\\
x_{12} & x_{13} & x_{14} & x_{15}
\end{bmatrix}
$$

Funkcja columnround jest jest niczym innym jak transpozycją funkcji rowround: modyfikuje kolumny macierzy wprowadzając permutację każdej kolumny dzięki funkcji quarterround. I tak, w pierwszej kolumnie, funkcja ta modyfikuje najpierw x_4, potem x_8, następnie x_{12}, a na końcu x_0; w drugiej kolumnie funkcja columnround modyfikuje najpierw x_9, potem x_{13}, następnie x_1, a na końcu x_5; w trzeciej kolumnie funkcja columnround modyfikuje najpierw x_{14}, potem x_2, następnie x_6, a na końcu x_{10}; w czwartej kolumnie funkcja columnround przekształca najpierw x_3, potem x_7, następnie x_{11}, a na końcu x_{15}.

4.2.5. Funkcja podwójnej rundy

Argumentem funkcji podwójnej rundy (ang. doubleround function) jest ciąg 16 słów oznaczony przez x. Funkcja zwraca sekwencję 16 słów zgodnie z definicją:

$$doubleround\,(x) = rowround(columnround\,(x))$$

Funkcja doubleround w pierwszej kolejności poddaje modyfikacji równoległe kolumny danych wejściowych, a następnie dokonuje przekształceń na równoległych wierszach danych, otrzymanych w wyniku poprzedniej operacji. Każde słowo ulega podwójnej modyfikacji.

4.2.6. Funkcja zmiany porządku bajtów

Elementem opisanej w następnym podrozdziale funkcji haszującej SALSA20 jest funkcja zmiany porządku bajtów (ang. littleendian function), której działanie polega na zmianie kolejności bajtów w słowie. Przyjmijmy, że b jest 4-bajtową sekwencją postaci $b = (b_0, b_1, b_2, b_3)$. Wówczas $littleendian(b)$ jest słowem:

$$littleedian\,(b) = b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3.$$

4.2.7. Funkcja haszująca Salsa20

Niech x będzie 64-bajtowym ciągiem. Wówczas funkcję haszującą Salsa20 (ang. Salsa20 hash function) można opisać wzorem:

$$Salsa20\,(x) = x + doubleround^{10}\,(x)\,.$$

Zatem $Salsa20\,(x)$ jest również ciągiem 64-bajtowym.

Przyglądając się funkcji haszującej Salsa20 można zauważyć, że wejściowy 64-bajtowy ciąg x przekształca ona na sekwencję o długości 16 słów $(x_0, x_1, \ldots, x_{15})$ w ten sposób, że jeśli $x = (x\,[0]\,, x\,[1]\,, x\,[2]\,, \ldots, x[63])$, to

$$\begin{aligned} x_0 &= littleendian\,(x\,[0]\,, x\,[1]\,, x\,[2]\,, x\,[3]) \\ x_1 &= littleendian(x\,[4]\,, x\,[5]\,, x\,[6]\,, x[7]) \\ &\vdots \\ x_{15} &= littleendian(x\,[60]\,, x\,[61]\,, x\,[62]\,, x[63]). \end{aligned}$$

Następnie ciąg $(x_0, x_1, \ldots, x_{15})$ jest modyfikowany w trakcie 10 iteracji funkcji doubleround.

W końcowym etapie funkcji haszującej Salsa20 następuje dodawanie 16 wejściowych słów i 16 słów otrzymanych jako dane wyjściowe z 10 iteracji funkcji doubleround i wynik ten jest zamieniany na ciąg 64-bajtowy przy użyciu funkcji odwrotnej do funkcji littleendian. Wyjściem z funkcji haszującej Salsa20 jest 64-bajtowa konkatenacja słów:

$$\begin{gathered} littleendian^{-1}\left(z_0 + x_0\right), \\ littleendian^{-1}\left(z_1 + x_1\right), \\ \vdots \\ littleendian^{-1}\left(z_{15} + x_{15}\right). \end{gathered}$$

4.2.8. Funkcja rozszerzająca Salsa20

Elementem algorytmu szyfrującego SALSA20 jest funkcja rozszerzająca Salsa20 (ang. Salsa20 expansion function). Niech k będzie 32-bajtowym lub 16-bajtowym ciągiem oraz niech n będzie sekwencję 16 bajtów. Wtedy $Salsa20_k(n)$ jest ciągiem 64-bajtów. Jeśli ciąg k jest 32- bajtowy, to jest on dzielony na dwa 16-bajtowe podciągi k_0 i k_1, a tekst „expand 32-byte k" zapisany w kodzie ASCII jest przedstawiony w postaci: $\sigma_0 = (101, 120, 112, 97,\ \sigma_1 = (110, 100, 32, 51)$, $\sigma_2 = (50, 45, 98, 121)$, $\sigma_3 = (116, 101, 32, 107)$. Wówczas funkcja rozszerzająca Salsa20 jest definiowana przy użyciu funkcji haszującej Salsa20 w sposób zapisany poniżej [12]:

$$Salsa20_{k_0,k_1}\left(n\right) = Salsa20(\sigma_0, k_0, \sigma_1, n, \sigma_2, k_1, \sigma_3).$$

Gdy ciąg k jest 16-bajtowy, a tekst „expand 16-byte k" w kodzie ASCII jest zapisany jest w postaci $\tau_0 = (101, 120, 112, 97)$, $\tau_1 = (110, 100, 32, 49)$, $\tau_2 = (54, 45, 98, 121)$, $\tau_3 = (116, 101, 32, 107)$, wówczas funkcja rozszerzająca Salsa20 jest definiowana przy użyciu funkcji haszującej Salsa20 w następujący sposób:

$$Salsa20_k\left(n\right) = Salsa20(\tau_0, k, \tau_1, n, \tau_2, k, \tau_3).$$

4.2.9. Funkcja szyfrująca Salsa20

Ostatnim przekształceniem, który jest użyty w szyfrze Salsa20, jest funkcja szyfrująca Salsa20 (ang. Salsa20 encryption function). Niech k będzie 32-bajtowym lub 16-bajtowym ciągiem, v będzie 8-bajtową sekwencją i niech m będzie l-bajtową sekwencją dla $l \in \{0, 1, \ldots, 2^{70}\}$. Wówczas funkcja Salsa20 szyfrująca dane m z kluczem k i nonce v oznaczamy $Salsa20_k(v) \oplus m$ jest ciągiem o długości l-bajtów. Wtedy k jest tajnym kluczem, najlepiej gdyby był

32-bitowy; m jest tekstem jawnym; v jest nonce, czyli unikalnym numerem wiadomości oraz $Salsa20_k(v) \oplus m$ jest szyfrogramem. Możne też być tak, że m jest szyfrogramem, a w tym przypadku $Salsa20_k(v) \oplus m$ jest oryginalnym tekstem jawnym.

$Salsa20_k(v)$ jest ciągiem 2^{70}-bajtowym takim, że:

$$Salsa20_k(v,0), Salsa20_k(v,1), Salsa20_k(v,2), \ldots, Salsa20_k(v,2^{64}-1),$$

gdzie i jest unikalną 8-bajtową sekwencją $(i_0, i_1, \ldots, i_7)$ taką, że:

$$i = i_0 + 2^8 i_1 + 2^{16} i_2 + ... + 2^{56} i_7.$$

Formuła $Salsa20_k(v) \oplus m$ niejawnie obcina $Salsa20_k(v)$ do tej samej długości co m. To znaczy:

$$Salsa20_k(v) \oplus (m[0], m[1], ..., m[l-1]) = (c[0], c[1], ..., c[l-1]),$$

gdzie $c[i] = m[i] \oplus Salsa20_k(v, [i/64])[i \ (mod\ 64)]$ [12].

4.3. Kodowanie elementów szyfru

W tym podrozdziale przedstawiono wykonane przez autorkę rozprawy translacje opisanych wcześniej funkcji użytych w algorytmie Salsa20 do formuł logicznych przy użyciu metody bezpośredniego kodowania boolowskiego. W pierwszej kolejności opisano procedurę kodowania podstawowych operacji: dodawanie dwóch słów, XOR dwóch słów oraz rotacja słowa u o w bitów w lewo. Następnie podano opis metody kodowania operacji funkcji quarterround, by na koniec przedstawić kodowanie funkcji littlendian.

4.3.1. Kodowanie podstawowych operacji

Suma dwóch słów. Niech $u = (u_1, u_2, \ldots, u_{32})$ i $v = (v_1, v_2, \ldots, v_{32})$ będą wektorami słów o długości 4 bajtów, gdzie $u_i \in \{0,1\}$ i $v_i \in \{0,1\}$, dla $i = 1, 2, \ldots, 32$, a c_i oznacza bit przeniesienia, $c_i \in \{0,1\}$, dla $i = 1, \ldots, 32$ oraz niech wektor $(q_1, q_2, \ldots, q_{32})$, gdzie $q_i \in \{0,1\}$, dla $i = 1, 2, \ldots, 32$ oznacza wynik dodawania dwóch słów u oraz v.

Przy przyjętych oznaczeniach kodowanie boolowskie sumowania dwóch słów ma postać:

$$\bigwedge_{i=1,\ldots,32} q_i \Leftrightarrow (u_i \oplus v_i) \oplus c_i,$$

przy czym $c_{32} = 0$ i $c_i \Leftrightarrow (c_{i+1} \wedge u_{i+1}) \vee (c_{i+1} \wedge v_{i+1}) \vee (u_{i+1} \wedge v_{i+1})$ dla $i = 1, \ldots, 31$.

XOR dwóch słów. Niech $u = (u_1, u_2, \ldots, u_{32})$ i $v = (v_1, v_2, \ldots, v_{32})$ będą wektorami słów odpowiednio u i v oraz $u_i \in \{0, 1\}$ i $v_i \in \{0, 1\}$, dla $i = 1, 2, \ldots, 32$. Wektor $(q_1, q_2, \ldots, q_{32})$, gdzie $q_i \in \{0, 1\}$, dla $i = 1, 2, \ldots, 32$, będzie przedstawiał wynik funkcji XOR. Wtedy kodowanie boolowskie operacji XOR dwóch 4-bajtowych słów u i v wygląda następująco:

$$\bigwedge_{i=1,\ldots,32} q_i \Leftrightarrow (u_i \oplus v_i)\,.$$

Rotacja o w-bitów w lewo. Niech $(u_1, u_2, \ldots, u_{32})$, gdzie $u_i \in \{0, 1\}$, dla $i = 1, 2, \ldots, 32$ będzie wektorem przedstawiającym słowo 4-bajtowe, a w będzie liczbą ze zbioru $\{7, 9, 13, 18\}$ oraz niech wektor $(q_1, q_2, \ldots, q_{32})$, gdzie $q_i \in \{0, 1\}$, dla $i = 1, 2, \ldots, 32$, będzie reprezentował słowo u po operacji rotacji. Wówczas rotacja słowa u o w bitów w lewo zapisana za pomocą formuły boolowskiej ma postać:

$$\bigwedge_{w\in\{7,9,13,18\}} \left(\bigwedge_{i=1,\ldots,32-w} q_i \Leftrightarrow u_{i+w} \wedge \bigwedge_{i=32-w+1,\ldots,32} q_i \Leftrightarrow u_{i-(32-w)} \right).$$

4.3.2. Kodowanie funkcji quarterround

Niech $(y_{0_1}, y_{0_2}, \ldots, y_{0_i})$, $(y_{1_1}, y_{1_2}, \ldots, y_{1_i})$, $(y_{2_1}, y_{2_2}, \ldots, y_{2_i})$, $(y_{3_1}, y_{3_2}, \ldots, y_{3_i})$, gdzie y_{0_i}, y_{1_i}, y_{2_i}, $y_{3_i} \in \{0, 1\}$ dla $i = 1, 2, \ldots, 32$ będą wektorami przedstawiającymi dane wejściowe do funkcji ćwiartki rundy oraz niech $(z_{0_1}, z_{0_2}, \ldots, z_{0_i})$, $(z_{1_1}, z_{1_2}, \ldots, z_{1_i})$, $(z_{2_1}, z_{2_2}, \ldots, z_{2_i})$, $(z_{3_1}, z_{3_2}, \ldots, z_{3_i})$, gdzie $z_{0_i}, z_{1_i}, z_{2_i}, z_{3_i} \in \{0, 1\}$ dla $i = 1, 2, \ldots, 32$, będą wektorami reprezentującymi dane wyjścia.

Funkcja $quarterround(y_0, y_1, y_2, y_3)$ w postaci kodowania boolowskiego ma

następujący zapis:

$$\wedge \begin{cases} \bigwedge\limits_{i=1\ldots25} z_{1_i} \Leftrightarrow y_{1_i} \oplus \Big(\Big(y_{0_{i+7}} \oplus y_{3_{i+7}}\Big) \oplus c_{i+7}\Big) \\ \bigwedge\limits_{i=26\ldots32} z_{1_i} \Leftrightarrow y_{1_i} \oplus \Big(\Big(y_{0_{i-25}} \oplus y_{3_{i-25}}\Big) \oplus c_{i-25}\Big) \\ \bigwedge\limits_{i=1\ldots23} z_{2_i} \Leftrightarrow y_{2_i} \oplus \Big(\Big(z_{1_{i+9}} \oplus y_{0_{i+9}}\Big) \oplus d_{i+9}\Big) \\ \bigwedge\limits_{i=24\ldots32} z_{2_i} \Leftrightarrow y_{2_i} \oplus \Big(\Big(z_{1_{i-23}} \oplus y_{0_{i-23}}\Big) \oplus d_{i-23}\Big) \\ \bigwedge\limits_{i=1\ldots19} z_{3_i} \Leftrightarrow y_{3_i} \oplus \Big(\Big(z_{2_{i+13}} \oplus z_{1_{i+13}}\Big) \oplus e_{i+13}\Big) \\ \bigwedge\limits_{i=20\ldots32} z_{3_i} \Leftrightarrow y_{3_i} \oplus \Big(\Big(z_{2_{i-19}} \oplus z_{1_{i-19}}\Big) \oplus e_{i-19}\Big) \\ \bigwedge\limits_{i=1\ldots14} z_{0_i} \Leftrightarrow y_{0_i} \oplus \Big(\Big(z_{3_{i+18}} \oplus z_{2_{i+18}}\Big) \oplus f_{i+18}\Big) \\ \bigwedge\limits_{i=15\ldots32} z_{0_i} \Leftrightarrow y_{0_i} \oplus \Big(\Big(z_{3_{i-14}} \oplus z_{2_{i-14}}\Big) \oplus f_{i-14}\Big). \end{cases}$$

gdzie c_i, d_i, e_i, f_i oznaczają bity przeniesienia w odpowiednich działaniach dodawania dwóch słów oraz $c_i,\ d_i,\ e_i,\ f_i \in \{0,1\}$ dla $i = 1, \ldots, 32$.

4.3.3. Kodowanie funkcji littlendian

Niech $(u_1, u_2, \ldots, u_{32})$, gdzie $u_i \in \{0,1\}$, dla $i = 1, 2, \ldots, 32$ będzie wektorem przedstawiającym słowo 4-bajtowe oraz niech wektor $(q_1, q_2, \ldots, q_{32})$, gdzie $q_i \in \{0,1\}$, dla $i = 1, 2, \ldots, 32$, będzie reprezentował słowo u po modyfikacji wprowadzonej przez funkcję littleendian. Wówczas można przedstawić kodowanie boolowskie funkcji littleendian w następujący sposób:

$$\bigwedge_{i=1,\ldots,8} q_i \Leftrightarrow u_{i+24} \wedge \bigwedge_{i=9,\ldots,16} q_i \Leftrightarrow u_{i+8} \wedge \bigwedge_{i=17,\ldots,24} q_i \Leftrightarrow u_{i-8} \wedge \bigwedge_{i=25,\ldots,32} q_i \Leftrightarrow u_{i-24}.$$

Można zauważyć, że funkcja littleendian jest odwracalna.

4.4. Translacje formuł do formatu DIMACS

Biorąc pod uwagę opisane w podrozdziale 4.3. formuły oraz metodę ich przekształcania zawartą w 2.6. w niniejszym podrozdziale przedstawimy transformację formuł boolowskich kodujących działanie poszczególnych funkcji szyfru Salsa20 do koniunkcyjnej postaci normalnej, a następnie do formatu akceptowalnego przez SAT-solvery, czyli do formatu DIMACS.

4.4.1. Podstawowe operacje

W przypadku operacji **sumy dwóch słów** otrzymujemy następującą koniunkcję alternatyw:

$$
\bigwedge \begin{cases}
\neg q_1 \vee u_1 \vee v_1 \vee c_0 \\
\neg q_1 \vee \neg u_1 \vee \neg v_1 \vee c_0 \\
\neg q_1 \vee \neg u_1 \vee v_1 \vee \neg c_0 \\
\neg q_1 \vee u_1 \vee \neg v_1 \vee \neg c_0 \\
q_1 \vee \neg u_1 \vee v_1 \vee c_0 \\
q_1 \vee u_1 \vee \neg v_1 \vee c_0 \\
q_1 \vee u_1 \vee v_1 \vee \neg c_0 \\
q_1 \vee \neg u_1 \vee \neg v_1 \vee \neg c_0 \\
\neg q_2 \vee u_2 \vee v_2 \vee c_1 \\
\neg q_2 \vee \neg u_2 \vee \neg v_2 \vee c_1 \\
\neg q_2 \vee \neg u_2 \vee v_2 \vee \neg c_1 \\
\neg q_2 \vee u_2 \vee \neg v_2 \vee \neg c_1 \\
q_2 \vee \neg u_2 \vee v_2 \vee c_1 \\
q_2 \vee u_2 \vee \neg v_2 \vee c_1 \\
q_2 \vee u_2 \vee v_2 \vee \neg c_1 \\
q_2 \vee \neg u_2 \vee \neg v_2 \vee \neg c_1 \\
\vdots \\
\neg q_{32} \vee u_{32} \vee v_{32} \vee c_{31} \\
\neg q_{32} \vee \neg u_{32} \vee \neg v_{32} \vee c_{31} \\
\neg q_{32} \vee \neg u_{32} \vee v_{32} \vee \neg c_{31} \\
\neg q_{32} \vee u_{32} \vee \neg v_{32} \vee \neg c_{31} \\
q_{32} \vee \neg u_{32} \vee v_{32} \vee c_{31} \\
q_{32} \vee u_{32} \vee \neg v_{32} \vee c_{31} \\
q_{32} \vee u_{32} \vee v_{32} \vee \neg c_{31} \\
q_{32} \vee \neg u_{32} \vee \neg v_{32} \vee \neg c_{31}.
\end{cases}
$$

Na rysunku 4.4 widzimy fragment pliku, w którym zapisana jest formuła kodująca działanie funkcji dodawania dwóch słów wykonywana w pierwszej rundzie szyfru Salsa20 podczas działania przekształcenia quarterround.

Dodatkowo do zakodowania funkcji dodawania dwóch słów należy przedstawić bity przeniesienia (ang. carry) w postaci formuły boolowskiej, co zostało

```
-2049 2945 2817 1537 0
-2049 -2945 -2817 1537 0
-2049 -2945 2817 -1537 0
-2049 2945 -2817 -1537 0
2049 -2945 2817 1537 0
2049 2945 -2817 1537 0
2049 2945 2817 -1537 0
2049 -2945 -2817 -1537 0
-2050 2946 2818 1538 0
-2050 -2946 -2818 1538 0
-2050 -2946 2818 -1538 0
-2050 2946 -2818 -1538 0
2050 -2946 2818 1538 0
2050 2946 -2818 1538 0
2050 2946 2818 -1538 0
2050 -2946 -2818 -1538 0
       ⋮
-2080 2976 2848 0
2080 -2976 2848 0
2080 2976 -2848 0
-2080 -2976 -2848 0
```

Rysunek 4.4: Fragment pliku z formułą kodującą sumę dwóch słów

omówione w podrozdziale 4.3., a formuła ta po przekształceniu do CNF przyjmuje postać:

$$
\wedge \begin{cases}
\neg c_{32} \\
\neg c_{31} \vee u_{31} \\
\neg c_{31} \vee v_{31} \\
c_{31} \vee \neg u_{31} \vee \neg v_{31} \\
\neg c_{30} \vee c_{30} \vee u_{30} \\
\neg c_{30} \vee c_{30} \vee v_{30} \\
\neg c_{30} \vee u_{30} \vee v_{30} \\
c_{30} \vee \neg c_{30} \vee \neg u_{30} \\
c_{30} \vee \neg c_{30} \vee \neg v_{30} \\
c_{30} \vee \neg u_{30} \vee \neg v_{30} \\
\vdots \\
\neg c_1 \vee c_1 \vee u_1 \\
\neg c_1 \vee c_1 \vee v_1 \\
\neg c_1 \vee u_1 \vee v_1 \\
c_1 \vee \neg c_1 \vee \neg u_1 \\
c_1 \vee \neg c_1 \vee \neg v_1 \\
c_1 \vee \neg u_1 \vee \neg v_1
\end{cases}
$$

Na rysunku 4.5 przedstawiono wycinek z pliku tekstowego zawierającego formułę kodującą bity przeniesienia w formacie DIMACS występujące podczas operacji dodawania dwóch słów.

```
-1568 0
-1567 2976 0
-1567 2848 0
1567 -2976 -2848 0
-1566 2975 2847 0
-1566 2975 1567 0
-1566 2847 1567 0
1566 -2975 -2847 0
1566 -2975 -1567 0
1566 -2847 -1567 0
        ⋮
-1538 2947 2819 0
-1538 2947 1539 0
-1538 2819 1539 0
1538 -2947 -2819 0
1538 -2947 -1539 0
1538 -2819 -1539 0
-1537 2946 2818 0
-1537 2946 1538 0
-1537 2818 1538 0
1537 -2946 -2818 0
1537 -2946 -1538 0
1537 -2818 -1538 0
```

Rysunek 4.5: Fragment pliku zawierającego formułę kodującą bit przeniesienia

Przechodząc do zapisu kodowania operacji **XOR dwóch słów** w koniunkcyjnej postaci normalnej otrzymano formułę, która jest równoważna tej opisującej funkcję XOR dwóch słów w poprzednim podrozdziale:

$$\bigwedge \begin{cases} \neg q_1 \vee u_1 \vee v_1 \\ q_1 \vee \neg u_1 \vee v_1 \\ q_1 \vee u_1 \vee \neg v_1 \\ \neg q_1 \vee \neg u_1 \vee \neg v_1 \\ \vdots \\ \neg q_{32} \vee u_{32} \vee v_{32} \\ q_{32} \vee \neg u_{32} \vee v_{32} \\ q_{32} \vee u_{32} \vee \neg v_{32} \\ \neg q_{32} \vee \neg u_{32} \vee \neg v_{32} \end{cases}$$

Operacja **rotacji o w bitów w lewo** występuje w algorytmie Salsa20 w kilku wariantach. Funkcja quarterround wykorzystuje przesunięcia o: 7, 8, 13 albo 19 bitów w słowie w lewą stronę. Zapis kodowania operacji rotacji o w bitów w lewo do formuły boolowskiej w koniunkcyjnej postaci normalnej poniżej przedstawiono dla $w = 7$

$$\bigwedge \begin{cases} \neg q_1 \vee u_8 \\ q_1 \vee \neg u_8 \\ \neg q_2 \vee u_9 \\ q_2 \vee \neg u_9 \\ \quad \vdots \\ \neg q_{25} \vee u_{32} \\ q_{25} \vee \neg u_{32} \\ \neg q_{26} \vee u_1 \\ q_{26} \vee \neg u_1 \\ \neg q_{27} \vee u_2 \\ q_{27} \vee \neg u_2 \\ \quad \vdots \\ \neg q_{32} \vee u_7 \\ q_{32} \vee \neg u_7. \end{cases}$$

Kodowanie boolowskie operacji XOR dwóch słów oraz rotacji o w bitów w lewo zostało wykorzystane do zapisu formuł logicznych kodujących działanie funkcji quarterround.

4.4.2. Funkcja quarterround

W przypadku przekształcenia formuły boolowskiej kodującej działanie funkcji quarterround do CNF użyliśmy dodatkowej zmiennej, która reprezentuje wynik operacji dodawania bitów przesunięty o w bitów w lewo. I tak, w przypadku rotacji o 7-bitów w lewo przyjmijmy oznaczenia: $q_{1_{i+7}} = \Big(\big(y_{0_{i+7}} \oplus y_{3_{i+7}}\big) \oplus c_{i+6}\Big)$ oraz $q_{1_{i-25}} = \Big(\big(y_{0_{i-25}} \oplus y_{3_{i-25}}\big) \oplus c_{i-26}\Big)$. Otrzymujemy wtedy:

$$\bigwedge \begin{cases} \bigwedge\limits_{i=1\ldots25} z_{1_i} \Leftrightarrow y_{1_i} \oplus q_{1_{i+7}} \\ \bigwedge\limits_{i=26\ldots32} z_{1_i} \Leftrightarrow y_{1_i} \oplus q_{1_{i-25}}. \end{cases}$$

W analogiczny sposób postępujemy w przypadku rotacji o 9, 13 i 18 bitów w lewo. Tak zamodelowane kodowanie funkcji quarterround zapisano następnie

w koniunkcyjnej postaci normalnej:

$$\bigwedge \begin{cases} \neg z_{1_1} \vee y_{1_1} \vee q_{1_8} \\ z_{1_1} \vee \neg y_{1_1} \vee q_{1_8} \\ z_{1_1} \vee y_{1_1} \vee \neg q_{1_8} \\ \neg z_{1_1} \vee \neg y_{1_1} \vee \neg q_{1_8} \\ \vdots \\ \neg z_{1_2 5} \vee y_{1_2 5} \vee q_{1_3 2} \\ z_{1_2 5} \vee \neg y_{1_2 5} \vee q_{1_3 2} \\ z_{1_2 5} \vee y_{1_2 5} \vee \neg q_{1_3 2} \\ \neg z_{1_2 5} \vee \neg y_{1_2 5} \vee \neg q_{1_3 2} \\ \neg z_{1_2 6} \vee y_{1_2 6} \vee q_{1_1} \\ z_{1_2 6} \vee \neg y_{1_2 6} \vee q_{1_1} \\ z_{1_2 6} \vee y_{1_2 6} \vee \neg q_{1_1} \\ \neg z_{1_2 6} \vee \neg y_{1_2 6} \vee \neg q_{1_1} \\ \vdots \\ \neg z_{1_3 2} \vee y_{1_3 2} \vee q_{1_7} \\ z_{1_3 2} \vee \neg y_{1_3 2} \vee q_{1_7} \\ z_{1_3 2} \vee y_{1_3 2} \vee \neg q_{1_7} \\ \neg z_{1_3 2} \vee \neg y_{1_3 2} \vee \neg q_{1_7}. \end{cases}$$

Funkcja quarterround jako dane wejściowe przyjmuje 4 słowa i zwraca ciąg 4 słów. Do zapisu w formacie DIMACS, zaprezentowanego w części 4.3.2. bezpośredniego kodowania boolowskiego jej działania, potrzeba 1024 klauzul. Rysunek 4.6 przedstawia fragment takiego zapisu.

```
-2657 1121 2152 0
2657 -1121 2152 0
2657 1121 -2152 0
-2657 -1121 -2152 0
-2658 1122 2153 0
2658 -1122 2153 0
2658 1122 -2153 0
-2658 -1122 -2153 0
        ⋮
-2688 1152 2151 0
2688 -1152 2151 0
2688 1152 -2151 0
-2688 -1152 -2151 0
```

Rysunek 4.6: Fragment pliku zawierającego formułę kodującą działanie funkcji quarterround

4.4.3. Funkcja littleendian

Zakodowane do postaci formuły logicznej działanie funkcji zmiany porządku bajtów w słowie (littleendian) po konwersji do CNF przyjmuje postać:

$$\bigwedge \begin{cases} \neg q_1 \vee u_{25} \\ q_1 \vee \neg u_{25} \\ \vdots \\ \neg q_8 \vee u_{32} \\ q_8 \vee \neg u_{32} \\ \neg q_9 \vee u_{17} \\ q_9 \vee \neg u_{17} \\ \vdots \\ \neg q_{16} \vee u_{24} \\ q_{16} \vee \neg u_{24} \\ \neg q_{17} \vee u_9 \\ q_{17} \vee \neg u_9 \\ \vdots \\ \neg q_{24} \vee u_{16} \\ q_{24} \vee \neg u_{16} \\ \neg q_{25} \vee u_1 \\ q_{25} \vee \neg u_1 \\ \vdots \\ \neg q_{32} \vee u_8 \\ q_{32} \vee \neg u_8 \end{cases}$$

Dobrym przykładem ilustrującym zapis kodowania boolowskiego działania funkcji littleendian, sprowadzony już do koniunkcyjnej postaci normalnej, jest fragment pliku zaprezentowany na rysunku 4.7, na którym można zauważyć część słowa stałego „expand 16-byte k", w tym przypadku „expa", który najpierw został zapisany w kodzie ASCII, by następnie zostać zapisanym w systemie binarnym. Tak powstały ciąg bitów poddano działaniu funkcji zmiany porządku bajtów w słowie. Efekty tego działania zapisane w formacie DIMACS przedstawiono na rysunku 4.7.

Wykorzystując opisaną powyżej metodę bezpośredniego kodowania otrzymano zbiór klauzul opisujących działanie całego szyfru Salsa20 ze 128- jak i 256-bitowym kluczem. Każdy pojedynczy bit jest reprezentowany tylko przez jedną

```
-1025 25 0
1025 -25 0
-1026 26 0
1026 -26 0
-1027 27 0
1027 -27 0
-1028 28 0
1028 -28 0
-1029 29 0
1029 -29 0
-1030 30 0
1030 -30 0
-1031 31 0
1031 -31 0
-1032 32 0
1032 -32 0
-1033 17 0
1033 -17 0
    ⋮
-1055 7 0
1055 -7 0
-1056 8 0
1056 -8 0
```

Rysunek 4.7: Fragment pliku zawierającego formułę kodującą ciąg znaków „expa" poddany działaniu littleendian

zmienną zdaniową, a liczby całkowite odpowiadające literałom dobrane są w taki sposób, aby opisać działanie algorytmu dla każdej rundy. W tabeli 4.1 przedstawiono wykaz zmiennych zdaniowych użytych do zakodowania jednej podwójnej rundy Salsa20, a także ilość zmiennych użytych do zapisu poszczególnych danych wejściowych i operacji.

Formuła kodująca działanie algorytmu Salsa20 dopuszczała użycie zarówno klucza 16- jak i 32-bajtowego. W tej pracy rozważania będą dotyczyć szyfru z kluczem 128-bitowym, a tabela 4.2 przedstawia liczbę użytych zmiennych i liczbę wygenerowanych klauzul, które zamodelowały działanie szyfru Salsa20 w każdej z 10 podwójnych rund.

Autorka przeprowadziła testy, które potwierdziły, że opracowana i przedstawiona w niniejszej pracy formuła kodująca działanie szyfru Salsa20 jest równoważna jego działaniu. Testy wykonywano zarówno podczas kodowania poszczególnych funkcji użytych w algorytmie Salsa20, jak i po zakodowaniu całego szy-

Fixed_word	$1, \ldots, 32, 33, \ldots, 64, 65, \ldots, 96, 97, \ldots, 128$	128
Key_128	$129, \ldots, 256$	128
Key_256	$257, \ldots, 384$	128
Nonce	$385, \ldots, 512$	128
Tekst jawny	$513, \ldots, 1024$	512
Column_word	$1025, \ldots, 1536$	512
Carry_word	$1537, \ldots, 2048$	512
Add_word	$2049, \ldots, 2560$	512
Rowround_word	$2561, \ldots, 3072$	512
RR_carry_word	$3073, \ldots, 3584$	512
RR_Add_word	$3585, \ldots, 4096$	512
Output_Rowround	$4097, \ldots, 4608$	512
Finish_carry_word	$4609, \ldots, 5120$	512
Finish_Add_word	$5121, \ldots, 5632$	512
Ciphertext_word	$5633, \ldots, 6144$	512

Tabela 4.1: Rejestr zmiennych zdaniowych dla formuły kodującej Salsa20/2

Liczba rund	Liczba użytych zmiennych	Liczba wygenerowanych klauzul
1	6144	29632
2	9216	47680
3	12288	65728
4	15360	83776
5	18432	101824
6	21504	119872
7	24576	137920
8	27648	155968
9	30720	174016
10	33792	192064

Tabela 4.2: Rezultaty otrzymane dla Salsa20 z 16-bajtowym kluczem

fru. Wartości wejściowe i wyjściowe na potrzeby tej weryfikacji zostały określone przy użyciu wektorów testowych podanych w pracy [12].

4.5. Badania eksperymentalne

Od czasu publikacji Salsa20 została poddana wielu analizom kryptograficznym [45]-[54]. Pierwszy atak przedstawił Paul Crowley w 2005 roku [45]. Zaproponował atak na szyfr Salsa20/5 o złożoności czasowej 2^{165}, oparty na różnicowej kryptoanalizie, za który dostał nagrodę Bernsteina w wysokości 1000 USD za „najciekawszą kryptoanalizę Salsa20".

W INDOCRYPT 2006 Fischer Meier, Berbain, Biasse i Robshaw skonstruowali atak odzyskiwania klucza na Salsa20/6 [61]. W SASC 2007, Tsnunoo wraz z Saito, Kubo, Suzaki i Nakashima przedstawili różnicową kryptoanalizę Salsa20/7 o złożoności czasowej 2^{190} [135]. W FSE 2008, Aumasson, Fischer, Khazaei, Meier i Rechberger przedstawili atak odzyskiwania klucza na Salsa20/8 o szacowanej złożoności czasowej 2^{251}, wykorzystując nową koncepcję z użyciem probabilistycznej techniki neutralnych bitów (PNB). Atak można dostosować do łamania Salsa20/7 z 128-bitowym kluczem [9]. Ten atak był ulepszony przez Shi i innych w ICISC 2012, który zmniejszył złożoność czasowa do 2^{250}, a przeciwko Salsa20/7 (klucz 128-bitowy) do złożoności czasowej 2^{109} [116].

Później Maitra i wsp., ponownie wykorzystując koncepcję PNB i kilka nowych pomysłów, skrócili złożoność czasową do $2^{247.2}$ dla Salsa20/8 [94]. Maitra poprawił złożoność ataku do 2245.5 [93], potem Choudhuri i Maitra poprawili go do $2^{244.9}$ [39]. Ostatnio Dey i wsp. podali nowy algorytm do konstrukcji PNB (Probabilistic Neutral Bits) i dalej ulepszyli złożoność do $2^{243.67}$ [49]. Poza tym w FSE 2008 Hernandez-Castro i in. wskazali na pewne słabości w podstawowej funkcji Salsa20 [70].W 2019 Ding przedstawia ulepszony atak odpowiednio powiązanych szyfrów Salsa20/12 i Salsa20/8 [54]. W międzyczasie, w 2013 roku Mouha i Preneel opublikowali dowód, że 15 rund Salsa20 z 128-bitowym kluczem jest zabezpieczone przed kryptoanalizą różnicową (w szczególności nie ma charakterystyki różnicowej z większym prawdopodobieństwem niż 2^{-130}, więc kryptoanaliza różnicowa byłaby trudniejsza niż wyczerpanie klucza 128-bitowego) [104]. Dla dowodu tej tezy autorzy artykułu użyli SAT-solvera, aby znaleźć charakterystyki różnicowe do pewnej wagi W.

W tej części przedstawiono wyniki eksperymentalne i próbę wyznaczenia marginesu bezpieczeństwa szyfru strumieniowego Salsa20 przy zastosowaniu SAT-solverów. Wszystkie wyniki eksperymentalne wykonano na maszynie o następujących parametrach: Architecture: x86_64, Little Endian, 4xCore - Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, Linux VirtualBox 4.15.0-88-generic Ubuntu GNU. Do pomiaru czasu wykorzystano wbudowane statystyki dla systemu Linux. Biorąc pod uwagę obliczeniową złożoność SAT, nie oczekiwano, że wszystkie instancje testowe zostaną przetworzone w rozsądnym czasie. Limit czasu ustalono na 24 godziny.

Przeprowadzając analizę kryptograficzną z wybranym tekstem jawnym szyfru Salsa20 ze 128-bitowym kluczem metodą bezpośredniego kodowania boolowskiego i wykorzystując przy tym wybrane narzędzia SAT, w pierwszej kolejności podjęto próbę złamania jednej podwójnej rundy, a zatem przeprowadzono atak z wybranym tekstem jawnym na Salsa20/2.

Do przeprowadzenia tego badania, jak i kolejnych, niezbędne było opracowanie narzędzia, które generuje formułę boolowską modelującą działanie szyfru Salsa20 zgodnie z zasadami przedstawionymi w rozdziale 4.3. dla poszczególnych rund szyfru. Istotnym było, by formuły te były generowane w formacie DIMACS i zapisywane do plików tekstowych, które następnie były przetwarzane przez SAT-solvery. Dodatkowo wykonano skrypty pomocnicze m.in.: generujące ciągi losowych bitów klucza i tekstu jawnego, które następnie są zapisywane formacie DIMACS w pliku tekstowym, konwertujące ciąg danych uzyskanych przez SAT-solver do postaci DIMACS.

Wyniki przeprowadzonego eksperymentu zawierają tabele: 4.3 i 4.4. Pierwsza z nich dotyczy wyników uzyskanych przez SAT-solvery jednowątkowe, druga przez SAT-solvery w których stosuje się algorytm wykorzystujący przetwarzanie równoległe.

SAT-solvery jednowątkowe	Time [s]
Minisat	0,028
Sat4J	0,453
PicoSAT	0,033
glu_vc	0,027

Tabela 4.3: Wyniki kryptoanalizy Salsa20/2

SAT-solvery wielowątkowe	Time [s]
PaInLeSS	1,58
Glucose Syrup	0,037
Plingeling	0,059

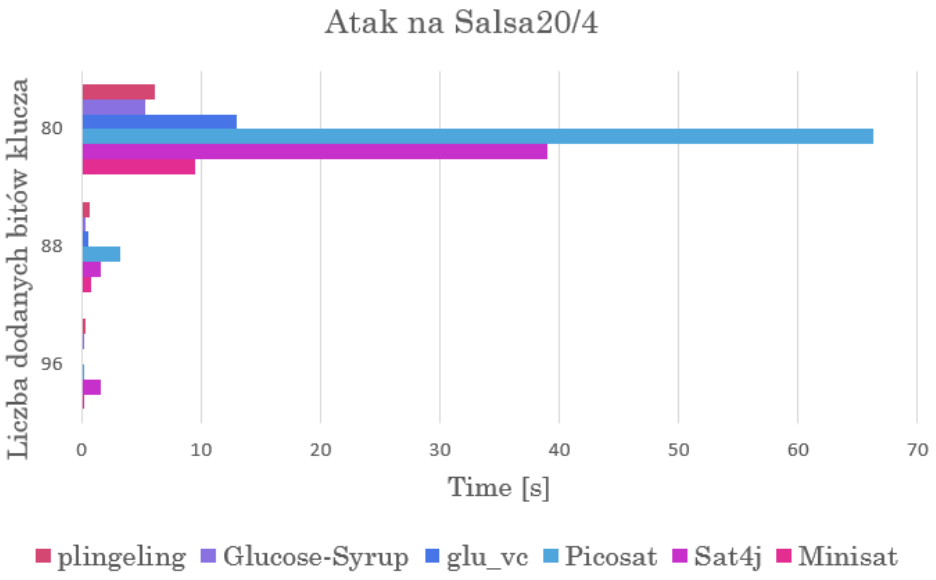
Tabela 4.4: Wyniki kryptoanalizy Salsa20/2 uzyskane przez SAT-solvery wielowątkowe

SAT-solvery działały zgodnie z oczekiwaniami i bez żadnych problemów poradziły sobie z odzyskaniem klucza w bardzo dobrym czasie.

Następny eksperyment polega na przeprowadzeniu ataku z wybranym tekstem jawnym na 4 rundy (2 podwójne rundy) szyfru Salsa20. W ustalonym na 24 godziny limicie czasu, żaden z wybranych SAT-solverów nie odzyskał klucza. Szukając granicy praktycznej obliczalności kryptoanalizy metodą SAT szyfru Salsa20/4 do wygenerowanych formuł dodano, w odpowiednim formacie, 96, 88, a następnie 80 początkowych bitów klucza i testowano, czy wybrane SAT-solvery obliczą pozostałe bity klucza.

		96 bitów	88 bitów	80 bitów
Minisat	Time [s]	0,251	0,767	9,55
Sat4j	Time [s]	1,63	1,65	39
Picosat	Time [s]	0,231	3,25	66,3
glu_vc	Time [s]	0,091	0,532	13
Glucose-Syrup	Time [s]	0,174	0,320	5,31
plingeling	Time [s]	0,311	0,707	6,13

Tabela 4.5: Atak na Salsa20/4 z dodanymi początkowymi bitami klucza cz. 1



Rysunek 4.8: Atak na Salsa20/4 z dodanymi początkowymi bitami klucza cz. 1

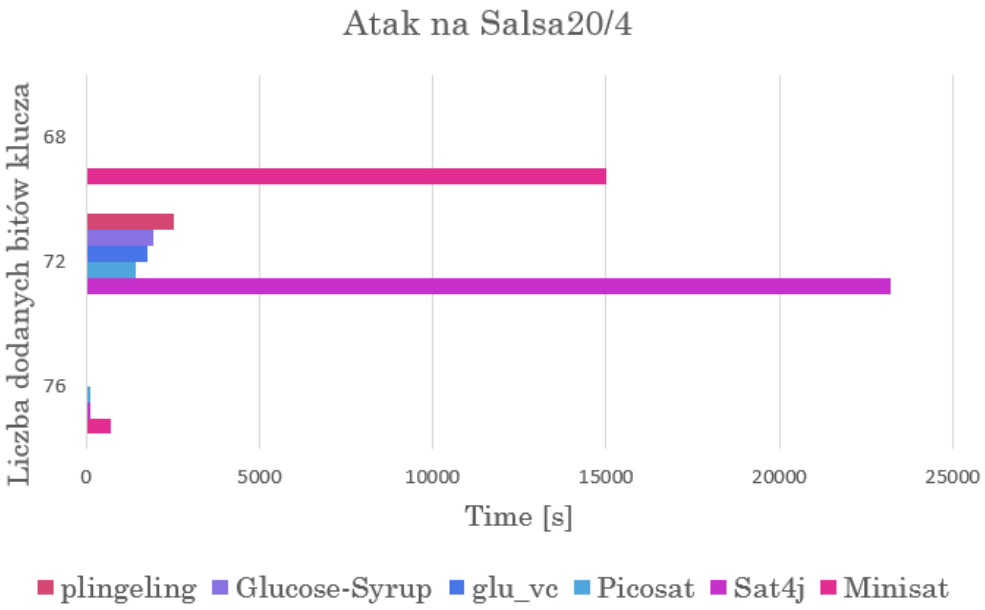
Przyglądając się tabeli 4.5 można zauważyć, że wszystkie SAT-solvery obliczyły brakujące bity klucza w rozsądnym czasie. Obliczenie 32 brakujących bitów klucza zajęło SAT-solverom maksymalnie nieco ponad 1,5 sekundy, 40 bi-

tów maksymalnie ponad 3 sekundy, a 48 bitów maksymalnie około jednej minuty. Rozkład uzyskanych wyników przedstawia rysunek 4.8.

Kontynuując poprzedni eksperyment zmniejszano liczbę dodanych bitów klucza kolejno do 76, 72 i 68 początkowych bitów klucza i przystąpiono do ataku na szyfr Salsa20/4 ze 128-bitowym kluczem (po przekroczeniu limitu czasu eksperymenty przerwano, co zostało oznaczone przez "–")

		76 bitów	72 bitów	68 bitów
Minisat	Time [s]	725	18	15000
Sat4j	Time [s]	110	23200	–
Picosat	Time [s]	132	1420	–
glu_vc	Time [s]	61,3	1790	–
Glucose-Syrup	Time [s]	15,3	1930	–
plingeling	Time [s]	45,5	2540	–

Tabela 4.6: Atak na Salsa20/4 z dodanymi początkowymi bitami klucza cz. 2



Rysunek 4.9: Atak na Salsa20/4 z dodanymi początkowymi bitami klucza cz. 2

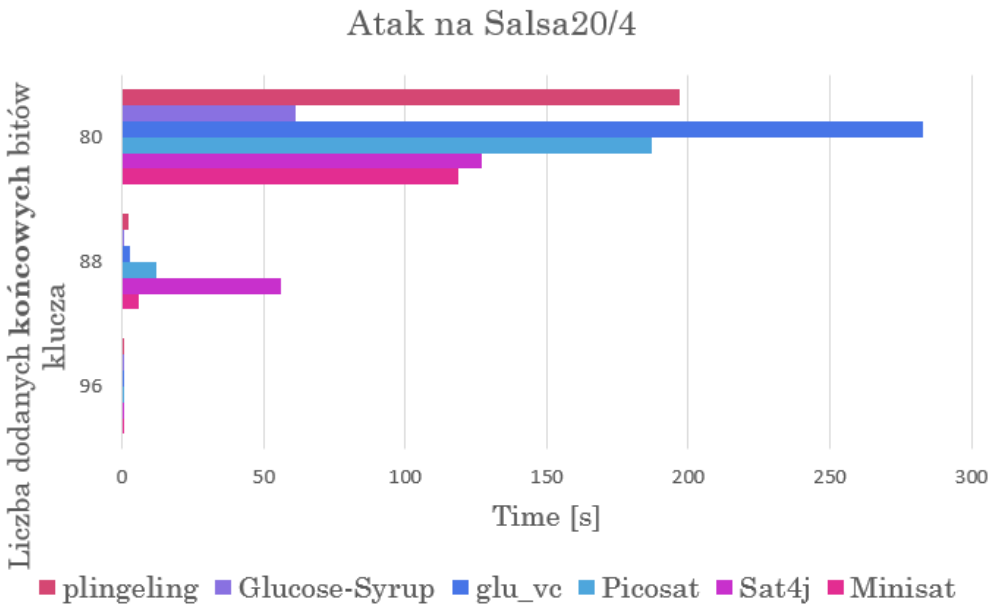
Czas potrzebny do obliczenia pozostałych bitów klucza znacznie się wydłużył w stosunku do poprzedniej części badania. Do obliczenia 52 końcowych bitów klucza SAT-solvery potrzebowały od nieco ponad 15 sekund (plingeling) do ponad 12 minut (Minisat), 56 końcowych bitów od ponad 18 sekund (Minisat) do

ponad 387 minut (SAT4j). Jedynym, spośród testowanych SAT-solverów, który obliczył końcowe 60 bitów klucza w ustalonym limicie czasu był Minisat. Dla pozostałych SAT-solverów czas 24 godzin nie był wystarczający do wykonania tych obliczeń. Wyniki tego eksperymentu zamieszczone są w tabeli 4.6, a graficznie przedstawia je rysunek 4.9 (brak słupka na wykresie oznacza, że dany SAT-solver nie złamał klucza w ustalonym czasie).

Dodatkowo przeprowadzono modyfikację powyższego eksperymentu dodając do formuły kodującej działanie szyfru odpowiednią liczbę końcowych, a nie jak poprzednio początkowych, bitów klucza.

		96 bitów	88 bitów	80 bitów
Minisat	Time [s]	0,101	5,90	119
Sat4j	Time [s]	0,642	56,3	127
Picosat	Time [s]	0,068	12,2	187
glu_vc	Time [s]	0,049	2,64	283
Glucose-Syrup	Time [s]	0,070	0,440	61,5
plingeling	Time [s]	0,165	2,38	197

Tabela 4.7: Atak na Salsa20/4 z dodanymi końcowymi bitami klucza cz.1



Rysunek 4.10: Atak Salsa20/4 z dodanymi końcowymi bitami klucza cz.1

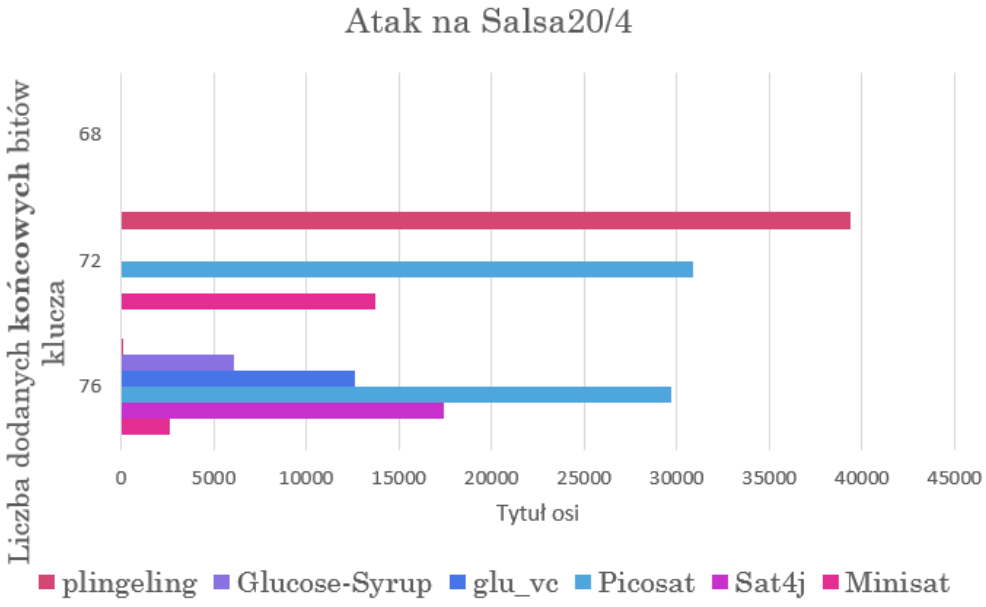
I tak, badając atak metodą opisaną w podrozdziale 2.8., z wybranym tek-

stem jawnym na Salsa20/4 przeprowadzono eksperyment, w którym do formuły kodującej działanie szyfru dodawano, w odpowiednim formacie, kolejno 96, 88, 80 końcowych bitów klucza, a następnie tak przygotowaną formułę wprowadzono do wybranych SAT-solverów w celu obliczenia początkowych bitów klucza. Wszystkie SAT-solvery, zgodnie z oczekiwaniami, poradziły sobie z odzyskaniem początkowych bitów klucza w rozsądnym czasie. Czasy uzyskane przez poszczególne SAT-solvery przedstawia tabela 4.7, a ich porównanie znajduje się na rysunku 4.10.

Autorka prowadząc dalej badanie, dodawała do formuły kodującej działanie szyfru Salsa20 kolejno coraz to mniejszą liczbę końcowych bitów klucza.

		76 bitów	72 bity	68 bitów
Minisat	Time [s]	2600	13700	–
Sat4j	Time [s]	17400	–	–
Picosat	Time [s]	29700	30900	–
glu_vc	Time [s]	12600	–	–
Glucose-Syrup	Time [s]	6110	–	–
plingeling	Time [s]	64,9	39400	–

Tabela 4.8: Atak na Salsa20/4 z dodanymi końcowymi bitami klucza cz.2



Rysunek 4.11: Atak na Salsa20/4 z dodanymi końcowymi bitami klucza cz.2

W przypadku dodania do formuły 76 końcowych bitów wszystkie SAT-solvery obliczyły początkowe 52 bity klucza, ale różnica w czasie jaki potrzebowały poszczególne SAT-solvery jest duża, a osiągnięte przez nie wyniki oscylują pomiędzy 65 sekund (plingeling), a ponad 8 godzin (Picosat). Wyniki tego eksperymentu przedstawia tabela 4.8 oraz rysunek 4.11. Warto też zauważyć, że w analogicznym badaniu z dodanymi 76 początkowymi bitami klucza czasy uzyskane przez SAT-solvery były znacznie krótsze.

Część z przedstawionych tutaj wyników została opublikowane w pracy [126].

4.6. Podsumowanie prac

Z przeprowadzonych testów wynika, że szyfr strumieniowy Salsa20 ze 128-bitowym kluczem zakodowany do formuły boolowskiej i poddany atakowi z wybranym tekstem jawnym przy użyciu narzędzi, jakimi są SAT-solvery, może zostać złamany w rozsądnym czasie w wersji zredukowanej do dwóch rund. Dodatkowo przy użyciu wybranych narzędzi SAT możliwe jest obliczenie brakujących bitów klucza w zredukowanym do 4 rund wariancie algorytmu Salsa20 (Salsa20/4) z dodanymi 68 początkowymi bitami w czasie krótszym niż 24 godziny.

Badawczy wkład własny autora rozprawy opisany w tym rozdziale jest następujący:

- opracowano bezpośrednie kodowanie boolowskie poszczególnych elementów składowych szyfru Salsa20, jak i całego szyfru dla każdej rundy,
- przeprowadzono testy, które potwierdziły, że opracowana i przedstawiona w niniejszej pracy formuła kodująca działanie szyfru Salsa20 jest równoważna jego działaniu,
- dokonano SAT-kryptoanalizy dla Salsa20/2, a także dla Salsa20/4, czyli czterech rund Salsa20 z dodanymi początkowymi, a także z dodanymi końcowymi bitami klucza,
- wykonano obliczenia mające na celu wyznaczenie granicy praktycznej obliczalności SAT-kryptoanalizy dla szyfru Salsa20 w obecnych realiach technicznych (software i moce obliczeniowe stosowanych maszyn).

Wszystkie prace programistyczne i obliczeniowe zostały samodzielnie wykonane przez autorkę rozprawy.

Rozdział 5.

SAT-kryptoanaliza AES

W tym rozdziale zostanie opisany sposób działania szyfru AES, a także wykorzystane w jego konstrukcji operacje matematyczne oraz przekształcenia używane przez algorytm szyfrujący. W dalszej części rozdziału zostanie przedstawione kodowanie poszczególnych elementów szyfru AES do formuł boolowskich, dzięki czemu kryptoanaliza AES zostanie sprowadzona do problemu SAT. W końcowej części rozdziału zostaną przedstawione wyniki uzyskane przez SAT-solvery, czyli narzędzia sprawdzające spełnialność formuł.

5.1. Advanced Encryption Standard

Aktualny standard szyfrowania Advanced Encryption Standard (AES) to symetryczny szyfr blokowy, który w 2001 roku został przyjęty przez NIST (National Institute of Standards and Technology) jako standard FIPS-197 [1, 47] w miejsce DES. Ponieważ siła algorytmu DES była niewystarczająca konieczne zatem stało się poszukiwanie nowego algorytmu szyfrowania zdolnego do ochrony wrażliwych informacji. W 1997 roku NIST ogłosił konkurs, w finale którego znalazło się pięć algorytmów szyfrujących: Rijndael, RC6, Mars, Serpent oraz Twofish. Zwycięzcą został ogłoszony Rijndael.

Rijndael jest rodziną szyfrów o różnych długościach klucza oraz różnych wielkościach bloków. Nazwa Rijndael pochodzi od nazwisk twórców szyfru, belgijskich kryptografów, Joana Daemena i Vincenta Rijmena. Dla AES, NIST wybrał trzy algorytmy z rodziny Rijndaela, z których każdy może przetwarzać bloki danych o długości 128 bitów przy użyciu klucza szyfrującego o długości odpowiednio 128, 192 lub 256 bitów, w zależności od wybranego algorytmu. Pomimo, że Rijndael został zaprojektowany do obsługi dodatkowych rozmiarów bloków i długości kluczy, to nie są one uwzględnione w standardzie szyfrowania AES.

5.2. Specyfikacja algorytmu

Wszystkie operacje we wnętrzu algorytmu AES są wykonywane na dwuwymiarowej tablicy bajtów zwanej *stanem wewnętrznym* lub krótko *stanem*. Stan składa się z czterech wierszy bajtów, a każdy wiersz z Nb bajtów, gdzie Nb oznacza długość bloku podzieloną przez 32. Dla standardu AES długość bloku Nb wynosi 4. Tablica *stanu* jest oznaczona literą s, a każdy znajdujący się w niej bajt ma dwa indeksy: r i c, gdzie r oznacza numer wiersza i $0 \leqslant r \leqslant 3$, a c wskazuje na numer kolumny i $0 \leqslant c \leqslant 3$. Zatem pojedynczy bajt *stanu* jest oznaczamy jako $s_{r,c}$ [1]. Tablicę *stanu wewnętrznego* AES ilustruje rysunek 5.1.

s

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

Rysunek 5.1: *Stan wewnętrzny* szyfru AES

Blok danych wejściowych $input_0 input_2 input_3 \ldots input_{127}$ długości 128 bitów zapisywany jest w tablicy bajtów w ten sposób, że:

$$\begin{aligned} in_0 &= input_0 input_1 \ldots input_7, \\ in_1 &= input_8 input_9 \ldots input_{15}, \\ &\vdots \\ in_{15} &= input_{120} input_{121} \ldots input_{127}, \end{aligned}$$

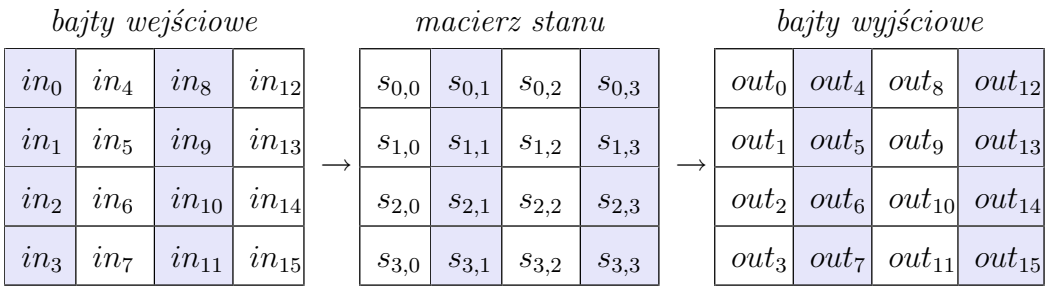
co obrazuje tabela 5.2.

in_0	in_4	in_8	in_{12}
in_1	in_5	in_9	in_{13}
in_2	in_6	in_{10}	in_{14}
in_3	in_7	in_{11}	in_{15}

Rysunek 5.2: Tablica bajtów wejściowych

Dłuższe sekwencje, jak na przykład klucze 192- i 256-bitowe umieszcza się w tablicy stanu według następującego wzoru:

$$in_n = input_{8n} input_{8n+1} \ldots input_{8n+7}.$$



Rysunek 5.3: Tablice stanu szyfru AES

Jak pokazuje rysunek 5.3 rozpoczynając szyfrowanie lub deszyfrowanie algorytm kopiuje dane wejściowe – tablicę bajtów $in_0, in_1, \ldots, in_{15}$ do tablicy stanu zgodnie ze schematem:

$$s_{r,c} = in_{r+4c}, \quad dla \quad 0 \leqslant r \leqslant 3 \quad i \quad 0 \leqslant c \leqslant 3.$$

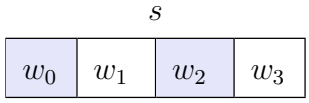
Następnie na tej tablicy stanu wykonuje operacje szyfrowania lub deszyfrowania, by ostatecznie skopiować jej końcową wartość do tablicy bajtów wyjściowych w następujący sposób:

$$out_{r+4c} = s_{r,c}, \quad dla \quad 0 \leqslant r \leqslant 3 \quad i \quad 0 \leqslant c \leqslant 3.$$

O tablicy stanu można też myśleć jak o jednowymiarowej macierzy 32-bitowych słów (kolumn), gdzie numer kolumny c jest jednocześnie indeksem do tej macierzy. Wówczas w tablicy stanu każda kolumna składa się z czterech bajtów tworząc 32-bitowe słowa, a numer wiersza r jest indeksem dla czterech bajtów w każdym słowie. Stan można traktować jako tablicę czterech słów zapisanych następująco:

$$w_0 = s_{0,0}s_{1,0}s_{2,0}s_{3,0} \quad w_2 = s_{0,2}s_{1,2}s_{2,2}s_{3,2}$$
$$w_1 = s_{0,1}s_{1,1}s_{2,1}s_{3,1} \quad w_3 = s_{0,3}s_{1,3}s_{2,3}s_{3,3}$$

Tak rozumiany stan wewnętrzny AES widzimy na rysunku 5.4.



Rysunek 5.4: Jednowymiarowa tablica stanu szyfru AES

Każde wejście i wyjście algorytmu AES składa się z sekwencji 128 bitów (cyfr o wartości 0 lub 1). Sekwencje te są określane jako bloki, a liczba bitów, które zawierają, określana jest jako długość bloku. Fakt ten zapisuje się następująco: $Nb = 4$, co oznacza liczbę 32-bitowych słów (liczbę kolumn) w stanie.

Klucz szyfrujący algorytmu AES to sekwencja 128, 192 lub 256 bitów. Inne długości klucza nie są dozwolone przez ten standard. Długość klucza jest oznaczana przez $Nk = 4, 6$ lub 8, co odzwierciedla liczbę 32-bitowych słów (liczbę kolumn) w kluczu szyfrującym.

W przypadku AES liczba rund do wykonania przez algorytm zależy od rozmiaru klucza. Liczba rund jest oznaczona przez Nr, gdzie $Nr = 10$, gdy $Nk = 4$, $Nr = 12$, gdy klucz szyfrujący ma długość sześciu 32-bitowych słów, i $Nr = 14$, gdy długość klucza wynosi 256-bitów.

Jedyne kombinacje tych parametrów, które są zgodne ze standardem szyfrowania AES, przedstawia tabela 5.1.

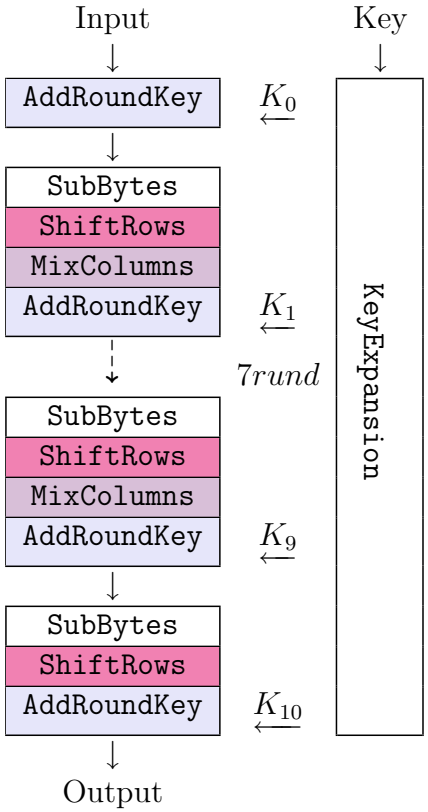
	Długość klucza (Nk **słów**)	**Długość bloku** (Nb **słów**)	**Liczba rund** (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Tabela 5.1: Możliwe kombinacje parametrów: Nk, Nb, Nr.

Algorytm AES korzysta ze struktury sieci podstawieniowo-permutacyjnej. Sieci podstawieniowo-permutacyjne (ang. substitution-permutate networks (SPN)) to struktury występujące w szyfrach blokowych, w których podstawianie pojawia się często w postaci S-boxów, które są małymi tablicami wyszukiwania przekształcającymi ciągi o długości 4 lub 8 bitów. W niektórych szyfrach wykorzystuje się podstawową algebrę liniową i mnożenie macierzy po to by zapewnić silną dyfuzję, czyli taki stan, w którym wynik tych przekształceń w równym stopniu zależy od każdego bitu na wejściu. Sieć SPN w realizacji bywa podobna do sieci Feistela, jednak użyte w niej S-boxy muszą być odwracalne, podczas gdy funkcja użyta w sieci Feistela odwracalna być nie musi.

Wykorzystywana przez algorytm AES sieć podstawieniowo-permutacyjna występuje w kilku wariantach w zależności od długości klucza: z 10 rundami dla kluczy 128-bitowych, 12 rundami dla kluczy 192-bitowych oraz 14 dla 256-bitowych, gdzie za podstawienie odpowiada przekształcenie `SubBytes()`, a za permutację połączenie przekształceń `ShiftRows()` i `MixColumns()`.

Rysunek 5.5 przedstawia SPN w wersji 10 rund z 128-bitowym kluczem x. Zarówno do szyfrowania, jak i deszyfrowania algorytm AES wykorzystuje funkcję rundową, która składa się z czterech różnych przekształceń dokonywanych na bajtach tablicy stanu: podstawienie bajtów przy użyciu tablicy podstawień (S-box), przesunięcie wierszy tablicy stanów, mieszanie danych w każdej kolumnie tablicy stanu, oraz dodawanie klucza rundowego do stanu.



Rysunek 5.5: Schemat działania szyfru AES dla 128-bitowego klucza

5.2.1. Podstawowe operacje matematyczne

Operacje wewnętrzne algorytmu AES są wykonywane na bajtach. Bajt b zapisany jako ciąg ośmiu bitów $b = (b_7, b_6, b_5, b_4, b_5, b_4, b_3, b_2, b_1, b_0)$ jest pojedynczą jednostką przetwarzania i jest traktowany przez AES jako element ciała skończonego $GF(256)$ i zapisywany za pomocą reprezentacji wielomianowej:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^{7} b_i x^i.$$

Na przykład bajt `01011010` jest reprezentowany przez wielomian: $x^6+x^4+x^3+x^1$. Wartości bajtów mogą być przedstawione za pomocą notacji szesnastkowej i używając oznaczenia `0x` powyższy bajt można zapisać jako `0x5a`. Na elementach ciała skończonego $GF(256)$ wykonuje się operacje dodawania i mnożenia, ale różnią się one od tych, które stosuje się do liczb.

Dodawanie w ciele $GF(256)$. Operacja dodawania dwóch elementów z ciała $GF(256)$ polega na xorowaniu tj. dodawaniu do siebie modulo 2 współczynników wielomianów występujących przy odpowiednich potęgach. Zatem operację dodawania dwóch wielomianów $a(x) = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ i $b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$ z ciała skończonego $GF(256)$ można zapisać następująco:

$$c(x) = a(x) + b(x) = c_7x^7 + c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0,$$

gdzie $c_i = a_i \oplus b_i$, dla $0 \leqslant i \leqslant 7$.

Ponieważ operacje algorytmu AES wykonywane są na bajtach, to zapisane powyżej działanie można opisać jako dodawanie modulo 2 odpowiednich bitów w bajcie. Dla dwóch bajtów $a = (a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0)$ i $b = (b_7, b_6, b_5, b_4, b_5, b_4, b_3, b_2, b_1, b_0)$ suma wynosi:

$$c = (c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0),$$

gdzie każde $c_i = a_i \oplus b_i$, dla $0 \leqslant i \leqslant 7$.

Na przykład niech $a(x) = x^7 + x^6 + x^4 + x^2 + 1$ i $b(x) = x^6 + x^5 + x + 1$ wówczas: używając reprezentacji wielomianowej otrzymujemy:

$$(x^7 + x^6 + x^4 + x^2 + 1) + (x^6 + x^5 + x + 1) = x^7 + x^5 + x^4 + x^2 + x,$$

stosując zapis binarny mamy:

$$\texttt{11010101} \oplus \texttt{01100011} = \texttt{10110110},$$

w notacji szesnastkowej uzyskujemy:

$$\texttt{0xd5} \oplus \texttt{0x63} = \texttt{0xb6}.$$

Wszystkie powyższe zapisy są równoważne.

Zastosowane w AES **mnożenie w ciele** $GF(256)$**.** Stosując reprezentację wielomianową mnożenie w ciele $GF(256)$ jest oznaczone przez $\bullet$ i polega na mnożeniu wielomianów modulo pewien nierozkładalny wielomian stopnia 8. Wielomian nazywany jest nierozkładalnym, jeśli jego jedynymi dzielnikami są jeden i on sam. Wybrany dla algorytmu AES nierozkładalny wielomian to:

$$m(x) = x^8 + x^4 + x^3 + x + 1,$$

a zapisany w notacji szesnastkowej jest postaci `0x011b`.

Na przykład $\texttt{0xad} \bullet \texttt{0x43} = \texttt{0x14}$, gdyż

$$\begin{aligned}(x^7 + x^5 + x^3 + x^2 + 1)(x^6 + x + 1) \;=\; & x^{13} + x^{11} + x^9 + x^8 + x^6 + \\ & + x^8 + x^6 + x^4 + x^3 + x + \\ & + x^7 + x^5 + x^3 + x^2 + 1 = \\ = \; & x^{13} + x^{11} + x^9 + x^7 + x^5 + x^4 + x^2 + x + 1\end{aligned}$$

i

$$\begin{aligned}x^{13} + x^{11} + x^9 + x^7 + x^5 + x^4 + x^2 + x + 1 \text{ modulo}(x^8 + x^4 + x^3 + x + 1) = \\ = x^4 + x^2.\end{aligned}$$

Stąd, używając notacji binarnej, mamy: $\texttt{10101101} \bullet \texttt{0100011} = \texttt{00010100}$.

Redukcja modularna o $m(x)$ zapewnia, że wynik działania $\bullet$ będzie wielomianem binarnym o stopniu mniejszym niż 8, a dzięki temu może być reprezentowany przez bajt. W przeciwieństwie do dodawania, nie ma prostej operacji na poziomie bajtów, która odpowiada temu mnożeniu.

Własności. Zdefiniowane powyżej działanie $\bullet$ jest łączne, a element `0x01` jest elementem neutralnym względem tego działania. Niech $b(x)$ będzie dowolnym niezerowym wielomianem stopnia niższego niż 8. Element odwrotny do elementu $b(x)$ oznaczamy $b^{-1}(x)$ do jego wyznaczenia używa się rozszerzonego algorytmu Euklidesa dla wielomianów, który służy do wyznaczenia wielomianów $a(x)$ i $c(x)$ takich, że

$$b(x)a(x) + m(x)c(x) = 1.$$

Stąd $a(x) \bullet b(x) \bmod m(x) = 1$, a zatem

$$b^{-1}(x) = a(x) \bmod m(x).$$

Ponadto tak zdefiniowane działanie mnożenia $\bullet$ elementów w ciele $GF(256)$ jest rozdzielne względem wyżej określonego działania dodawania $+$ elementów w ciele $GF(256)$ tzn. dla dowolnych $a(x)$, $b(x)$ i $c(x)$ zachodzi:

$$a(x) \bullet (b(x) + c(x)) = a(x) \bullet b(x) + a(x) \bullet c(x).$$

Z powyższych definicji wynika, że zbiór 256 możliwych wartości bajtowych, z działaniami dodawania $+$ i mnożenia $\bullet$, ma strukturę ciała skończonego $GF(256)$.

Mnożenie przez x w ciele $GF(256)$. Rozpatrując przypadek mnożenia $\bullet$ dowolnego wielomianu $b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$ z ciała $GF(256)$ przez wielomian x mamy:

$$b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x.$$

Wynik działania $x \bullet b(x)$ otrzymujemy wykonując redukcję powyższego wielomianu modulo $m(x) = x^8 + x^4 + x^3 + x + 1$.

$$x \bullet b(x) = xb(x) \bmod m(x) =$$

$$= b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x \bmod m(x).$$

W przypadku, gdy $b_7 = 0$, to wynik działania $x \bullet b(x)$ jest już w zredukowanej formie tzn.:

$$x \bullet b(x) = xb(x) =$$

$$= b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x.$$

W przeciwnym wypadku redukcja jest wykonywana przez odjęcie (sksorowanie) wielomianu $m(x)$, zatem

$$x \bullet b(x) = xb(x) \oplus m(x) = (x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 +$$

$$+b_0x) \oplus (x^8 + x^4 + x^3 + x + 1) = b_6x^7 + b_5x^6 + b_4x^5 + (b_3 \oplus 1)x^4 +$$

$$+(b_2 \oplus 1)x^3 + b_1x^2 + (b_0 \oplus 1)x + 1.$$

Na poziomie bajtów, mnożenie $\bullet$ przez x tzn. przez `00000010` (lub inaczej przez `0x02`) może być zaimplementowane jako przesunięcie w lewo, a następnie warunkowe bitowe wykonanie operacji XOR z `0x1b`. Ta wyżej opisana operacja na

bajtach jest oznaczana przez $xtime()$. Aby zaimplementować mnożenie przez wyższe potęgi x wystarczy wielokrotnie zastosować funkcję $xtime()$. Zatem – dodając pośrednie wyniki – możliwe jest zaimplementowanie mnożenia $\bullet$ przez dowolną stałą.

Na przykład $\texttt{0xad} \bullet \texttt{0x43} = \texttt{0x14}$, gdyż:

$$\texttt{0xad} \bullet \texttt{0x02} = xtime(\texttt{0xad}) = \texttt{0x41}$$

$$\texttt{0xad} \bullet \texttt{0x04} = xtime(\texttt{0x41}) = \texttt{0x82}$$

$$\texttt{0xad} \bullet \texttt{0x08} = xtime(\texttt{0x82}) = \texttt{0x1f}$$

$$\texttt{0xad} \bullet \texttt{0x10} = xtime(\texttt{0x1f}) = \texttt{0x3e}$$

$$\texttt{0xad} \bullet \texttt{0x20} = xtime(\texttt{0x3e}) = \texttt{0x79}$$

$$\texttt{0xad} \bullet \texttt{0x40} = xtime(\texttt{0x79}) = \texttt{0xf8},$$

stąd

$$\texttt{0xad} \bullet \texttt{0x43} = \texttt{0xad} \bullet (\texttt{0x01} \oplus \texttt{0x02} \oplus \texttt{0x40}) =$$

$$= \texttt{0xad} \oplus \texttt{0x41} \oplus \texttt{0xf8} = \texttt{0x14}.$$

Stosując notację binarną do powyższego przykładu otrzymujemy:

$$\texttt{10101101} \bullet \texttt{00000010} = \texttt{01011010} \oplus \texttt{00011011} = \texttt{01000001}$$

$$\texttt{10101101} \bullet \texttt{01000001} = \texttt{10000010}$$

$$\texttt{10101101} \bullet \texttt{10000010} = \texttt{00000100} \oplus \texttt{00011011} = \texttt{00011111}$$

$$\texttt{10101101} \bullet \texttt{00011111} = \texttt{00111110}$$

$$\texttt{10101101} \bullet \texttt{00111110} = \texttt{01111100}$$

$$\texttt{10101101} \bullet \texttt{01111100} = \texttt{11111000},$$

zatem

$$\texttt{10101101} \bullet \texttt{01000011} = \texttt{10101101} \bullet (\texttt{00000001} \oplus \texttt{00000010} \oplus \texttt{01000000}) =$$

$$= \texttt{10101101} \oplus \texttt{01000001} \oplus \texttt{11111000} = \texttt{00010100}.$$

Wielomiany ze współczynnikami w $GF(256)$. Kolejnymi operacjami matematycznymi zastosowanymi w algorytmie AES, są operacje na wielomianach stopnia co najwyżej trzeciego i współczynnikach z ciała $GF(256)$ zdefiniowanych następująco:

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0,$$

gdzie $a_i \in GF(256)$ dla $0 \leqslant i \leqslant 3$, które traktuje się jak 4-bajtowe słowo (a_0, a_1, a_2, a_3). Warto zauważyć, że w powyżej zdefiniowanym wielomianie współczynniki są same w sobie elementami z ciała $GF(256)$ tj. bajtami nie zaś, jak poprzednio rozpatrywaliśmy, bitami.

Jeżeli $b(x) = b_3x^3 + b_2x^2 + b_1x + b_0$ i $b_i \in GF(256)$ dlb $0 \leqslant i \leqslant 3$, to

$$a(x) + b(x) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_0 \oplus b_0).$$

Zatem dodawanie odbywa sie poprzez dodanie współczynników przy odpowiednich potęgach x, co odpowiada operacji xorowania odpowiednich bajtów z każdego słowa.

Mnożenie tak określonych wielomianów przebiega dwuetapowo. W pierwszym kroku rozwija się algebraicznie iloczyn $c(x) = a(x) \bullet b(x)$:

$$\begin{aligned}
c(x) &= (a_3x^3 + a_2x^2 + a_1x + a_0) \bullet (b_3x^3 + b_2x^2 + b_1x + b_0) = \\
&= a_3x^3 \bullet b_3x^3 + a_3x^3 \bullet b_2x^2 + a_3x^3 \bullet b_1x + a_3x^3 \bullet b_0 + \\
&\quad + a_2x^2 \bullet b_3x^3 + a_2x^2 \bullet b_2x^2 + a_2x^2 \bullet b_1x + a_2x^2 \bullet b_0 + \\
&\quad + a_1x \bullet b_3x^3 + a_1x \bullet b_2x^2 + a_1x \bullet b_1x + a_1x \bullet b_0 + \\
&\quad + a_0 \bullet b_3x^3 + a_0 \bullet b_2x^2 + a_0 \bullet b_1x + a_0 \bullet b_0 = \\
&= (a_3 \bullet b_3)x^6 + (a_3 \bullet b_2 \oplus a_2 \bullet b_3)x^5 + (a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3)x^4 + \\
&\quad + (a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3)x^3 + (a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2)x^2 + \\
&\quad + (a_1 \bullet b_0 \oplus a_0 \bullet b_1)x + (a_0 \oplus b_0).
\end{aligned}$$

Ponieważ $c(x)$ nie przedstawia słowa 4-bajtowego, to w drugim kroku mnożenia redukuje się ten wielomian modulo wielomian stopnia 4. Wówczas otrzymujemy wielomian stopnia co najwyżej trzeciego. Do tego celu algorytm AES wykorzystuje wielomian $x^4 + 1$, stąd

$$x^i \bmod (x^4 + 1) = x^{i \bmod 4}.$$

Wynikiem mnożenia wielomianów $a(x)$ i $b(x)$ modulo wielomian $x^4 + 1$, oznaczonego przez $a(x) \otimes b(x)$, jest wielomian stopnia trzeciego $d(x)$ taki, że

$$d(x) = d_3x^3 + d_2x^2 + d_1x + d_0,$$

przy czym

$$d_0 = (a_0 \bullet b_0)(a_3 \bullet b_1) \oplus (a_2 \bullet b_2) \oplus (a_1 \bullet b_3),$$

$$d_1 = (a_1 \bullet b_0) \oplus (a_0 \bullet b_1) \oplus (a_3 \bullet b_2) \oplus (a_2 \bullet b_3),$$

$$d_2 = (a_2 \bullet b_0) \oplus (a_1 \bullet b_1) \oplus (a_0 \bullet b_2) \oplus (a_3 \bullet b_3),$$

$$d_3 = (a_3 \bullet b_0) \oplus (a_2 \bullet b_1) \oplus (a_1 \bullet b_2) \oplus (a_0 \bullet b_3).$$

Jeżeli $a(x)$ i $b(x)$ są ustalonymi wielomianami stopnia co najwyżej trzeciego o współczynnikach w $GF(256)$, to mnożenie tych wielomianów nad $GF(256)$ modulo wielomian $x^4 + 1$, czyli operację $d(x) = a(x) \otimes b(x)$, można zapisać w postaci macierzowej jako:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Ponieważ wielomian $x^4 + 1$ jest wielomianem nierozkładalnym w ciele $GF(256)$, więc operacja $\otimes$ nie jest zawsze odwracalna. Dlatego też, do operacji `MixColumns()` i operacji odwrotnej do niej, AES używa następującej pary wielomianów, z których jeden jest odwrotnością drugiego:

$$a(x) = \texttt{0x03}x^3 + \texttt{0x01}x^2 + \texttt{0x01}x + \texttt{0x02}$$

$$a^{-1}(x) = \texttt{0x0b}x^3 + \texttt{0x0d}x^2 + \texttt{0x09}x + \texttt{0x0e}.$$

Dodatkowo, w funkcji `RotWord()`, algorytm AES stosuje wielomian $a(x) = \texttt{0x0}x^3 + \texttt{0x00}x^2 + \texttt{0x00}x + \texttt{0x00} = x^3$. Wynikiem działania $b(x) \otimes x^3$, gdzie $b = (b_0, b_1, b_2, b_3)$ jest słowem, jest 4-bajtowe słowo (b_1, b_2, b_3, b_0), co odpowiada rotacji bajtów w danym słowie, a dokładniej przesunięciu bajtów słowa wejściowego o jedno miejsce w lewo [1].

5.2.2. Przekształcenie `SubBytes()`

Funkcja `SubBytes()` jest nieliniowym przekształceniem, które działa niezależnie na każdym bajcie stanu za pomocą tablicy podstawień (S-box). S-box, zapisany w systemie szesnastkowym, użyty w przekształceniu `SubBytes()` przedstawia tabela 5.2 [1].

		y															
HEX		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Tabela 5.2: S-box: wartości wyjściowe dla bajtu xy.

Ten S-box jest zbudowany na podstawie dwóch przekształceń: odwrotności multiplikatywnej w ciele skończonym GF(256), przyjmując dodatkowo, że odwrotnością multiplikatywną elementu `0x00` jest on sam (jest to element samoodwrotny) oraz następującego przekształcenia afinicznego nad GF(2):

$$b'_i = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i$$

dla $0 \leqslant i \leqslant 7$, gdzie b_i jest i-tym bitem bajtu, a c_i jest i-tym bitem bajtu c, którego wartość wynosi `0x63` lub `01100011`.

W postaci macierzowej element transformacji afinicznej S-boxa można wyrazić

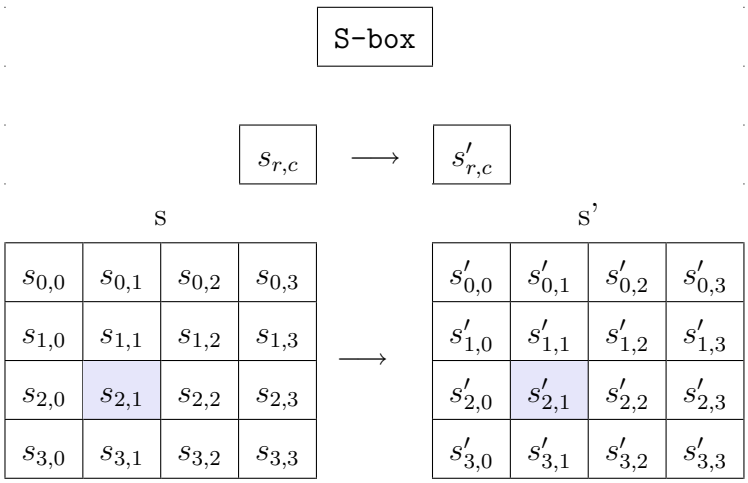
jako:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Zatem przekształcenie to polega na wyznaczeniu bajtu $b' = (b'_7, b'_6, b'_5, b'_4, b'_5, b'_4, b'_3, b'_2, b'_1, b'_0)$, którego odpowiednie bity oblicza się przy pomocy następujących równań nad $GF(2)$:

$$\begin{aligned} b'_0 &= b_7 + b_6 + b_5 + b_4 + b_0 + 1 \\ b'_1 &= b_7 + b_6 + b_5 + b_1 + b_0 + 1 \\ b'_2 &= b_7 + b_6 + b_2 + b_1 + b_0 \\ b'_3 &= b_7 + b_3 + b_2 + b_1 + b_0 \\ b'_4 &= b_4 + b_3 + b_2 + b_1 + b_0 \\ b'_5 &= b_5 + b_4 + b_3 + b_2 + b_1 + 1 \\ b'_6 &= b_6 + b_5 + b_4 + b_3 + b_2 + 1 \\ b'_7 &= b_7 + b_6 + b_5 + b_4 + b_3. \end{aligned}$$

Rysunek 5.6 przedstawia wpływ przekształcenia `SubBytes()` na tablicę stanu wewnętrznego szyfru AES.



Rysunek 5.6: Przekształcenie `SubBytes()`

Warto zauważyć, że S-box użyty w algorytmie AES jest odwracalny. Przykład działania podstawienia `S-box` przedstawiono w dodatku B1.

5.2.3. Przekształcenie `ShiftRows()`

Przekształcenie `ShiftRows()` przesuwa cyklicznie bajty w lewo w trzech ostatnich wierszach stanu odpowiednio o jedno, dwa i trzy miejsca. Pierwszy wiersz nie ulega zmianie. Tablica stanu jest modyfikowana zgodnie z równaniem:

$$s'_{r,c} = s_{r,(c+shift(r,Nb))modNb} \quad dla \quad 0 < r < 4 \quad i \quad 0 \leqslant c \leqslant Nb,$$

gdzie wartość przesunięcia $shift(r, Nb)$ zależy od numeru wiersza r i dla $Nb = 4$ wynosi:

$$shift(0,4) = 0; \quad shift(1,4) = 1; \quad shift(2,4) = 2; \quad shift(3,4) = 3.$$

Rysunek 5.3 przedstawia tablicę Stanu po przekształceniu `ShiftRows()`.

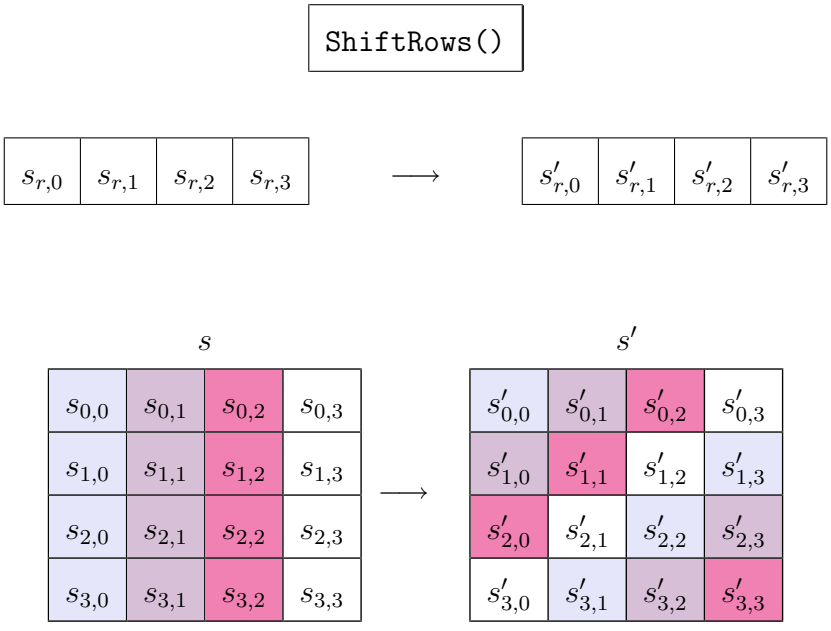


Tabela 5.3: Przekształcenie `ShiftRows()`

5.2.4. Przekształcenie `MixColumns()`

Przekształcenie `MixColumns()` pobiera wszystkie kolumny stanu i miesza ich dane (niezależnie od siebie) w celu utworzenia nowych kolumn wykonując następujące działania:

$$
\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c})
\end{aligned}
$$

Rysunek 5.4 przedstawia przekształcenie tablicy stanu, kolumna po kolumnie, przez procedurę `MixColumns()`.

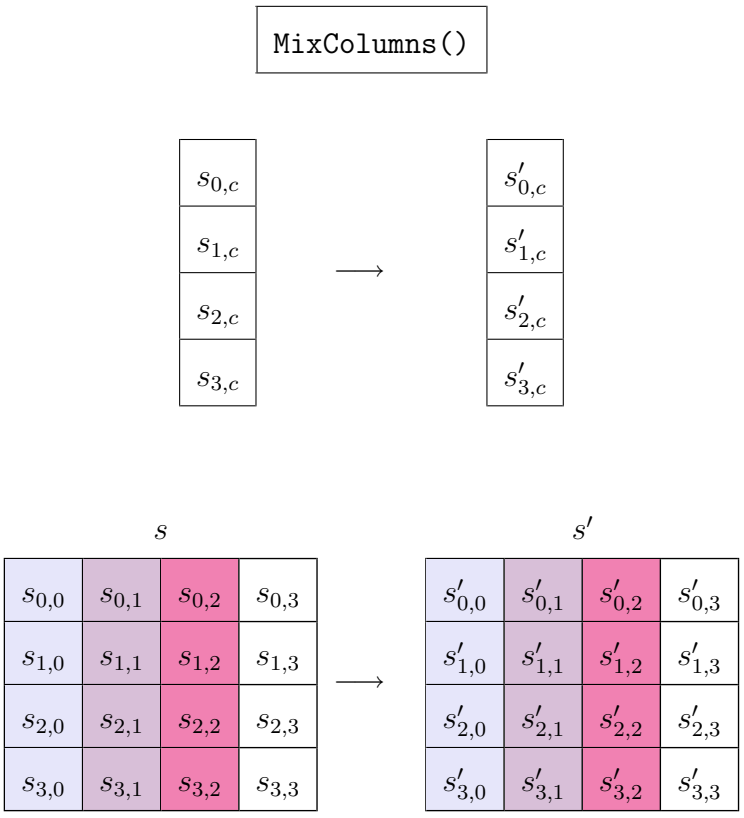


Tabela 5.4: Przekształcenie `MixColumns()`

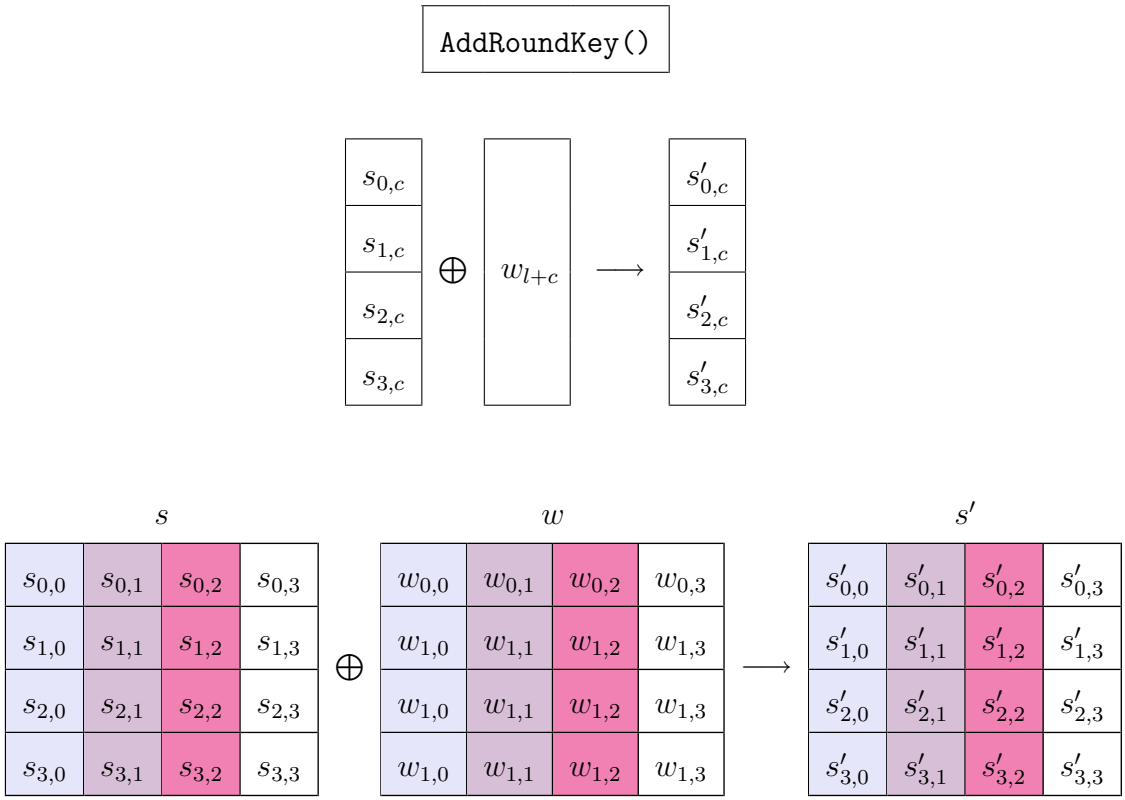
5.2.5. Przekształcenie `AddRoundKey()`

Działanie procedury `AddRoundKey()` oparte jest na prostej bitowej operacji XOR. Przekształcenie to dodaje modulo dwa klucz rundowy do tablicy stanu. Każdy klucz rundowy jest złożony z Nb słów umieszczonych w harmonogramie kluczy, którego tworzenie zostanie opisane w podrozdziale 5.2.6. Słowa te są dodawane do kolumn stanu według wzoru:

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{round*Nb+c}] \quad dla \quad 0 \leqslant c \leqslant Nb,$$

gdzie $[w_i]$ oznaczają słowa klucza zawarte w harmonogramie kluczy, a $round$ przyjmuje wartość z przedziału $0 \leqslant round \leqslant Nr$. Wywołanie przekształcenia `AddRoundKey()` następuje przed pierwszym zastosowaniem funkcji rundowej - klucz inicjujący zostaje sksorowany z tablicą stanu, gdy $round = 0$. Procedura `AddRoundKey()` jest zastosowana w Nr rundach szyfru AES, gdy $1 \leqslant round \leqslant Nr$ [1].

Działanie tego przekształcenia ilustruje rysunek 5.7, gdzie $l = round * Nb$.



Rysunek 5.7: Przekształcenie `AddRoundKey()`

5.2.6. Rozszerzenie klucza

Jednym z parametrów funkcji rundowej wykorzystywanej przez algorytm AES jest zestaw kluczy rundowych, który składa się z jednowymiarowej tablicy czterobajtowych słów utworzonych przy użyciu procedury Key Expansion. Algorytm pobiera klucz szyfru K i wykonując procedurę rozszerzenia klucza generuje w sumie $Nb(Nr + 1)$ słów. Algorytm wymaga początkowego zestawu Nb słów, a każda z Nr rund wymaga Nb słów klucza. Uzyskany zestaw kluczy składa się z liniowej tablicy 4-bajtowych słów, oznaczonych $[w_i]$, gdzie $i \in [0, Nb(Nr+1))$. W przypadku klucza 128-bitowego tablica ta ma postać:

$$[w_0, w_1, ..., w_{43}].$$

Rozszerzenie klucza wejściowego do zestawu kluczy przebiega według następującego pseudokodu:

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp

    i = 0

    while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
    end while

    i = Nk

    while (i < Nb * (Nr+1)]
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
    end while
end
```

Funkcja `SubWord()` jako argument przyjmuje czterobajtowe słowo, stosuje S-box do każdego z czterech bajtów i zwraca słowo wyjściowe. `RotWord()` to funkcja, która jako dane wejściowe przyjmuje słowo $[a_0, a_1, a_2, a_3]$, wykonuje cykliczną permutację wynikiem której jest słowo $[a_1, a_2, a_3, a_0]$. Tak zwana stała rundy, `Rcon[i]`, jest tablicą słów

$$\texttt{Rcon[i]} = \left[\{02\}^{i-1}, \{00\}, \{00\}, \{00\}\right], \quad i = 1, 2, ..., 10,$$

których pierwszym bajtem jest $i - ta$ potęga elementu $\{02\}$ ciała $GF(256)$.

Stąd

i	1	2	3	4	5
$\{02\}^{i-1}$	$\{01\}$	$\{02\}$	$\{04\}$	$\{08\}$	$\{10\}$
i	6	7	8	9	10
$\{02\}^{i-1}$	$\{20\}$	$\{40\}$	$\{08\}$	$\{1b\}$	$\{36\}$

W procedurze Key Expansion pierwsze Nk słów rozszerzonego klucza $w[0], w[1], \ldots, w[Nk-1]$ to klucz szyfrujący K. Natomiast każde kolejne słowo $w[i]$ jest równe XOR poprzedniego słowa, $w[i-1]$ i słowa występującego Nk pozycji wcześniej $w[i-Nk]$ przy czym do słów, których indeksy są wielokrotnościami Nk przed wykonaniem operacji XOR stosowane jest przekształcenie słowa $w[i-1]$ polegające na cyklicznym przesunięciu bajtów w słowie (`RotWord()`), po którym następuje zastosowanie funkcji `SubWord()`, a następnie wynik ten jest XORowany ze stałą rundy `Rcon`[i/Nk].

Gdy zastosujemy tę procedurę do 128- bitowego klucza, wówczas otrzymujemy $Nk = 4$ i $w[0], w[1], \ldots, w[3]$ są słowami pochodzącymi z klucza K. Pozostałe 40 słów, które są efektem działania procedury Key Expansion powstaje w ten sposób, że dla $i = 4, \ldots, 43$ $w[i] = w[i-1] \oplus w[i-4]$, przy czym należy pamiętać, że przed wykonaniem operacji XOR na słowach $w[i-1]$, gdzie $i \in \{4, 8, 12, 16, 20, 24, 28, 32, 36, 40\}$ stosowane są dodatkowe przekształcenia: `RotWord()`, po którym następuje `SubByte()` do wszystkich czterech bajtów słowa i dodatkowo wynik ten jest poddawany działaniu funkcji XOR ze stałą rundy `Rcon`[i/4].

Warto zauważyć, że procedura rozszerzania klucza dla 256-bitowych kluczy szyfrowania ($Nk = 8$) różni się nieco od tej dla 128- i 196-bitowych kluczy szyfrowania. Jeśli $Nk = 8$ oraz $i-4$ jest wielokrotnością Nk, to przed wyznaczeniem $w[i]$ do słowa $w[i-1]$ jest stosowane przekształcenie `SubWord()`.

5.2.7. Algorytm szyfrowania

Algorytm szyfrowania można opisać za pomocą następującego pseudokodu:

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]

    state = in

    AddRoundKey(state, w[0, Nb-1])

    for round = 1 step 1 to Nr{1
    SubBytes(state)
    ShiftRows(state)
    MixColumns(state)
    AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    end for

    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

    out = state
end
```

Na początku szyfrowania dane wejściowe są kopiowane do tablicy stanu według reguł podanych wcześniej. Następnie, po wykonaniu dodawania modulo dwa kolumn tablicy stanu i kolumn początkowej tablicy klucza szyfrującego, tablica stanu jest przekształcana przez funkcję rundową 10, 12 lub 14 razy w zależności od długości klucza, przy czym wszystkie rundy są identyczne z wyjątkiem ostatniej, w której nie wykonuje się mieszania danych w kolumnach tablicy stanu. Szyfrowanie kończy skopiowanie tablicy stanu do tablicy wyjściowej `out`.

Aby odszyfrować szyfrogram, AES stosuje funkcje odwrotne do poszczególnych działań, zaimplementowane w odwrotnej kolejności. I tak odwrotna tabela wyszukiwania `InvSubBytes()` odwraca przekształcenie `SubBytes()`, `InvShiftRows()` przesuwa w odwrotnym kierunku, stosowane jest odwrócenie `MixColumns()` - `InvMixColumns()` (jak w macierzy odwrotnej do macierzy kodującej to działanie), a działanie XOR wykorzystywane w funkcji `AddRoundKey()` pozostaje bez zmian, gdyż działaniem odwrotnym do XOR jest kolejne XOR.

5.3. Kodowanie boolowskie AES

W tym podrozdziale opisano wykonane przez autorkę przekształcenia do formuł boolowskich przedstawionych wcześniej operacji występujących w algorytmie AES, używając do tego metody bezpośredniego kodowania boolowskiego. W pierwszej kolejności przedstawiono procedurę kodowania funkcji `SubBytes()`, a następnie przekształcenia `MixColumns()`oraz operacji `AddRoundKey()`. Przekształcenia `RotWord()` oraz `ShiftRows()` zostały zaimplementowane w ten sposób, że do ich wykonania nie użyto nowych zmiennych, zatem nie było konieczności kodowania boolowskiego tych funkcji.

5.3.1. Kodowanie przekształcenia `SubBytes()`

Przekształcenie `SubBytes()` jest wykorzystywane w algorytmie AES zarówno w procedurze tworzenia tablicy kluczy, jak i w głównej funkcji rundowej. Przypomnijmy, że zastosowany w szyfrze AES S-box jest macierzą kwadratową 16×16, przy czym każdy wyraz tej macierzy jest innym ośmiobitowym wektorem. Wektory te są przedstawione w postaci szesnastkowej jako dwuznaki. Zatem możemy S-box rozważyć jako funkcję typu $S_{box} : \{0,1\}^8 \to \{0,1\}^8$. Dla uproszczenia przez $\bar{x}$ oznaczmy wektor $(x_1, \ldots, x_8)$ oraz przez $S^k_{box}(\bar{x})$ k-tą współrzędną wartości $S_{box}(\bar{x})$ dla $k = 1, 2, \ldots, 8$. S-box możemy zakodować jako następującą formułę boolowską:

$$\phi_{S_{box}} : \bigwedge_{\bar{x} \in \{0,1\}^8} \left(\bigwedge_{i=1}^{8} (\neg)^{1-x_i} u_i \Rightarrow \bigwedge_{j=1}^{8} (\neg)^{1-S^j_{box}(\bar{x})} q_j \right),$$

gdzie $(u_1, \ldots, u_8)$ jest wektorem reprezentującym dane wejściowe do S-boxa i $(q_1, \ldots, q_8)$ jest wektorem danych wyjściowych. Dodatkowo przez $(\neg)^0 u$ i $(\neg)^1 u$ oznaczamy odpowiednio u i $\neg u$. Formuła kodująca będzie miała postać koniunkcji 256 implikacji. Implikacje zbudowane są w taki sposób, że przesłanka każdej z nich zależna jest od innego wektora z przestrzeni $\{0,1\}^8$. Oznacza to, że dla zadanego wektora u tylko jedna z tych implikacji będzie prawdziwa.
Zauważmy, że funkcja `SubBytes()` jest odwracalna.

5.3.2. Kodowanie przekształcenia `MixColumns()`

Przez $(u_1, u_2, \ldots, u_{128})$, gdzie $u_i \in \{0,1\}$, dla $i = 1, 2, \ldots, 128$ oznaczmy wektor przedstawiającym dane wejściowe zapisane w tablicy stanu oraz przez $(q_1, q_2, \ldots, q_{128})$, gdzie $q_i \in \{0,1\}$, dla $i = 1, 2, \ldots, 128$, oznaczmy wektor reprezentujące dane wyjściowe po operacji `MixColumns()`. Procedura `MixColumns()`

jest wykonywana na poszczególnych kolumnach stanu, przy czym przypomnijmy, że kolumny są numerowane od 0. Zatem działanie funkcji `MixColumns()` można zapisać następująco: równanie

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

dla pierwszej kolumny stanu zapisane za pomocą formuły boolowskiej jest postaci:

$$\wedge \begin{cases} \bigwedge\limits_{i=1,\ldots,3} & q_i \Leftrightarrow u_{i+1} \oplus \Big(u_{(i+8)+1} \oplus u_{i+8}\Big) \oplus u_{i+16} \oplus u_{i+24} \\ \bigwedge\limits_{i=4,5} & q_i \Leftrightarrow u_{i+1} \oplus u_1 \oplus \Big(u_{(i+8)+1} \oplus u_9 \oplus u_{i+8}\Big) \oplus u_{i+16} \oplus u_{i+24} \\ \bigwedge\limits_{i=6} & q_i \Leftrightarrow u_{i+1} \oplus \Big(u_{(i+8)+1} \oplus u_{i+8}\Big) \oplus u_{i+16} \oplus u_{i+24} \\ \bigwedge\limits_{i=7} & q_i \Leftrightarrow u_{i+1} \oplus u_1 \oplus \Big(u_{(i+8)+1} \oplus u_9 \oplus u_{i+8}\Big) \oplus u_{i+16} \oplus u_{i+24} \\ \bigwedge\limits_{i=8} & q_i \Leftrightarrow u_1 \oplus (u_9 \oplus u_{i+8}) \oplus u_{i+16} \oplus u_{i+24}. \end{cases}$$

Drugie równanie opisujujące działanie procedury `MixColumns()`

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

zastosowane do kolumny o indeksie $c = 0$ można zapisać następująco:

$$\wedge \begin{cases} \bigwedge\limits_{i=9,\ldots,11} & q_i \Leftrightarrow u_{i-8} \oplus u_{i+1} \oplus \Big(u_{(i+8)+1} \oplus u_{i+8}\Big) \oplus u_{i+16} \\ \bigwedge\limits_{i=12,13} & q_i \Leftrightarrow u_{i-8} \oplus u_{i+1} \oplus u_9 \oplus \Big(u_{(i+8)+1} \oplus u_{17} \oplus u_{i+8}\Big) \oplus u_{i+16} \\ \bigwedge\limits_{i=14} & q_i \Leftrightarrow u_{i-8} \oplus u_{i+1} \oplus \Big(u_{(i+8)+1} \oplus u_{i+8}\Big) \oplus u_{i+16} \\ \bigwedge\limits_{i=15} & q_i \Leftrightarrow u_{i-8} \oplus u_{i+1} \oplus u_9 \oplus \Big(u_{(i+8)+1} \oplus u_{17} \oplus u_{i+8}\Big) \oplus u_{i+16} \\ \bigwedge\limits_{i=16} & q_i \Leftrightarrow u_{i-8} \oplus u_9 \oplus (u_{17} \oplus u_{i+8}) \oplus u_{i+16}. \end{cases}$$

Stosując kodowanie boolowskie do trzeciego równia

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

otrzymujemy:

$$\wedge \begin{cases} \bigwedge\limits_{i=17,\ldots,19} & q_i \Leftrightarrow u_{i-16} \oplus u_{i-8} \oplus u_{i+1} \oplus \Big(u_{(i+8)+1} \oplus u_{i+8}\Big) \\ \bigwedge\limits_{i=20,21} & q_i \Leftrightarrow u_{i-16} \oplus u_{i-8} \oplus u_{i+1} \oplus u_{17} \oplus \Big(u_{(i+8)+1} \oplus u_{25} \oplus u_{i+8}\Big) \\ \bigwedge\limits_{i=22} & q_i \Leftrightarrow u_{i-16} \oplus u_{i-8} \oplus u_{i+1} \oplus \Big(u_{(i+8)+1} \oplus u_{i+8}\Big) \\ \bigwedge\limits_{i=23} & q_i \Leftrightarrow u_{i-16} \oplus u_{i-8} \oplus u_{i+1} \oplus u_{17} \oplus \Big(u_{(i+8)+1} \oplus u_{25} \oplus u_{i+8}\Big) \\ \bigwedge\limits_{i=24} & q_i \Leftrightarrow u_{i-16} \oplus u_{i-8} \oplus u_{17} \oplus \big(u_{25} \oplus u_{i+8}\big)\,. \end{cases}$$

Ostatnie równanie opisujące zmiany dokonywane przez funkcję `MixColumns()` w ostatnim wierszu stanu

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c})$$

zastosowane do pierwszej kolumny przyjmuje postać:

$$\wedge \begin{cases} \bigwedge\limits_{i=25,\ldots,27} & q_i \Leftrightarrow \Big(u_{(i-24)+1} \oplus u_{i-24}\Big) \oplus u_{i-16} \oplus u_{i-8} \oplus u_{i+1} \\ \bigwedge\limits_{i=28,29} & q_i \Leftrightarrow \Big(u_{(i-24)+1} \oplus u_1 \oplus u_{i-24}\Big) \oplus u_{i-16} \oplus u_{i-8} \oplus u_{i+1} \oplus u_{25} \\ \bigwedge\limits_{i=30} & q_i \Leftrightarrow \Big(u_{(i-24)+1} \oplus u_{i-24}\Big) \oplus u_{i-16} \oplus u_{i-8} \oplus u_{i+1} \\ \bigwedge\limits_{i=31} & q_i \Leftrightarrow \Big(u_{(i-24)+1} \oplus u_1 \oplus u_{i-24}\Big) \oplus u_{i-16} \oplus u_{i-8} \oplus u_{i+1} \oplus u_{25} \\ \bigwedge\limits_{i=32} & q_i \Leftrightarrow (u_1 \oplus u_{i-24}) \oplus u_{i-16} \oplus u_{i-8} \oplus u_{25}. \end{cases}$$

Pamiętając o tym, że kolumny stanu są oznaczane przez c, przy czym kolumna pierwsza ma indeks 0, możemy zapisać formułę kodującą przekształcenie `MixColumns()` dla wszystkich kolumn stanu i ma ona postać koniunkcji odpo-

wiednich równoważności, tak jak widzimy to poniżej:

$$
\begin{cases}
q_{1+32c} \Leftrightarrow u_{2+32c} \oplus (u_{10+32c} \oplus u_{9+32c}) \oplus u_{17+32c} \oplus u_{25+32c} \\
q_{2+32c} \Leftrightarrow u_{3+32c} \oplus (u_{11+32c} \oplus u_{10+32c}) \oplus u_{18+32c} \oplus u_{26+32c} \\
q_{3+32c} \Leftrightarrow u_{4+32c} \oplus (u_{12+32c} \oplus u_{11+32c}) \oplus u_{19+32c} \oplus u_{27+32c} \\
q_{4+32c} \Leftrightarrow u_{5+32c} \oplus u_{1+32c} \oplus (u_{13+32c} \oplus u_{9+32c} \oplus u_{12+32c}) \oplus u_{20+32c} \oplus u_{28+32c} \\
q_{5+32c} \Leftrightarrow u_{6+32c} \oplus u_{1+32c} \oplus (u_{14+32c} \oplus u_{9+32c} \oplus u_{13+32c}) \oplus u_{21+32c} \oplus u_{29+32c} \\
q_{6+32c} \Leftrightarrow u_{7+32c} \oplus (u_{10+32c} \oplus u_{9+32c}) \oplus u_{22+32c} \oplus u_{30+32c} \\
q_{7+32c} \Leftrightarrow u_{8+32c} \oplus u_{1+32c} \oplus (u_{16+32c} \oplus u_{9+32c} \oplus u_{15+32c}) \oplus u_{23+32c} \oplus u_{31+32c} \\
q_{8+32c} \Leftrightarrow u_{1+32c} \oplus (u_{9+32c} \oplus u_{16+32c}) \oplus u_{24+32c} \oplus u_{32+32c} \\
q_{9+32c} \Leftrightarrow u_{1+32c} \oplus u_{10+32c} \oplus (u_{18+32c} \oplus u_{17+32c}) \oplus u_{25+32c} \\
q_{10+32c} \Leftrightarrow u_{2+32c} \oplus u_{11+32c} \oplus (u_{19+32c} \oplus u_{18+32c}) \oplus u_{26+32c} \\
q_{11+32c} \Leftrightarrow u_{3+32c} \oplus u_{12+32c} \oplus (u_{20+32c} \oplus u_{19+32c}) \oplus u_{27+32c} \\
q_{12+32c} \Leftrightarrow u_{4+32c} \oplus u_{13+32c} \oplus u_{9+32c} \oplus (u_{21+32c} \oplus u_{17+32c} \oplus u_{20+32c}) \oplus u_{28+32c} \\
q_{13+32c} \Leftrightarrow u_{5+32c} \oplus u_{14+32c} \oplus u_{9+32c} \oplus (u_{23+32c} \oplus u_{17+32c} \oplus u_{21+32c}) \oplus u_{29+32c} \\
q_{14+32c} \Leftrightarrow u_{6+32c} \oplus u_{15+32c} \oplus (u_{23+32c} \oplus u_{22+32c}) \oplus u_{30+32c} \\
q_{15+32c} \Leftrightarrow u_{7+32c} \oplus u_{16+32c} \oplus u_{9+32c} \oplus (u_{24+32c} \oplus u_{17+32c} \oplus u_{23+32c}) \oplus u_{31+32c} \\
q_{16+32c} \Leftrightarrow u_{8+32c} \oplus u_{9+32c} \oplus (u_{17+32c} \oplus u_{24+32c}) \oplus u_{32+32c} \\
q_{17+32c} \Leftrightarrow u_{1+32c} \oplus u_{9+32c} \oplus u_{18+32c} \oplus (u_{26+32c} \oplus u_{25+32c}) \\
q_{18+32c} \Leftrightarrow u_{2+32c} \oplus u_{10+32c} \oplus u_{19+32c} \oplus (u_{27+32c} \oplus u_{26+32c}) \\
q_{19+32c} \Leftrightarrow u_{3+32c} \oplus u_{11+32c} \oplus u_{20+32c} \oplus (u_{28+32c} \oplus u_{27+32c}) \\
q_{20+32c} \Leftrightarrow u_{4+32c} \oplus u_{12+32c} \oplus u_{21+32c} \oplus u_{17+32c} \oplus (u_{29+32c} \oplus u_{25+32c} \oplus u_{28+32c}) \\
q_{21+32c} \Leftrightarrow u_{5+32c} \oplus u_{13+32c} \oplus u_{22+32c} \oplus u_{17+32c} \oplus (u_{30+32c} \oplus u_{25+32c} \oplus u_{29+32c}) \\
q_{22+32c} \Leftrightarrow u_{6+32c} \oplus u_{14+32c} \oplus u_{23+32c} \oplus (u_{31+32c} \oplus u_{30+32c}) \\
q_{23+32c} \Leftrightarrow u_{7+32c} \oplus u_{15+32c} \oplus u_{24+32c} \oplus u_{17+32c} \oplus (u_{32+32c} \oplus u_{25+32c} \oplus u_{31+32c}) \\
q_{24+32c} \Leftrightarrow u_{8+32c} \oplus u_{16+32c} \oplus u_{17+32c} \oplus (u_{25+32c} \oplus u_{32+32c}) \\
q_{25+32c} \Leftrightarrow (u_{2+32c} \oplus u_{1+32c}) \oplus u_{9+32c} \oplus u_{17+32c} \oplus u_{26+32c} \\
q_{26+32c} \Leftrightarrow (u_{3+32c} \oplus u_{2+32c}) \oplus u_{10+32c} \oplus u_{18+32c} \oplus u_{27+32c} \\
q_{27+32c} \Leftrightarrow (u_{4+32c} \oplus u_{3+32c}) \oplus u_{11+32c} \oplus u_{19+32c} \oplus u_{28+32c} \\
q_{28+32c} \Leftrightarrow (u_{5+32c} \oplus u_{1+32c} \oplus u_{4+32c}) \oplus u_{12+32c} \oplus u_{20+32c} \oplus u_{29+32c} \oplus u_{25+32c} \\
q_{29+32c} \Leftrightarrow (u_{6+32c} \oplus u_{1+32c} \oplus u_{5+32c}) \oplus u_{13+32c} \oplus u_{21+32c} \oplus u_{30+32c} \oplus u_{25+32c} \\
q_{30+32c} \Leftrightarrow (u_{7+32c} \oplus u_{6+32c}) \oplus u_{14+32c} \oplus u_{22+32c} \oplus u_{31+32c} \\
q_{31+32c} \Leftrightarrow (u_{8+32c} \oplus u_{1+32c} \oplus u_{6+32c}) \oplus u_{14+32c} \oplus u_{22+32c} \oplus u_{31+32c} \oplus u_{25+32c} \\
q_{32+32c} \Leftrightarrow (u_{1+32c} \oplus u_{6+32c}) \oplus u_{16+32c} \oplus u_{24+32c} \oplus u_{25+32c}.
\end{cases}
$$

5.3.3. Kodowanie przekształcenia `MixColumns()` dla dowolnego wielomianu pierwotnego

Jak już zaznaczono wcześniej przekształcenie `MixColumns()` wykorzystuje jako działanie mnożenia kolumn mnożenie w ciele reszt modulo pewien, charakterystyczny wielomian pierwotny: $m(x) = x^8 + x^4 + x^3 + x + 1$.

Poniżej zaprezentowana zostanie metoda bezpośredniego kodowania przekształcenia `MixColumns()` z zastosowaniem dowolnego wielomianu pierwotnego stopnia 7. Poniższy przykład jest prezentowany dla wielomianu $m(x)$, jednak metodę łatwo dostosować do dowolnego wielomianu.

Aby zakodować tę operację boolowsko należy zakodować dzielenie z resztą dla wielomianów o współczynnikach ze zbioru $\{0, 1\}$ o maksymalnym stopniu 7. Aby to uczynić przedstawiony będzie przykład takiej operacji na konkretnym przykładzie:

Rozważmy dzielenie wielomianu:

$f(x) = x^{14} + x^{12} + x^{11} + x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$ przez wielomian

$m(x) = x^8 + x^4 + x^3 + x + 1$.

Wykonując kolejno odpowiednie operacje matematyczne otrzymujemy:

$$
\begin{array}{l}
x^6 + x^4 + x^3 + x \\
\hline
(x^{14} + x^{12} + x^{11} + x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1) : (x^8 + x^4 + x^3 + x + 1) \\
\quad x^{14} + x^{10} + x^9 + x^7 + x^6 \\
+ \quad \hline
\qquad (x^{12} + x^{11} + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^2 + x + 1) \\
\qquad x^{12} + x^8 + x^7 + x^5 + x^4 \\
\qquad + \quad \hline
\qquad\qquad (x^{11} + x^9 + x^6 + x^2 + x + 1) \\
\qquad\qquad x^{11} + x^7 + x^6 + x^4 + x^3 \\
\qquad\quad + \quad \hline
\qquad\qquad (x^9 + x^7 + x^4 + x^3 + x^2 + x + 1) \\
\qquad\qquad x^9 + x^5 + x^4 + x^2 + x \\
\qquad\quad + \quad \hline
\qquad\qquad x^7 + x^5 + x^3 + 1
\end{array}
$$

Otrzymujemy zależność:

$$(x^{14} + x^{12} + x^{11} + x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1) =$$
$$= (x^8 + x^4 + x^3 + x + 1) \cdot (x^6 + x^4 + x^3 + x) + (x^7 + x^5 + x^3 + 1).$$

Zatem resztą z dzielenia wielomianu

$f(x) = x^{14} + x^{12} + x^{11} + x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1$ przez wielomian $m(x) = x^8 + x^4 + x^3 + x + 1$ jest $h(x) = x^7 + x^5 + x^3 + 1$.

Do opracowania kodowania boolowskiego opisanej operacji posłużmy się nieco prostszym przykładem dla wielomianów dwa razy niższych stopni, czyli wielomianów reprezentowanych przez czwórki bitów. W tym przypadku mamy wielomiany trzeciego stopnia, które mogą być wzajemnie mnożone przez siebie otrzymując w ten sposób wielomian szóstego stopnia. Aby określić pierścień reszt modulo należy wyniki tych mnożeń dzielić przez wielomian czwartego stopnia.

Rozważmy poniższe dzielenie. Niech $f(x) = x^6 + x^5 + x^4 + x^2 + 1$ oraz $h(x) = x^4 + x + 1$.

Mamy wtedy następujące operacje:

$$
\begin{array}{l}
 \quad x^2 + x + 1 \\
\hline
(x^6 + x^5 + x^4 + x^2 + 1) : (x^4 + x + 1) \\
\ x^6 + x^3 + x^2 \\
+ \ \overline{\qquad\qquad\qquad} \\
\qquad\quad (x^5 + x^4 + x^3 + 1) \\
\qquad\quad\ x^5 + x^2 + x \\
\qquad + \ \overline{\qquad\qquad\qquad} \\
\qquad\qquad\quad (x^4 + x^3 + x^2 + x + 1) \\
\qquad\qquad\quad\ x^4 + x + 1 \\
\qquad\qquad + \ \overline{\qquad\qquad\qquad} \\
\qquad\qquad\qquad\quad x^3 + x^2
\end{array}
$$

Czyli otrzymaliśmy:

$$(x^6 + x^5 + x^4 + x^2 + 1) = (x^2 + x + 1) \cdot (x^4 + x + 1) + (x^3 + x^2)$$

Rozważmy to samo działanie kodując już współczynniki wielomianów przez odpowiednie zmienne zdaniowe oraz uwzględniając współczynniki zerowe. Te ostatnie oznaczane będą oznaczone podkreśleniami nad zmiennymi.

Otrzymujemy poniższe operacje:

$$
\begin{array}{l}
w_2 + w_1 + w_0 \\
\hline
(p_6^1 + p_5^1 + p_4^1 + \overline{p_3^1} + p_2^1 + \overline{p_1^1} + p_0^1) \mid (d_4^1 + \overline{d_3^1} + \overline{d_2^1} + d_1^1 + d_0^1) \\
(q_6^1 + \overline{q_5^1} + \overline{q_4^1} + q_3^1 + q_2^1) \\
\hline
\qquad (p_5^2 + p_4^2 + p_3^2 + \overline{p_2^2} + \overline{p_1^2} + p_0^2) \\
\qquad q_5^2 + \overline{q_4^2} + \overline{q_3^2} + q_2^2 + q_1^2 \\
\qquad \hline
\qquad\qquad (p_4^3 + p_3^3 + p_2^3 + p_1^3 + p_0^3) \\
\qquad\qquad q_4^3 + \overline{q_3^3} + \overline{q_2^3} + q_1^3 + q_0^3 \\
\qquad\qquad \hline
\qquad\qquad\qquad p_3^4 + p_2^4 + \overline{p_1^4} + \overline{p_0^4}
\end{array}
$$

Jak widać w powyższym przykładzie biorą już udział wszystkie współczynniki niezależnie od tego, czy są zerowe czy nie.

Po analizie powyższych zależności można już zacząć kodowanie operacji dzielenia z resztą w pierścieniach wielomianów modulo wielomian.

$$
\begin{array}{l}
w_2 \Leftrightarrow (p_6^1 \wedge d_4^1) \\
q_6^1 \Leftrightarrow (w_2 \wedge d_4^1) \\
q_5^1 \Leftrightarrow (w_2 \wedge d_3^1) \\
q_4^1 \Leftrightarrow (w_2 \wedge d_2^1) \\
q_3^1 \Leftrightarrow (w_2 \wedge d_1^1) \\
q_2^1 \Leftrightarrow (w_2 \wedge d_0^1)
\end{array}
$$

W notacji uogólnionej mamy:

$$w_2 \Leftrightarrow (p_6^1 \wedge d_4^1) \wedge \bigwedge_{i=6,\ldots,2} (q_i^1 \Leftrightarrow (w_2 \wedge d_{i+2}^1)).$$

Po obliczeniu pierwszego wiersza obliczamy wiersz drugi:

$$
\begin{array}{l}
p_5^2 \Leftrightarrow (p_5^1 \wedge q_5^1) \\
p_4^2 \Leftrightarrow (p_4^1 \wedge q_4^1) \\
p_3^2 \Leftrightarrow (p_3^1 \wedge q_3^1) \\
p_2^2 \Leftrightarrow (p_2^1 \wedge q_2^1) \\
p_1^2 \Leftrightarrow (p_1^1 \wedge q_1^1) \\
p_0^2 \Leftrightarrow (p_0^1 \wedge q_0^1)
\end{array}
$$

W notacji uogólnionej mamy:

$$\bigwedge_{i=5,\ldots,0} (p_i^2 \Leftrightarrow (p_i^1 \wedge q_i^1)).$$

Dalej obliczając mamy

$w_1 \Leftrightarrow (p_5^1 \wedge d_4^1)$
$q_4^2 \Leftrightarrow (w_1 \wedge d_4^1)$
$q_3^2 \Leftrightarrow (w_1 \wedge d_3^1)$
$q_2^2 \Leftrightarrow (w_1 \wedge d_2^1)$
$q_1^2 \Leftrightarrow (w_1 \wedge d_1^1)$
$q_0^2 \Leftrightarrow (w_1 \wedge d_0^1)$

W notacji uogólnionej:

$$w_1 \quad \Leftrightarrow \quad (p_5^1 d_4^1) \wedge \bigwedge_{i=4,\ldots,0} (q_i^2 \Leftrightarrow (w_1 \wedge d_i^2)).$$

Postępując tak dalej otrzymujemy kolejno kodowanie dzielenia używając wszystkich zmiennych określających wielomiany wsadowe działania. W ten sposób możemy kodować przekształcenie `MixColumns()` z zastosowaniem dowolnego wielomianu pierwotnego określonego stopnia.

5.3.4. Kodowanie przekształcenia `AddRoundKey()`

Procedura `AddRoundKey()` polega na XORowaniu dwóch ciągów bitów. Stosując metodę bezpośredniego kodowania przekształcenia `AddRoundKey()` przyjmijmy, że $(u_1, u_2, \ldots, u_{128})$ i $(v_1, v_2, \ldots, v_{128})$ będą wektorami reprezentującymi dane wejściowe oraz $u_i \in \{0, 1\}$ i $v_i \in \{0, 1\}$, dla $i = 1, 2, \ldots, 128$ i niech wektor $(q_1, q_2, \ldots, q_{128})$, gdzie $q_i \in \{0, 1\}$, dla $i = 1, 2, \ldots, 128$, będzie przedstawiał wynik zastosowania procedury `AddRoundKey()`. Wtedy kodowanie boolowskie przekształcenia `AddRoundKey()` można przedstawić w następującej postaci:

$$\bigwedge_{i=1,\ldots,128} q_i \Leftrightarrow (u_i \oplus v_i)\,.$$

Jak można zauważyć zakodowana do postaci formuły boolowskiej procedura `AddRoundKey()` składa się z koniunkcji odpowiednich równoważności.

Będąc w posiadaniu tak zakodowanych przekształceń, które wchodzą w skład algorytmu AES, można utworzyć formułę kodującą dowolną liczbę rund AES.

5.4. Translacje formuł do formatu DIMACS

Ten podrozdział jest poświęcony opisowi konwersji otrzymanych w wyniku bezpośredniego kodowania boolowskiego formuł zdaniowych opisujących działanie funkcji występujących w algorytmie AES, przedstawionych w podrozdziale 5.3., do koniunkcyjnej postaci normalnej, wykorzystując metodę ich przekształcania opisaną w podrozdziale 2.6., a następnie do formatu DIMACS, który jest formatem danych wejściowych do SAT-solverów.

5.4.1. Przekształcenie `SubBytes()`

Stosując kodowanie funkcji `SubBytes()` opisane w części 5.3.1. dla jednego bajta danych wejściowych reprezentowanego przez wektor $(u_1, \ldots, u_8)$ i jednego bajta danych wyjściowych $(q_1, \ldots, q_8)$ można zapisać następującą formułę :

$$\wedge \begin{cases} (u_1 \wedge u_2 \wedge u_3 \wedge u_4 \wedge u_5 \wedge u_6 \wedge u_7 \wedge u_8) \Rightarrow q_1 \\ (u_1 \wedge u_2 \wedge u_3 \wedge u_4 \wedge u_5 \wedge u_6 \wedge u_7 \wedge u_8) \Rightarrow q_2 \\ (u_1 \wedge u_2 \wedge u_3 \wedge u_4 \wedge u_5 \wedge u_6 \wedge u_7 \wedge u_8) \Rightarrow q_3 \\ (u_1 \wedge u_2 \wedge u_3 \wedge u_4 \wedge u_5 \wedge u_6 \wedge u_7 \wedge u_8) \Rightarrow q_4 \\ (u_1 \wedge u_2 \wedge u_3 \wedge u_4 \wedge u_5 \wedge u_6 \wedge u_7 \wedge u_8) \Rightarrow q_5 \\ (u_1 \wedge u_2 \wedge u_3 \wedge u_4 \wedge u_5 \wedge u_6 \wedge u_7 \wedge u_8) \Rightarrow q_6 \\ (u_1 \wedge u_2 \wedge u_3 \wedge u_4 \wedge u_5 \wedge u_6 \wedge u_7 \wedge u_8) \Rightarrow q_7 \\ (u_1 \wedge u_2 \wedge u_3 \wedge u_4 \wedge u_5 \wedge u_6 \wedge u_7 \wedge u_8) \Rightarrow q_8. \end{cases}$$

Dokonując odpowiednich przekształceń przedstawionych w 2.6. otrzymano formułę równoważną danej:

$$\wedge \begin{cases} (\neg u_1 \vee \neg u_2 \vee \neg u_3 \vee \neg u_4 \vee \neg u_5 \vee \neg u_6 \vee \neg u_7 \vee \neg u_8 \vee q_1) \\ (\neg u_1 \vee \neg u_2 \vee \neg u_3 \vee \neg u_4 \vee \neg u_5 \vee \neg u_6 \vee \neg u_7 \vee \neg u_8 \vee q_2) \\ (\neg u_1 \vee \neg u_2 \vee \neg u_3 \vee \neg u_4 \vee \neg u_5 \vee \neg u_6 \vee \neg u_7 \vee \neg u_8 \vee q_3) \\ (\neg u_1 \vee \neg u_2 \vee \neg u_3 \vee \neg u_4 \vee \neg u_5 \vee \neg u_6 \vee \neg u_7 \vee \neg u_8 \vee q_4) \\ (\neg u_1 \vee \neg u_2 \vee \neg u_3 \vee \neg u_4 \vee \neg u_5 \vee \neg u_6 \vee \neg u_7 \vee \neg u_8 \vee q_5) \\ (\neg u_1 \vee \neg u_2 \vee \neg u_3 \vee \neg u_4 \vee \neg u_5 \vee \neg u_6 \vee \neg u_7 \vee \neg u_8 \vee q_6) \\ (\neg u_1 \vee \neg u_2 \vee \neg u_3 \vee \neg u_4 \vee \neg u_5 \vee \neg u_6 \vee \neg u_7 \vee \neg u_8 \vee q_7) \\ (\neg u_1 \vee \neg u_2 \vee \neg u_3 \vee \neg u_4 \vee \neg u_5 \vee \neg u_6 \vee \neg u_7 \vee \neg u_8 \vee q_8). \end{cases}$$

Ponieważ liczba różnych wektorów ośmiobitowych jest równa 256, to do zapisania w formacie DIMACS formuły kodującej przekształcenie przez funkcję `SubBytes()` jednego bajta potrzeba 16 zmiennych oraz 2048 klauzul. Natomiast zapisanie formuły kodującej przekształcenie `SubBytes()` w złożonej z 16 bajtów tablicy stanu wymaga zapisania ponad 30 tysięcy klauzul z użyciem 256 zmiennych. W pracy [105] oszacowano, że do zakodowania S-boxa użytego w szyfrze AES (przy użyciu narzędzia CryotLogVer do formuły) potrzeba około 4800 klauzul i 900 zmiennych.

Na rysunku 5.8 znajduje się tylko niewielki wycinek z pliku tekstowego zawierającego zakodowaną funkcję `SubBytes()` przekonwertowaną do formatu DIMACS.

```
577 578 579 580 581 582 583 584 -897 0
577 578 579 580 581 582 583 584 898 0
577 578 579 580 581 582 583 584 899 0
577 578 579 580 581 582 583 584 -900 0
577 578 579 580 581 582 583 584 -901 0
577 578 579 580 581 582 583 584 -902 0
577 578 579 580 581 582 583 584 903 0
577 578 579 580 581 582 583 584 904 0
585 586 587 588 589 590 591 592 -905 0
585 586 587 588 589 590 591 592 906 0
585 586 587 588 589 590 591 592 907 0
585 586 587 588 589 590 591 592 -908 0
585 586 587 588 589 590 591 592 -909 0
585 586 587 588 589 590 591 592 -910 0
585 586 587 588 589 590 591 592 911 0
585 586 587 588 589 590 591 592 912 0
593 594 595 596 597 598 599 600 -913 0
593 594 595 596 597 598 599 600 914 0
593 594 595 596 597 598 599 600 915 0
593 594 595 596 597 598 599 600 -916 0
593 594 595 596 597 598 599 600 -917 0
593 594 595 596 597 598 599 600 -918 0
593 594 595 596 597 598 599 600 919 0
593 594 595 596 597 598 599 600 920 0
601 602 603 604 605 606 607 608 -921 0
601 602 603 604 605 606 607 608 922 0
601 602 603 604 605 606 607 608 923 0
601 602 603 604 605 606 607 608 -924 0
601 602 603 604 605 606 607 608 -925 0
```

Rysunek 5.8: Fragment pliku z formułą kodującą `SubBytes()` w formacie DIMACS

5.4.2. Przekształcenie `MixColumns()`

Przekształcając formułę kodującą funkcję `MixColumns()` do koniunkcyjnej postaci normalnej, dla uproszczenia występujących w niej złożonych koniunkcji alternatyw, wprowadzono dodatkowe zmienne, które reprezentują wyniki częściowych operacji występujących w przekształceniu `MixColumns()`. I tak **kodowanie boolowskie** operacji $02 \bullet r$, gdzie r jest zmienną reprezentującą bajt danych wejściowych oznaczoną przez $r = (r_1, \ldots, u_8)$, gdzie $u_i \in \{0, 1\}$, dla $i = 1, 2, \ldots, 8$, a wektor $(p_1, \ldots, p_8)$, gdzie $p_i \in \{0, 1\}$, dla $i = 1, 2, \ldots, 8$ reprezentuje bajt danych wyjściowych, ma postać:

$$\wedge \begin{cases} p_1 \Leftrightarrow r_2 \\ p_2 \Leftrightarrow r_3 \\ p_3 \Leftrightarrow r_4 \\ p_4 \Leftrightarrow r_5 \oplus r_1 \\ p_5 \Leftrightarrow r_6 \oplus r_1 \\ p_6 \Leftrightarrow r_7 \\ p_7 \Leftrightarrow r_8 \oplus r_1 \\ p_8 \Leftrightarrow r_1 \end{cases}$$

Dodatkowo **kodowanie boolowskie** operacji $03 \bullet r$, gdzie podobnie jak wyżej r jest wektorem reprezentującym dane wejściowe, a sekwencja $(t_1, \ldots, t_8)$, gdzie $t_i \in \{0, 1\}$, dla $i = 1, 2, \ldots, 8$ reprezentuje bajt wyjścia, ma postać:

$$\wedge \begin{cases} t_1 \Leftrightarrow r_2 \oplus r_1 \\ t_2 \Leftrightarrow r_3 \oplus r_2 \\ t_3 \Leftrightarrow r_4 \oplus r_3 \\ t_4 \Leftrightarrow (r_5 \oplus r_1) \oplus r_4 \\ t_5 \Leftrightarrow (r_6 \oplus r_1) \oplus r_5 \\ t_6 \Leftrightarrow r_7 \oplus r_6 \\ t_7 \Leftrightarrow (r_8 \oplus r_1) \oplus r_7 \\ t_8 \Leftrightarrow r_1 \oplus r_8 \end{cases}$$

Używając wyżej zdefiniowanych zmiennych można zapisać przekształcenie **jednego bita** przez funkcję `MixColumns` jako następującą formułę boolowską:

$$q_1 \Leftrightarrow p_1 \oplus t_1 \oplus w_1 \oplus z_1.$$

Wówczas koniunkcyjna postać normalna powyższej formuły jest postaci:

$$\begin{gathered} (\neg p_1 \vee \neg t_1 \vee \neg w_1 \vee \neg z_1 \vee \neg q_1) \wedge (\neg p_1 \vee \neg t_1 \vee \neg w_1 \vee z_1 \vee q_1) \wedge \\ \wedge (\neg p_1 \vee \neg t_1 \vee w_1 \vee \neg z_1 \vee q_1) \wedge (\neg p_1 \vee \neg t_1 \vee w_1 \vee z_1 \vee \neg q_1) \wedge \\ \wedge (\neg p_1 \vee t_1 \vee \neg w_1 \vee \neg z_1 \vee q_1) \wedge (\neg p_1 \vee t_1 \vee \neg w_1 \vee z_1 \vee \neg q_1) \wedge \\ \wedge (\neg p_1 \vee t_1 \vee w_1 \vee \neg z_1 \vee \neg q_1) \wedge (\neg p_1 \vee t_1 \vee w_1 \vee z_1 \vee q_1) \wedge \\ \wedge (p_1 \vee \neg t_1 \vee \neg w_1 \vee \neg z_1 \vee q_1) \wedge (p_1 \vee \neg t_1 \vee \neg w_1 \vee z_1 \vee \neg q_1) \wedge \\ \wedge (p_1 \vee \neg t_1 \vee w_1 \vee \neg z_1 \vee \neg q_1) \wedge (p_1 \vee \neg t_1 \vee w_1 \vee z_1 \vee q_1) \wedge \\ \wedge (p_1 \vee t_1 \vee \neg w_1 \vee \neg z_1 \vee \neg q_1) \wedge (p_1 \vee t_1 \vee \neg w_1 \vee z_1 \vee q_1) \wedge \\ \wedge (p_1 \vee t_1 \vee w_1 \vee \neg z_1 \vee q_1) \wedge (p_1 \vee t_1 \vee w_1 \vee z_1 \vee \neg q_1). \end{gathered}$$

Przyglądając się tabeli 5.5, można zauważyć, że konwertując otrzymaną formułę do formatu DIMACS, dla jednego bita danych wejściowych, otrzymamy 16 klauzul.

p	cnf	5	16		
$\neg p_1$	$\neg t_1$	$\neg w_1$	$\neg z_1$	$\neg q_1$	0
$\neg p_1$	$\neg t_1$	$\neg w_1$	z_1	q_1	0
$\neg p_1$	$\neg t_1$	w_1	$\neg z_1$	q_1	0
$\neg p_1$	$\neg t_1$	w_1	z_1	$\neg q_1$	0
$\neg p_1$	t_1	$\neg w_1$	$\neg z_1$	q_1	0
$\neg p_1$	t_1	$\neg w_1$	z_1	$\neg q_1$	0
$\neg p_1$	t_1	w_1	$\neg z_1$	$\neg q_1$	0
$\neg p_1$	t_1	w_1	z_1	q_1	0
p_1	$\neg t_1$	$\neg w_1$	$\neg z_1$	q_1	0
p_1	$\neg t_1$	$\neg w_1$	z_1	$\neg q_1$	0
p_1	$\neg t_1$	w_1	$\neg z_1$	$\neg q_1$	0
p_1	$\neg t_1$	w_1	z_1	q_1	0
p_1	t_1	$\neg w_1$	$\neg z_1$	$\neg q_1$	0
p_1	t_1	$\neg w_1$	z_1	q_1	0
p_1	t_1	w_1	$\neg z_1$	q_1	0
p_1	t_1	w_1	z_1	$\neg q_1$	0

Tabela 5.5: Formuła kodująca przekształcenie 1 bita przez funkcję `MixColumns()`

Proporcjonalnie, poddając działaniu operacji `MixColumns` 1 bajt danych i kodując to przekształcenie do formuły boolowskiej uzyskujemy 128 klauzul. Zapisanie kodowania przekształcenia całego stanu wewnętrznego wymaga 2048 klauzul, należy jednak pamiętać o dodatkowo wygenerowanych klauzulach wprowadzających zmienne pomocnicze. Rysunek 5.9 przedstawia fragment pliku tekstowego zawierającego formułę kodującą opisywane przekształcenie. Proporcjonalnie, poddając działaniu operacji `MixColumns` 1 bajt danych i kodując to przekształcenie do formuły boolowskiej uzyskujemy 128 klauzul. Zapisanie kodowania przekształcenia całego stanu wewnętrznego wymaga 2048 klauzul, należy jednak pamiętać o dodatkowo wygenerowanych klauzulach wprowadzających zmienne pomocnicze.

Rysunek 5.9 przedstawia fragment pliku tekstowego zawierającego formułę kodującą opisywane przekształcenia.

```
-1025 -1153 -977 -1017 -1281 0
-1025 -1153 -977 1017 1281 0
-1025 -1153 977 -1017 1281 0
-1025 -1153 977 1017 -1281 0
-1025 1153 -977 -1017 1281 0
-1025 1153 -977 1017 -1281 0
-1025 1153 977 -1017 -1281 0
-1025 1153 977 1017 1281 0
1025 -1153 -977 -1017 1281 0
1025 -1153 -977 1017 -1281 0
1025 -1153 977 -1017 -1281 0
1025 -1153 977 1017 1281 0
1025 1153 -977 -1017 -1281 0
1025 1153 -977 1017 1281 0
1025 1153 977 -1017 1281 0
1025 1153 977 1017 -1281 0
-1026 -1154 -978 -1018 -1282 0
-1026 -1154 -978 1018 1282 0
                ⋮
1152 1280 -912 952 1408 0
1152 1280 912 -952 1408 0
1152 1280 912 952 -1408 0
```

Rysunek 5.9: Fragment formuły kodującej `MixColumns()`

5.4.3. Przekształcenie `AddRoundKey()`

Mając daną formułę przedstawioną w części 5.3.4., autorka postępując podobnie, jak w przypadku konwersji kodowania boolowskiego sieci Feistela do koniunkcyjnej postaci normalnej przekształciła funkcję `AddRoundKey()` do CNF:

$$\wedge \begin{cases} (\neg u_1 \vee v_1 \vee q_1) \\ (u_1 \vee \neg v_1 \vee q_1) \\ (u_1 \vee v_1 \vee \neg q_1) \\ (\neg u_1 \vee \neg v_1 \vee \neg q_1) \\ (\neg u_2 \vee v_2 \vee q_2) \\ (u_2 \vee \neg v_2 \vee q_2) \\ (u_2 \vee v_2 \vee \neg q_2) \\ (\neg u_2 \vee \neg v_2 \vee \neg q_2) \\ \qquad \vdots \\ (\neg u_{128} \vee v_{128} \vee q_{128}) \\ (u_{128} \vee \neg v_{128} \vee q_{128}) \\ (u_{128} \vee v_{128} \vee \neg q_{128}) \\ (\neg u_{128} \vee \neg v_{128} \vee \neg q_{128}). \end{cases}$$

Na rysunku 5.10 przedstawiono wycinek pliku tekstowego zawierającego formułę kodującą przekształcenie `AddRoundKey()` dla klucza inicjującego.

```
-1 129 577 0
1 -129 577 0
1 129 -577 0
-1 -129 -577 0
-2 130 578 0
2 -130 578 0
2 130 -578 0
-2 -130 -578 0
-3 131 579 0
3 -131 579 0
3 131 -579 0
-3 -131 -579 0
      ⋮
-128 256 704 0
128 -256 704 0
128 256 -704 0
-128 -256 -704 0
```

Rysunek 5.10: Fragment formuły kodującej procedurę `AddRoundKey()`

Postępując zgodnie z opisaną metodą bezpośredniego kodowania boolowskiego autorka rozprawy otrzymała zbiór klauzul odpowiadający działaniu algorytmu AES. Tabela 5.6 przedstawiono rejestr zmiennych zdaniowych użytych do zakodowania pierwszej rundy AES.

Input	$1, ..., 128$	128
Key	$129, ..., 256$	128
Rcon	$257, ..., 576$	320
AddRoundKey_InputKey	$577, ..., 704$	128
SubWord_LastWord_Key	$705, ...736$	32
XorWithRcon_Key	$737, ..., 768$	32
RoundKeyValue	$769, ..., 896$	128
SubBytes_StartOfRound	$897, ..., 1024$	128
MultiBy02	$1025, ..., 1152$	128
MultiBy03	$1153, ..., 1280$	128
MixColumns	$1281, ..., 1408$	128
StartOfRound	$1409, ..., 1536$	128

Tabela 5.6: Rejestr zmiennych zdaniowych dla formuły kodującej pierwszą rundę AES

W tabeli poniżej przedstawiono liczbę zmiennych wykorzystanych do zakodowania poszczególnych rund szyfru AES oraz liczbę wygenerowanych klauzul opisujących działanie szyfru w kolejnych iteracjach.

Liczba rund	Liczba użytych zmiennych	Liczba wygenerowanych klauzul
1	1536	46112
2	2368	91136
3	3200	136160
4	4032	181184
5	4864	226208
6	5696	271232
7	6528	316256
8	7360	361280
9	8192	406304
10	8640	447840

Tabela 5.7: Dane dla szyfru AES ze 128-bitowym kluczem

Autorka wykonała również testy, które wykazały poprawność tej translacji. Testy te dotyczyły poszczególnych operacji używanych w szyfrze AES, jak i całego szyfru AES. Wartości wejściowe i wyjściowe na potrzeby tej weryfikacji zostały określone przy użyciu wektorów testowych podanych w pracy [58].

Gwynne i Kullmann w swojej pracy [67] opisują kodowanie przekształcenia `SubBytes()` oraz mnożenia $\bullet$ wielomianu z ciała $GF(256)$ przez x oraz $x+1$ stosując różne heurystyki. Autorzy przedstawili również translację AES do CNF, ale niestety nie została ona przez nich w pełni przetestowana i nie ma pewności, że jest równoważna działaniu szyfru AES.

5.5. Badania eksperymentalne

Odkąd w 2001 roku AES został przyjęty przez NIST jako standard FIPS-197, sukcesywnie dokonywał się postęp w jego kryptoanalizie. I tak do 2009 roku najlepsze rezultaty to atak na 7 rund AES-128 [63, 60]. Pierwszy z nich był nieznacznie szybszy niż wyczerpujące wyszukiwanie, a złożoność czasowa drugiego wyniosła 2^{120}. Następnie opublikowano ataki na 10 rund AES-192 i na 10 rund AES-256 z odpowiednio 10, 12 oraz 14 rundami [29, 81]. W 2009 roku Biryukov i inni w pracy [34] opublikowali atak z użyciem kluczy pokrewnych (ang. key related attack) na pełną wersję AES-256 o złożoności czasowej 2^{96} dla jednego z 2^{35} kluczy. Pokazali oni również praktyczne ataki na AES-256

[36]. Ponadto Biryukov z zespołem w pracy [35] zaprezentowali pierwszy atak z użyciem kluczy pokrewnych na AES-256, który działa na wszystkie klucze, o lepszej czasowej złożoności $2^{99,5}$, a także pierwszy tego rodzaju atak na pełną wersję szyfru AES-192.

W dalszej części przedstawione zostaną autorskie wyniki eksperymentalne dotyczące SAT kryptoanalizy szyfru AES oraz podjęta zostanie próba wyznaczenia marginesu bezpieczeństwa AES przy zastosowaniu kodowania bezpośredniego i SAT-krypoanalizy. Platformą testową dla tych eksperymentów był komputer przenośny z sześciordzeniowym procesorem Intel Core i7-10750H pracującym z bazową częstotliwością 2,60 GHz wraz z obsługą funkcji Hyper-V. System dysponował 32 GB pamięci RAM i działał pod kontrolą systemu Windows 10 Professional. Do pomiaru czasu wykorzystano wbudowane statystyki dla systemu Linux. Biorąc pod uwagę obliczeniową złożoność SAT, nie oczekiwano, że wszystkie instancje testowe zostaną przetworzone w rozsądnym czasie.

Do przeprowadzenia przedstawionych w tym podrozdziale badań konieczne było opracowanie narzędzia, które generuje formułę boolowską modelującą działanie poszczególnych operacji wykorzystywanych w algorytmie AES, a w konsekwencji modelującej działanie całego szyfru, według zasad przedstawionych w podrozdziale 5.3., dla poszczególnych rund szyfru. Formuły te zostały od razu wygenerowane w odpowiednim formacie, czyli w formacie DIMACS. Niewątpliwą zaletą tego rozwiązania jest brak konieczności przeprowadzania dodatkowych konwersji formuł i związanych z tą operacją czynności sprawdzania poprawności translacji. Opracowane narzędzie zapisuje wygenerowane formuły do plików tekstowych, a te służą jako dane wejściowe do SAT-solverów. Na potrzeby badań szyfru AES opracowano również dodatkowe narzędzia m.in.: skrypty generujące ciągi losowych bitów klucza i tekstu jawnego, które następnie są zapisywane w formacie DIMACS w pliku tekstowym, a także skrypty konwertujące ciąg danych uzyskanych przez SAT-solver do postaci DIMACS.

Do przeprowadzenia analizy kryptograficznej z wybranym tekstem jawnym szyfru AES w wersji ze 128-bitowym kluczem metodą bezpośredniego kodowania boolowskiego wykorzystano SAT-solvery, które w ostatnich latach zajmowały czołowe miejsca m.in. w konkursie *SAT Competition 2020* tj. CaDiCal, Cryptominisat, Kissat oraz Plingeling [134]. Dodatkowo ponownie użyto stabilnego SAT-solvera jakim jest Minisat oraz SAT-solvera lingeling - jednowątkowej wersji Plingeling. Pierwsze podjęte badanie kryptoanalityczne polegało na próbie złamania 1 rundy AES-128. Pomimo ustalenia limitu czasu na poziomie jednego tygodnia, żaden z SAT-solverów nie znalazł w tym czasie wartościowania spełniającego zadaną formułę, a tym samym nie odnalazł szukanych bitów klucza. Zatem dokonano zmiany formuły logicznej poprzez dodanie do niej, w odpo-

wiedniej postaci, kolejno 64, 40, 32, 24 i ostatecznie 16 początkowych bitów klucza i ustawiono limit czasu na 96 godzin. Uzyskane przez poszczególne SAT-solvery wyniki prezentują tabele 5.8 oraz 5.9 (w przypadku braku wyniku po 24 godzinach obliczeń eksperymenty przerwano, co zostało oznaczone przez "–").

Liczba dodanych początkowych bitów klucza	MiniSat 2.2.1 Time[s]	CryptoMiniSat 5.8.0 Time[s]	lingeling bcp2020 Time[s]
64	0,205	0,031	0,112
40	276	15,2	9,67
32	10	289	5514
24	–	29799	–
16	–	6484	332387

Tabela 5.8: Wyniki SAT-solverów uzyskane podczas łamania jednej rundy AES z dodanymi początkowymi bitami klucza

Liczba dodanych początkowych bitów klucza	CaDiCal 1.4.1/2021 Time[s]	Kissat 2.0.0 Time[s]	Plingeling bcp/2020 Time[s]
64	0,127	0,136	0,093
40	10,5	13,1	24,6
32	189	81,8	467
24	1728	6929	6368
16	81780	–	–

Tabela 5.9: Wyniki SAT-solverów uzyskane podczas łamania jednej rundy AES z dodanymi początkowymi bitami klucza

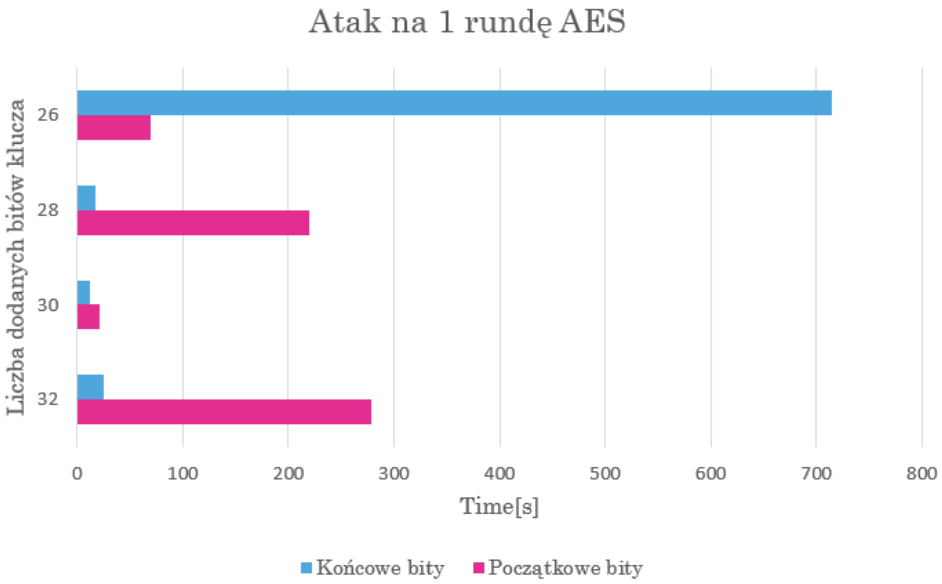
W przypadku, gdy do formuły dodano połowę bitów klucza, wszystkie użyte SAT-solvery szybko obliczyły brakujące bity klucza. Kiedy dodano mniej, bo 40 początkowych bitów klucza, czas potrzebny na znalezienie pozostałych 88 bitów klucza wzrósł wielokrotnie, ale nadal każdy z SAT-solverów sprostał zadaniu. Podobna sytuacja miała miejsce w kolejnej próbie, gdzie dodano 32 początkowe bity klucza. Następny krok polegał na dodaniu 24 bitów klucza. Czas potrzebny poszczególnym SAT-solverom do podania wartościowania zmiennych reprezentujących brakujące bity klucza ponownie znacząco wzrósł, za wyjątkiem SAT-solvera CryptoMiniSat. Dodatkowo dwa spośród używanych SAT-solverów: MiniSat oraz lingeling nie zdołały wykonać obliczeń w zadanym czasie. Mając dane 16 początkowych bitów klucza SAT-solvery na poszukiwanie pozostałych

112 bitów potrzebowały od około 18 godzin - CryptoMiniSAT do ponad 92 godzin - lingeling. Ponadto MiniSat, Kissat oraz Plingeling nie wykonały zadani w założonym czasie.

Kolejne badanie przeprowadzono w dwóch wariantach. W pierwszym wariancie, podobnie jak w poprzednim eksperymencie, do formuły boolowskiej modelującej działanie szyfru AES-128 w pierwszej rundzie, dodano pewną ustaloną liczbę początkowych bitów klucza, a w drugim wariancie końcowych bitów klucza. Do przeprowadzenia każdego z nich użyto SAT-solvera CryptoMiniSat, a limit czasowy ustalono na 12 godzin. W pierwszym etapie badania dodano kolejno 32, 30, 28 i 26 odpowiednio początkowych lub końcowych bitów klucza. Wyniki tego etapu eksperymentu przedstawiono w tabeli 5.10 i na rysunku 5.11.

	Liczba dodanych bitów klucza			
	32	30	28	26
Początkowe bity Time [s]	278	21,5	219	69,2
Końcowe bity Time [s]	25,2	12,2	17	714

Tabela 5.10: Wyniki SAT-solvera CryptoMiniSat uzyskane podczas próby łamania jednej rundy AES cz. 1



Rysunek 5.11: Atak na 1 rundę AES z dodanymi bitami klucza cz.1

W pierwszych trzech przypadkach SAT-solver CryptoMiniSat szybciej obliczył brakujące początkowe bity klucza, a w ostatniej próbie mniej czasu potrze-

bował na obliczenie pozostałych 102 końcowych bitów klucza niż 102 początkowych bitów klucza. Mając dane 26 początkowych bitów klucza SAT-solver potrzebował nieco ponad minutę by odnaleźć brakujące bity klucza, a znając 26 końcowych bitów klucza na obliczenie pozostałych potrzebował niespełna 12 minut.

Przechodząc do drugiego etapu badania dodawano kolejno 24, 22, 18, oraz 16, odpowiednio do wariantu, początkowych lub końcowych bitów klucza. Wyniki tego eksperymentu przedstawiono w tabeli 5.11 oraz na wykresie 5.12. Można zauważyć, że w przeprowadzonym eksperymencie SAT-solver szybciej oblicza pozostałe bity klucza mając dodane początkowe bity. Dodatkowo, w żadnym z dwóch wariantów tego badania, CryptoMiniSat nie obliczył brakujących bitów, a w ostatniej próbie do obliczenia końcowych bitów klucza potrzebował ponad 18 godzin. W wariancie z dodanymi 16 końcowymi bitami klucza SAT-solver nie podał początkowych bitów klucza w ustalonym limicie czasu.

	Liczba dodanych bitów klucza			
	24	22	18	16
Początkowe bity Time [s]	28,9	–	–	6564
Końcowe bity Time [s]	75,6	448	–	–

Tabela 5.11: Wyniki SAT-solvera CryptoMiniSat uzyskane podczas próby łamania jednej rundy AES cz. 2

Rysunek 5.12: Atak na 1 rundę AES z dodanymi klucza cz.2

Wykonano również testy dodając coraz to mniejszą liczbę początkowych (końcowych) bitów klucza, ale SAT-solver CryptoMiniSat nie był w stanie w ciągu 12 godzin podać wartościowania pozostałych bitów klucza.

W kolejnym badaniu podjęto próbę złamania jednej rundy AES-128 dodając bity klucza w ten sposób, że ze środka ciągu bitów klucza usuwano kolejno 100, 102, 104, 106, 108 oraz 110 bitów, pozostawiając w formule kodującej działanie szyfru AES-128 w pierwszej rundzie odpowiednio 28, 26, 24, 22, 20, oraz 18 dodanych bitów.

	Liczba usuniętych środkowych bitów klucza					
	100	102	104	106	108	110
Time [s]	437	135	13566	1774	322	9740

Tabela 5.12: Wyniki SAT-solvera CryptoMiniSat uzyskane podczas łamania jednej rundy AES z dodanymi bitami klucza

Rysunek 5.13: Atak na 1 rundę AES z dodanymi bitami klucza

Limit czasu ustalony dla SAT-solvera CryptoMiniSat na wykonanie obliczeń został ustalony na 12 godzin. Otrzymane wyniki ilustruje tabela 5.12 oraz wykres 5.13. W przypadku ataku na jedną rundę AES-128 z dodanymi, w opisany powyżej sposób, bitami klucza powiódł się nawet w przypadku dodania tylko 18 bitów klucza - SAT-solver pomyślnie znalazł wartościowanie pozostałych bitów

klucza w zadanym czasie. Kontynuując badanie i dodając coraz to mniejszą liczbę bitów klucza (w formacie DIMACS) w formule kodującej działanie AES-128 w pierwszej rundzie, SAT-solver CryptoMiniSat nie był obliczył pozostałych bitów klucza w czasie ustalonym limitem 12 godzin.

Z przeprowadzonych badań wynika, że dodanie do formuły kodującej działanie AES-128 w pierwszej rundzie 16 początkowych bitów klucza powoduje, że SAT-solver obliczył pozostałe bity klucza w czasie krótszym niż 12 godzin. Natomiast dodanie do tej formuły 16 końcowych czy też 8 początkowych i 8 końcowych (wersja z wyciętymi 112 środkowymi bitami klucza) bitów klucza nie pozwoliło SAT-solverowi CryptoMiniSat odnaleźć wartościowania pozostałych bitów klucza w ustalonym limicie czasu.

5.6. Podsumowanie prac

Przeprowadzono testy z których wynika, że wykonane bezpośrednie kodowanie boolowskie szyfru symetrycznego AES-128 jest równoważne jego działaniu. Wykonane eksperymenty pokazały, że algorytm AES w wariancie wykorzystującym 128-bitowy klucz zakodowany do formuły boolowskiej i poddany atakowi z wybranym tekstem jawnym przy użyciu narzędzi, jakimi są SAT-solvery, może zostać złamany w rozsądnym czasie w wersji zredukowanej do jednej rundy z dodanymi co najmniej 16 bitami klucza.

Badawczy wkład własny autorki rozprawy opisany w tym rozdziale jest następujący:

- opracowano bezpośrednie kodowanie boolowskie poszczególnych elementów szyfru AES, jak i całego szyfru,
- wykonano testy potwierdzające równoważność przedstawionego kodowania boolowskiego szyfru AES z jego działaniem,
- opracowano bezpośrednie kodowanie boolowskie dla mnożenia w pierścieniu wielomianów modulo dowolny wielomian 7go stopnia (przekształcenie `MixColumns()`),
- dokonano SAT-kryptoanalizy dla zredukowanego do jednej rundy wariantu AES,
- wykonano obliczenia mające na celu wyznaczenie granicy praktycznej obliczalności SAT-kryptoanalizy dla szyfru AES w obecnych realiach technicznych (software i moce obliczeniowe stosowanych maszyn).

Wszystkie prace programistyczne i obliczeniowe zostały samodzielnie wykonane przez autorkę rozprawy.

Rozdział 6.

Podsumowanie

W niniejszej rozprawie zawarto rozważania dotyczące SAT-kryptoanalizy wybranych szyfrów symetrycznych. Przedstawiona tutaj metoda bazuje na translacji problemu bezpieczeństwa algorytmów kryptograficznych do problemu SAT, która wykorzystuje kodowanie bezpośrednie do formuły boolowskiej. W ramach prowadzonych prac opracowano i opisano formuły kodujące dla trzech różnych szyfrów symetrycznych: Salsa20, AES oraz wybranych modyfikacji DES. Ponadto przeprowadzono serię badań eksperymentalnych łamania brutalnego szyfrów metodą SAT-kryptoanalizy z wybranym tekstem jawnym. Otrzymane wyniki pozwoliły wyznaczyć granice możliwości przeprowadzenia na ww. szyfry ataku brutalnego metodą SAT. Rozprawa potwierdza przypuszczenia, że zastosowanie bezpośredniego kodowania boolowskiego oraz SAT-solverów jest efektywną metodą do badania własności szyfrów symetrycznych i ich modyfikacji.

6.1. Podsumowanie wyników opisanych w rozprawie

Zgodnie z sygnalizowanymi w rozdziale pierwszym celami badawczymi udało się przeprowadzić wiele eksperymentów na powstałych w ramach prac kodowaniach rozważanych szyfrów i/lub ich modyfikacji.

Pełny, własny wkład badawczy autorki opisany w niniejszej rozprawie jest następujący:

- dokonano ponownego badania szyfru DES metodą bezpośredniej SAT-kryptoanalizy dla wielu, w tym nowych, SAT-solverów,
- zbadano wpływ na stabilność pracy SAT-solverów ze względu ma dobór wartościowania bitów tekstu jawnego i klucza,

- opracowano bezpośrednie kodowanie boolowskie liniowych Sboxów,
- dokonano SAT-kryptoanalizy dla nowego kodowania liniowych Sboxów,
- określono praktyczną granicę obliczalności łamania szyfru DES i jego różnych modyfikacji metodą bezpośredniej SAT-kryptoanalizy w obecnych realiach technicznych (software i moce obliczeniowe stosowanych maszyn),
- opracowano bezpośrednie kodowanie boolowskie poszczególnych elementów składowych szyfru Salsa20, jak i całego szyfru,
- dokonano SAT-kryptoanalizy dla Salsa20/2, a także dla Salsa20/4, czyli czterech rund Salsa20 z dodanymi początkowymi, a także z dodanymi końcowymi bitami klucza,
- określono praktyczną granicę obliczalności łamania szyfru Salsa20 metodą bezpośredniej SAT-kryptoanalizy w obecnych realiach technicznych (software i moce obliczeniowe stosowanych maszyn),
- opracowano bezpośrednie kodowanie boolowskie poszczególnych elementów szyfru AES, jak i całego szyfru,
- opracowano bezpośrednie kodowanie boolowskie dla przekształcenia `MixColumns()` dla dowolnego wielomianu pierwotnego,
- dokonano SAT-kryptoanalizy dla zredukowanego do jednej rundy wariantu AES,
- określono praktyczną granicę obliczalności łamania szyfru AES metodą bezpośredniej SAT-kryptoanalizy w obecnych realiach technicznych (software i moce obliczeniowe stosowanych maszyn).

Wszystkie prace programistyczne i obliczeniowe zostały samodzielnie wykonane przez autorkę rozprawy.

Analizując otrzymane wyniki należy zauważyć, że nie można jednoznacznie określić, który z proponowanych dzisiaj solverów SAT jest najefektywniejszy w kryptoanalizie szyfrów symetrycznych. Praca solverów zależy od wielu czynników. Pracują one deterministyczne lub niedeterministyczne. Szczegółowe wyniki zależą od losowanych wektorów bitów tekstu jawnego i klucza, choć w tym przypadku nie wykracza to nigdy poza ramy odpowiedniego rzędu. Można stwierdzić, że udało się określić granicę obliczalności SAT-kryptoanalizy w rozpatrywanych przypadkach.

Po zakodowaniu badanych szyfrów do formuł boolowskich metodą kodowania bezpośredniego we wszystkich przypadkach potwierdzono poprawność wykonanego kodowania przeprowadzając proces szyfrowania za pomocą SAT solvera dla wartości zawartych w odpowiednich normach.

Wykonane eksperymenty pokazały, że algorytm DES zakodowany do formuły boolowskiej i poddany atakowi z wybranym tekstem jawnym przy użyciu narzędzi, jakimi są SAT-solvery, może zostać złamany w rozsądnym czasie w wersji zredukowanej do pięciu rund z dodanymi 6 bitami klucza.

Z przeprowadzonych testów wynika, że szyfr strumieniowy Salsa20 ze 128-bitowym kluczem zakodowany do formuły boolowskiej i poddany atakowi z wybranym tekstem jawnym przy użyciu narzędzi, jakimi są SAT-solvery, może zostać złamany w rozsądnym czasie w wersji zredukowanej do dwóch rund. Dodatkowo przy użyciu wybranych narzędzi SAT możliwe jest obliczenie brakujących bitów klucza w zredukowanym do 4 rund wariancie algorytmu Salsa20 (Salsa20/4) z dodanymi 68 początkowymi bitami klucza w czasie krótszym niż 24 godziny.

Wykonane eksperymenty pokazały, że algorytm AES w wariancie wykorzystującym 128-bitowy klucz zakodowany do formuły boolowskiej i poddany atakowi z wybranym tekstem jawnym przy użyciu narzędzi, jakimi są SAT-solvery, może zostać złamany w rozsądnym czasie w wersji zredukowanej do do jednej rundy z dodanymi co najmniej 16 bitami klucza.

Część opisanych w rozprawie wyników została opublikowana w pracach [125, 126]. Kolejne dwie prace omawiające niepublikowane dotąd wyniki są w przygotowaniu [127, 128].

W eksperymentach wykorzystano wiele solverów SAT, a wyniki zaprezentowano dla kilku wybranych dających najlepsze rezultaty [16, 18, 19]. Dobrym przykładem jest, zaprezentowany na konkursie SAT Competition w 2017 roku, SAT-solver glu_vc - rozwiązanie z rodziny CDCL (Conflict-Driven, Clause-Learning), które wykorzystuje heurystykę VSIDS, wskazując zmienne zdaniowe preferowane do dalszego rozgałęziania drzewa. Do naszych badań wykorzystano również dobrze znany i szeroko stosowany MiniSAT, a także kilka innych, w tym równoległych rozwiązań: Plingeling i Glucose-Syrup.

Podsumowując, przeprowadzone w ramach niniejszej rozprawy badania nad SAT-kryptoanalizą przedstawionych szyfrów stanowią obiecujący punkt wyjścia do realizacji dalszych eksperymentów. Przyszłe badania autorka planuje poświęcić opracowaniu optymalizacji formuł kodowania boolowskiego AES i Salsa20 oraz kryptoanalizie ich zredukowanych wersji z wykorzystaniem udoskonalonych i nowo powstających narzędzi SAT. Dodatkowo zapisanie formuły kodującej

działanie szyfru Salsa20 dla nieparzystej ilości rund i wykonanie badań eksperymentalnych łamania zredukowanej wersji szyfru pomoże uzupełnić obraz wyników SAT-kryptoanalizy tego algorytmu. Planowane jest również użycie do obliczeń maszyn wieloprocesorowych oraz architektur dedykowanych.

Spis rysunków

Spis tabel

Bibliografia

[1] Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, November 2001.

[2] Agrawal M., Kayal N., i Saxena N., *Primes is in P*. In Annals of Mathematics 160, volume 2, pages 781–793, 2004.

[3] Armando A., et.al., *The AVISPA tool for the automated validation of internet security protocols and applications*, In Proc. of 17th International Conference on Computer Aided Verification (CAV'05), vol. 3576 of LNCS, pp. 281–285, Springer-Verlag, 2005.

[4] Armando A., Compagna L., *SAT-based model-checking for security protocols analysis*, International Journal of Information Security, vol. 7(1), pp. 3–32, 2008.

[5] Armando A., and Compagna L., *SATMC: a SAT-based Model Checker for Security Protocols*, In Proc. of JELIA'04, LNAI 3229, Springer Verlag, 2004.

[6] Armando A., Compagna L., Ganty P. SAT-based Model-Checking of Security Protocols, In Proc. of the 12th Intern. FME Symposium (FME 2003), LNCS 2805, Springer, 2003.

[7] Audemard G., Simon L. GLUCOSE 2.1 in the SAT Challenge 2012 . In Proceedings of SAT Competition 2012: Solver and Benchmark Descriptions - SAT Competition 2012, page 21, 2012.

[8] Audemard G., Simon L., *Glucose and syrup in the SAT race 2015*, SAT Race, 2015.

[9] Aumasson J. P., Fischer S., Khazaei S., Meier W., Rechberger Ch., *New Features of Latin Dances, Analysis of Salsa, ChaCha, and Rumba*, Fast Software Encryption, pages 470–488, Springer Berlin Heidelberg, 2008.

[10] Aumasson J. P., *Nowoczesna kryptografia. Praktyczne wprowadzenie do szyfrowania*, Wydawnictwo Naukowe PWN SA, Warszawa 2018.

[11] Bar-On A., Dunkelman O., Keller N., Weizman A., *DLCT, A new tool for differential-linear cryptanalysis*, In Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Proceedings, Part I. LNCS, vol. 11476, pp. 313–342, Springer, 2019.

[12] Bernstein D. J., *Salsa20 specification*, Department of Mathematics, Statistics, and Computer Science (M/C249) The University of Illinois at Chicago, Chicago, IL 60607 – 7045, snuffle@box.cr.yp.to.

[13] Bernstein D.J., Salsa20. Tech. rep., eSTREAM, ECRYPT Stream Cipher Project (2005), http://cr.yp.to/snuffle.html

[14] Bernstein D.J., *The ChaCha family of stream ciphers*, https://cr.yp.to/chacha.html

[15] Le Berre D., Rapicault P., *Dependency management for the eclipse ecosystem, Eclipse p2, metadata and resolution*, in Proceedings of the 1st International Workshop on Open Component Ecosystems, IWOCE '09, pp. 21–30, New York, NY, USA, ACM Press, 2009.

[16] Biere A., *PicoSAT Essentials*. Journal on Satisfiability, Boolean Modeling and Computation (JSAT), vol. 4, pp. 75 – 97, Delft University, 2008.

[17] Biere A., Heule M., H. van Maaren, Walsh T., *Handbook of Satisfiability*, vol. 185 of Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[18] Biere A., *Lingeling, Plingeling, Picosat and Precosat at SAT Race 2010*. Technical Report FMV Reports Series 10/1, Institute for Formal Models and Verification, Johannes Kepler University, 2010.

[19] Biere A., *Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013*. In Proceedings of SAT Competition 2013, vol. B-2013-1 of Department of Comp. Sci. Series of Publications B, pp. 51-52, University of Helsinki, 2013.

[20] Biere, A.: Lingeling, http://fmv.jku.at/lingeling

[21] Biere A., *Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016*. In Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions - SAT Competition 2016, pp. 44–45, 2016.

[22] Biere A., *Lingeling and friends entering the SAT challenge 2012*, Proceedings of SAT Challenge, pp. 33–34, 2012.

[23] Bertoni G., Daemen J., Peeters M., Van Assche G., Keccak sponge function family main document, http: //keccak.noekeon.org/Keccak-main-2.1.pdf

[24] Biham E., Shamir A.,*Differential cryptanalysis of DES-like cryptosystems*, Journal of Cryptology, vol. 4, pp. 3-72, 1991.

[25] Biham E., Shamir A., *Differential Cryptanalysis of the Data Encryption Standard*, Springer, New York, 1993.

[26] Biham E., *New Types of Cryptoanalytic Attacks Using Related Keys*, Journal of Cryptology, vol. 7, pp. 229-246, 1994.

[27] Biham E., Biryukov A., Shamir A., *Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials*, In: Stern J. (eds) Advances in Cryptology — EUROCRYPT '99, Lecture Notes in Computer Science, vol. 1592, pp. 12–23, Springer, Berlin, Heidelberg, 1999.

[28] Biham E., Dunkelman O., Keller N., *The Rectangle Attack — Rectangling the Serpent*, In: Pfitzmann B. (eds) Advances in Cryptology — EUROCRYPT 2001, Lecture Notes in Computer Science, vol. 2045, pp. 340-357, Springer, Berlin, Heidelberg, 2001.

[29] Biham E., Dunkelman O., Keller N., *Related-key boomerang and rectangle attacks*, In EUROCRYPT'05, volume 3494 of LNCS, pages 507–525, Springer, 2005.

[30] Biham E., Dunkelman O., Keller N., *A Related-Key Rectangle Attack on the Full KASUMI*, In: Roy B. (eds) Advances in Cryptology – ASIACRYPT 2005, Lecture Notes in Computer Science, vol. 3788, pp. 443-461, Springer, Berlin, Heidelberg, 2005.

[31] Biham E., Dunkelman O., Keller N., *Related-Key Impossible Differential Attacks on 8-Round AES-192*, In: Pointcheval D. (eds) Topics in Cryptology – CT-RSA 2006. CT-RSA 2006, Lecture Notes in Computer Science, vol 3860, pp. 21-33, Berlin, Heidelberg, 2006.

[32] Biryukov A., Wagner D., *Slide attacks*, In Proceedings of Fast Software Encryption – FSE'99, Lecture Notes in Computer Science, vol. 1636, pp. 245–259, Springer-Verlag, 1999.

[33] Biryukov A., Wagner D., *Advanced Slide Attacks*, EUROCRYPT 2000, Lecture Notes in Computer Science, vol. 1807, pp. 589-606, Springer, 2000.

[34] Biryukov A., Khovratovich D., Nikolić I., *Distinguisher and Related-Key Attack on the Full AES-256*, In: Halevi S. (eds) Advances in Cryptology - CRYPTO 2009. CRYPTO 2009, Lecture Notes in Computer Science, vol. 5677, pp. 231-249, Springer, Berlin, Heidelberg, 2009.

[35] Biryukov A., Khovratovich D., *Related-Key Cryptanalysis of the Full AES-192 and AES-256*, In: Matsui M. (eds) Advances in Cryptology – ASIACRYPT 2009, Lecture str. 223-231, Notes in Computer Science, vol. 5912, pp. 1-18, Springer, Berlin, Heidelberg, 2009.

[36] Biryukov A., Dunkelman O., Keller N., Khovratovich D., Shamir A.,*Key Recovery Attacks of Practical Complexity on AES Variants with up to10 Rounds*, EUROCRYPT, Springer, 2010.

[37] Biryukov A., *Impossible Differential Attack*, In: van Tilborg H.C.A. (eds) Encyclopedia of Cryptography and Security, Springer, Boston, MA, 2005.

[38] Blondeau C., Leander G., Nyberg K., *Differential-Linear Cryptanalysis Revisited*, Journal of Cryptology, vol. 30, pp. 859–888, 2017.

[39] Choudhuri A. R. and Maitra S., *Signifcantly improved multi-bit differentials for reduced round Salsa and ChaCha*, IACR Trans. Symmetric Cryptol., vol. 2, pp. 261-287, Feb. 2017.

[40] Chowaniec M., Kurkowski M., Mazur M., *New Results in Direct SAT-Based Cryptanalysis of DES-Like Ciphers*. In Proc. of ACS'18. AISC, vol. 889, pp. 282-294, Springer, Cham, 2018.

[41] Cook S.A., *The complexity of theorem-proving procedures*, Proceedings of the third annual ACM symposium on Theory of computing, Stoc., pp. 151–158, 1971.

[42] Cook, S.A., Mitchell, D.G.: Finding hard instances of the satisfiability problem: A survey, pp. 1–17. American Mathematical Society, 1997.

[43] Courtois, N.T., Pieprzyk, J.: Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 267–287. Springer, Heidelberg, 2002.

[44] Courtois N. T., Bard G. V., *Algebraic cryptanalysis of the data encryption standard*, Cryptology ePrint Archive, 2006.

[45] Crowley P., *Truncated differential cryptanalysis of five rounds of Salsa20*, IACR Cryptology ePrint Archive, 2005.

[46] Daemen J. and Rijmen V., *AES submission document on Rijndael, Version 2*, September 1999.
http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf

[47] Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Information Security and Cryptography, Springer (2002)

[48] Davis M., Logemann G., Loveland D., *A machine program for theorem proving*, Communications of the ACM, vol. 5, no. 7, pp. 394–397, 1962.

[49] Dey S., Sarkar S., *Improved analysis for reduced round Salsa and Chacha*, Discrete Appl. Math, vol. 227, pp. 58-69, Aug. 2017.

[50] Dembinski P., Janowska A., Janowski P., Penczek W., Półrola A., Szreter M., Wozna B. and Zbrzezny A., *VerICS: A Tool for Verifying Timed Automata and Estelle Specifications*, In Proc. of the 9th Int. Conf. TACAS'03, vol. 2619 of LNCS, pp. 278–283 Springer-Verlag, 2003.

[51] Dhar Dwivedi A., Kloucek M., Morawiecki P., Ivica N., Pieprzyk J., Wójtowicz S., *SAT-based Cryptanalysis of Authenticated Ciphers from the CAESAR Competition*, Proceedings of SECRYPT 2017, volume 4, Madrid, Spain, 2017.

[52] *DES Modes of Operation [includes Change Notice of May 31, 1996]*, Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 1980.

[53] Diffie W., Hellman M., *New Directions in Cryptography*, IEEE Transactions on Information Theory, IT-22(6), pp. 644-665, 1976.

[54] Ding L., *Improved Related-Cipher Attack on Salsa20 Stream Cipher*, IEEE Access, vol. 7, pp. 30197-30202, 2019.

[55] Dudek P., Kurkowski M., Srebrny M., *Towards Parallel Direct SAT-based Cryptanalysis*, in PPAM'11 Proc., pp. 266-275, vol. 7203 of LNCS, Springer, 2012.

[56] https://www.ecrypt.eu.org/stream/project.html

[57] Fiorini, C., Martinelli, E., Massacci, F.: How to fake an RSA signature by encoding modular root finding as a SAT problem. Discrete Applied Mathematics 130, pp. 101–127, 2003.

[58] FIPS PUB 197, *Advanced Encryption Standard (AES)*, National Institute of Standards and Technology, U.S. Department of Commerce, November 2001. http//csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[59] Feistel H., *Block Cipher Cryptographic System*, 1971 dostępny `https//patents.google.com/patent/US3798359A/en`.

[60] Ferguson N., Kelsey J., Lucks S., Schneier B., Stay M., Wagner D., Whiting D.,*Improved cryptanalysis of Rijndael*, In FSE'00, volume 1978 of LNCS, pages 213–230. Springer, 2000.

[61] Fischer S., Meier W., Berbain C., Biasse J.-F., Robshaw M. J. B., *Non-randomness in eSTREAM candidates salsa20 and TSC-4*, Proc. INDOCRYPT, Kolkata, India, pp. 2-16, 2006.

[62] Gąsiecki A., Misztal M., *Zastosowanie technik algebraicznych w kryptoanalizie różnicowej na przykładzie szyfru blokowego DES*, Biuletyn WAT Vol LX Nr 3; 2011.

[63] Gilbert H., Minier M., *A collision attack on 7 rounds of Rijndael*, In AES Candidate Conference, pages 230–241, 2000.

[64] Gorski M., Lucks S., *New related-key boomerang attacks on AES*, In Chowdhury, D.R. Rijmen, V., Das, A., eds., INDOCRYPT, Lecture Notes in Computer Science, vol. 5356, pp. 266-278, Springer, 2008.

[65] Grajek M., *Enigma. Bliżej prawdy*, Dom Wydawniczy Rebis, 2007.

[66] Grygiel J., Kurkowski M., *Wybrane elementy logiki, teorii mnogości i teorii grafów*, Oficyna Wydawnicza EU, Warszawa, 2015.

[67] Gwynne M., Kullmann O., *Attacking AES via SAT*, 2010. https://www.academia.edu/2594954/Attacking_AES_via_SAT [dostęp: 08.11.2021 r.].

[68] Hawkes P., *Differential-Linear Weak Key Classes of IDEA*, EUROCRYPT '98, pp. 112-126, 1998.

[69] Homsirikamol E., Morawiecki P., Rogawski M., Srebrny M., Security Margin Evaluation of SHA-3 Contest Finalists through SAT-Based Attacks. CISIM 2012,

[70] Hernandez-Castro J. C., Tapiador J. M. E., Quisquater J.-J., *On the Salsa20 core function*, Proc. FSE, Lausanne, Switzerland, pp. 462-469, 2008.

[71] Hong S., Kim J., Lee S., Preneel B., *Related-Key Rectangle Attacks on Reduced Versions of SHACAL-1 and AES-192*, In Henri Gilbert and Helena Handschuh, editors, FSE, Lecture Notes in Computer Science, vol. 3557, pp. 368-383, Springer, 2005.

[72] Jakubowska G., Penczek W., *Is your security protocol on time?*, In Proc. of the Int. Symp. on Fundamentals of Software Engineering (FSEN'07), vol. 4767 of LNCS, pp. 65–80. Springer-Verlag, 2007.

[73] Jakubowska G., Penczek W., Srebrny M., *Verifying security protocols with timestamps via translation to timed automata*, In Proc. of the International Workshop on Concurrency, Specification and Programming (CS&P'05), pp. 100–115. Warsaw University Press, 2005.

[74] Jovanović, D., Janićić, P.: Logical Analysis of Hash Functions. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 200–215. Springer, Heidelberg, 2005.

[75] Kahn D., *Łamacze kodów. Historia kryptologii*, Wydawnictwo Zysk i S-ka, 2019.

[76] Kaivola R. i inni, *Replacing Testing with Formal Verification in Intel Core I7 Processor Execution Engine Validation*, In Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09, pp. 414–429, Berlin, Heidelberg, 2009. Springer-Verlag.

[77] Karbowski M., *Podstawy kryptografii, Wydanie III*, Wydawnictwo Helion, Gliwice, 2014.

[78] Kerckhoffs A., *La cryptographie militaire*, Journal des sciences militaires, vol. IX, pp. 5–38, Jan. 1883, pp. 161–191, Feb. 1883.

[79] Kelsey J., Schneier B., Wagner D., *Key-Schedule Cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES*, Advances in Cryptology, CRYPTO '96 Proceedings, Springer-Verlag, pp. 237-251, 1996.

[80] Kim J., Hong S., Preneel B., Biham E., Dunkelman O., Keller N., *Related-Key Boomerang and Rectangle Attacks*, IACR, 2010/019, http://eprint.iacr.org/2010/019, 2010.

[81] Kim J., Hong S., Preneel B., *Related-key rectangle attacks on reduced AES-192 and AES-256*, In FSE'07, volume 4593 of LNCS, pages 225–241, Springer, 2007.

[82] Knudsen L.R., Mathiassen J.E., *On the Role of Key Schedules in Attacks on Iterated Ciphers*, In: Samarati P., et.al. (eds) Computer Security – ESORICS 2004, Lecture Notes in Computer Science, vol 3193, pp. 322-334, Springer, Berlin, Heidelberg, 2004.

[83] Kościelny C., Kurkowski M., Srebrny M., *Kryptografia. Teoretyczne podstawy i praktyczne zastosowania*, Wydawnictwo Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych, Warszawa, 2009.

[84] Kościelny C.,Kurkowski M., Srebrny M., *Modern Cryptography Primer*, Springer Verlag, 2013.

[85] Kurkowski M., *Formalne metody weryfikacji własności protokołów zabezpieczających w sieciach komputerowych*, Akademicka Oficyna Wydawnicza EXIT, Warszawa, 2013.

[86] Kurkowski M., Siedlecka-Lamch O., Dudek P., *Using Backward Induction Techniques in (Timed) Security Protocols Verification*. In Proc. of CISIM'2013, pp. 265-276, 2013.

[87] Kurkowski M., Penczek W., *Applying Timed Automata to Model Checking of Security Protocols. Handbook of Finite State Based Models and Applications*, pp. 223-254, 2012.

[88] Kurkowski M., Penczek W., *Verifying Security Protocols Modelled by Networks of Automata*. Fundam. Informaticae 79(3-4), pp. 453-471, 2007.

[89] Kurkowski M., Penczek W., *Timed Automata Based Model Checking of Timed Security Protocols*. Fundam. Informaticae 93(1-3), pp. 245-259, 2009.

[90] Lafitte F., Lerman L., Markowitch O.,van Heule D., *SAT-based cryptanalysis of ACORN*, IACR Cryptology ePrint Archive, pp. 521, vol. 2016, 2016.

[91] Lafitte F., i in., *Applications of SAT Solversin Cryptanalysis, Finding Weak Keys and Preimages*, JSAT, vol. 9, pp. 1–25, 2014.

[92] Langford S.K., Hellman M.E., *Differential-Linear Cryptanalysis*, In: Desmedt Y.G. (eds) Advances in Cryptology — CRYPTO '94, Lecture Notes in Computer Science, vol. 839, pp. 17-25, Springer, Berlin, Heidelberg, 1994.

[93] Maitra S., *Chosen IV cryptanalysis on reduced round ChaCha and Salsa*, Discrete Appl. Math, vol. 208, pp. 88-97, Jul. 2016.

[94] Maitra S., Paul G., Meier W., *Salsa20 Cryptanalysis, New Moves and Revisiting Old Styles*, http://eprint.iacr.org/2015/217

[95] Matsui M., Yamagishi A. (1993). A New Method for Known Plaintext Attack of FEAL Cipher. In: Rueppel, R.A. (eds) Advances in Cryptology — EUROCRYPT' 92. EUROCRYPT 1992. Lecture Notes in Computer Science, vol 658. Springer, Berlin, Heidelberg, 1992.

[96] Matsui M., *The first experimental cryptanalysis of the Data Encryption Standard*. In Y. Desmedt, editor, CRYPTO, vol. 839 of LNCS, pp. 1–11. Springer, 1994.

[97] Massacci F., *Using Walk-SAT and Rel-SAT for cryptographic key search*. In T. Dean, editor, IJCAI, pages 290–295. Morgan Kaufmann, 1999.

[98] Massacci F., Marraro L., *Logical Cryptanalysis as a SAT Problem*, Journal of Automated Reasoning, pp. 165 – 203, vol. 24, 2000.

[99] Matsui M., *Linear Cryptanalysis Method for DES Cipher*, In: Helleseth T. (eds) Advances in Cryptology — EUROCRYPT '93, Lecture Notes in Computer Science, vol. 765, pp. 386-397, Springer, Berlin, Heidelberg, 1994.

[100] Mazur Sz., *Enigma. Poznańskie ślady*, Wydawnictwo Miejskie Posnania, 2021.

[101] Meissner A., *Sztuczna inteligencja. Problem spełnialności (SAT)*, `http://users.man.poznan.pl/ameis/Si-SAT.pdf`

[102] Menezes A. J., Oorschot P. C., Vanstone S. A., *Kryptografia stosowana*, WNT, Warszawa, 2005.

[103] Mironov I., Zhang L., *Applications of SAT solvers to cryptanalysis of hash functions*, In SAT, volume 4121 of Lecture Notes in Computer Science, pp. 102–115. Springer-Verlag.

[104] Mouha N., Preneel B., *A Proof that the ARX Cipher Salsa20 is Secure against Differential Cryptanalysis*,Cryptology ePrint Archive, Report 2013/328, 2013.

[105] Morawiecki P., Srebrny M., *A SAT-based preimage analysis of reduced Keccak hash functions*, Information Processing Letters, volume 113(10-11), 2013.

[106] de Moura L., Bjorner N., Bugs, Moles oraz Skeletons, *Symbolic Reasoning for Software Development*, pp. 400–411. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[107] Needham R., Schroeder M., Using Encryption for Authentication in large networks of computers, Communications ofthe ACM, vol. 21(12), pp. 993-999, 1978.

[108] Niewiadomski A., Kacprzak M., Kurpiewski D., Knapik M., Penczek W., Jamroga W., *MsATL, : A Tool for SAT-Based ATL Satisfiability Checking*. In Proc. of AAMAS 2020, pp. 2111-2113, 2020.

[109] Niewiadomski A., Świtalski P., Sidoruk T., Penczek W., *Applying Modern SAT-solvers to Solving Hard Problems*. Fundam. Informaticae 165(3-4), pp. 321-344, 2019.

[110] Niewiadomski A., Penczek W., Pólrola A., Szreter M., Zbrzezny A., *Towards Automatic Composition of Web Services, SAT-Based Concretisation of Abstract Scenarios*. Fundam. Informaticae 120(2), pp. 181-203, 2012.

[111] Penczek W., Pólrola A., Zbrzezny A., SAT-Based (Parametric) Reachability for a Class of Distributed Time Petri Nets. Trans. Petri Nets Other Model. Concurr. 4, pp. 72-97, 2010.

[112] Penczek W., Szreter M., SAT-based Unbounded Model Checking of Timed Automata. Fundam. Informaticae 85(1-4), pp. 425-440, 2008.

[113] Pieprzyk J., Hardjono T., Seberry J., *Teoria bezpieczeństwa systemów komputerowych*, Wydawnictwo HELION, Gliwice, 2003.

[114] PrecoSAT, http://fmv.jku.at/precosat/.

[115] Selianinau M., Kamiński P. *Schemat zabezpieczenia wymiany informacji pomiędzy trzema użytkownikami kryptograficznym systemem RSA*, Akademia im. Jana Długosza w Częstochowie, z. VII, p. 98., 2012.

[116] Shi Z., Zhang B., Feng D., Wu W., *Improved key recovery attacks on reduced-round Salsa20 and ChaCha*, in Proc. ICISC, Seoul, South Korea, pp. 337-351, 2012.

[117] Soboń A., Kurkowski M., Stachowiak S., *Towards Complete SAT-based Cryptanalysis of RC5 Cipher*, in Proc. of. 2019 IEEE 15th International Scientific Conference on Informatics, pp. 397-402, IEEE Press, 2019.

[118] Soboń A., Kurkowski M., and S. Stachowiak. *Complete SAT Based Cryptanalysis of RC5 Cipher*. Journal of Information and Organizational Sciences, vol. 44, no. 2, Dec. 2020.

[119] Soos M.: CryptoMiniSat 2.5.0. In: SAT Race Competitive Event Booklet (July 2010), http://www.msoos.org/cryptominisat2

[120] Soos M., The CryptoMiniSat 5 set of solvers at SAT Competition 2016. In Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions - SAT Competition 2016, p. 28, 2016.

[121] Soos M.: Grain of Salt — An Automated Way to Test Stream Ciphers through SAT Solvers. In: Workshop on Tools for Cryptanalysis, 2010.

[122] Soos M., Nohl K., and Castelluccia C., *Extending SAT Solvers to Cryptographic Problems*, Theory and Applications of Satisfiability Testing SAT 2009, In Proc. of 12th Int. Conf., SAT 2009, pp. 244 – 257, 2009.

[123] Sörensson N., Een N., *Minisat v1.13-a SAT solver with conflict-clause minimization*, International Conference on Theory and Applications of Satisfiability Testing, 2005.

[124] Srebrny M., Srebrny M., Stępień L., *SAT as a Programming Environment for Linear Algebra and Cryptanalysis*. In Proc. of ISAIM 2008.

[125] Stachowiak S., Kurkowski M., and Soboń A., *SAT vs. Substitution Boxes of DES like Ciphers*, 2021 IEEE 30th International Conference on Enabling Technologies, Infrastructure for Collaborative Enterprises (WETICE), pp. 113-118, 2021.

[126] Stachowiak S., Kurkowski M., and Soboń A., *SAT-Based Cryptanalysis of Salsa20 Cipher*, Progress in Image Processing, Pattern Recognition and Communication Systems, pp. 252-266, Springer International Publishing, 2021.

[127] Stachowiak S., Kurkowski M., and Soboń A., *SAT vs. Advanced Encryption Standard*, praca w przygotowaniu.

[128] Stachowiak S., Kurkowski M., and Soboń A., *New boolean encoding of some AES cryptosystem mathematical aspects*, praca w przygotowaniu.

[129] Stinson D. R., *Kryptogtafia. W teorii i praktyce*, Wydawnictwo Naukowo-Techniczne, Warszawa, 2005.

[130] Stoffelen K., Optimizing S-Box Implementations for Several Criteria Using SAT Solvers. In Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers, pp. 140–160, 2016.

[131] Strumph Wojtkiewicz S., Sekret Enigmy, Wydawnictwo Iskry, 1978.

[132] Swetoniusz Trankwillus G., *Żywoty cezarów*. Wrocław, Zakład Narodowy im. Ossolińskich, 1987.

[133] Szymoniak S., et.al., *SAT and SMT-Based Verification of Security Protocols Including Time Aspects*. Sensors 21(9), p. 3055, 2021.

[134] The International SAT Competitions web page `https://satcompetition.github.io`

[135] Tsunoo Y., Saito T., Kubo H., Suzaki T., Nakashima H., *Differential cryptanalysis of Salsa20/8*, 2007.

[136] Wagner D., *The Boomerang Attack*, In: Knudsen L. (eds) Fast Software Encryption. FSE 1999, Lecture Notes in Computer Science, vol. 1636, pp. 156-170, Springer, Berlin, Heidelberg, 1999.

[137] `https://pl.wikipedia.org/wiki/Dane_tabelaryczne_algorytmu_DES` [dostęp: 1.11.2021].

[138] Wobst R., *Kryptologia. Budowa i łamanie zabezpieczeń*, wyd. Read Me, 2002.

[139] Woźna B., Zbrzezny A., Penczek W., *Checking Reachability Properties for Timed Automata via SAT*. Fundam. Informaticae 55(2), pp. 223-241, 2003.

[140] Zbrzezny A. M., Szymoniak S., Kurkowski M., *Practical Approach in Verification of Security Systems Using Satisfiability Modulo Theories.* Log. J. IGPL 30(2), pp. 289-300, 2022.

[141] Zbrzezny A., *Improvements in SAT-based Reachability Analysis for Timed Automata*, Fundamenta Informaticae, vol. 60(1-4), pp. 417-434, IOS Press, 2004.

[142] Zbrzezny A., *SAT-based Reachability Checking for Timed Automata with Diagonal Constraints*, Fundamenta Informaticae, vol. 67(1-3), pp. 303-322, IOS Press, 2005.

[143] Zbrzezny A. M., Zbrzezny A., Szymoniak S., Siedlecka-Lamch O., Kurkowski M., VerSecTis - An Agent based Model Checker for Security Protocols. AAMAS 2020, pp. 2123-2125, 2020.

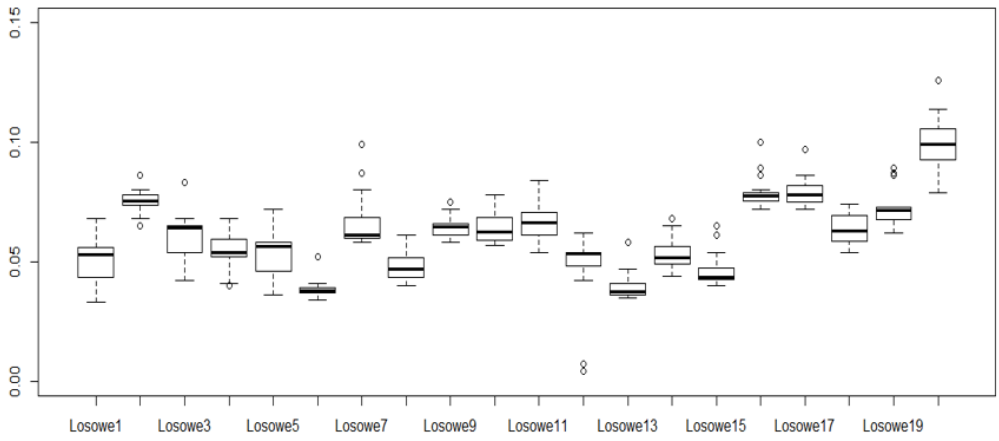
[144] Zhang W., Zhang L., Wu W., Feng D., *Related-Key Differential-Linear Attacks on Reduced AES-192*, In: Srinathan K., Rangan C.P., Yung M., (eds) Progress in str. 231 / 231 Cryptology – INDOCRYPT 2007, Lecture Notes in Computer Science, vol. 4859, pp. 73-85, Springer, Berlin, Heidelberg, 2007.

[145] Zhang J., Wu W., Zheng Y., *Security of SM4 Against (Related-Key) Differential Cryptanalysis*, In: Bao F., Chen L., Deng R., Wang G., (eds) Information Security Practice and Experience, ISPEC 2016, Lecture Notes in Computer Science, vol. 10060, pp. 65-78, Springer, Cham, 2016.

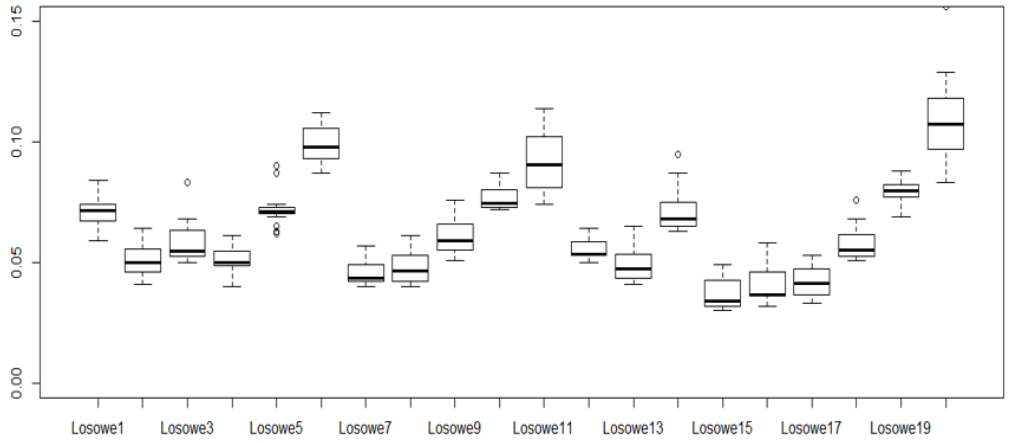
Dodatek A

Ten dodatek zawiera uzyskane wyniki eksperymentu polegającego na tym, że dla każdego solvera (MiniSat, Glu_vc, Glucosen_syrup, Plingeling) wybrano próbkę losową składającą się z 64 bitów tekstu jawnego i próbkę 64 bitowego tekstu reprezentującego klucz. Dla każdego rozważanego solvera pobrano 20 różnych tak powstałych próbek losowych a następnie każdą próbkę replikowano 20 razy dla 3 i 4 rund szyfru DES. Przedstawiono boxploty rozkładów czasów złamania szyfrów dla poszczególnych czterech solverow i badanych próbek dla 3 i 4 rund DES. Dodatkowo podano też podstawowe statystyki dla poszczególnych 20 próbek (3 rundy: Tabele A1-A4, 4 rundy: A7-A10) wraz ze współczynnikami zmienności (Tabele A5-A6 dla 3 rund, Tabele A11-A12 dla 4 rund).

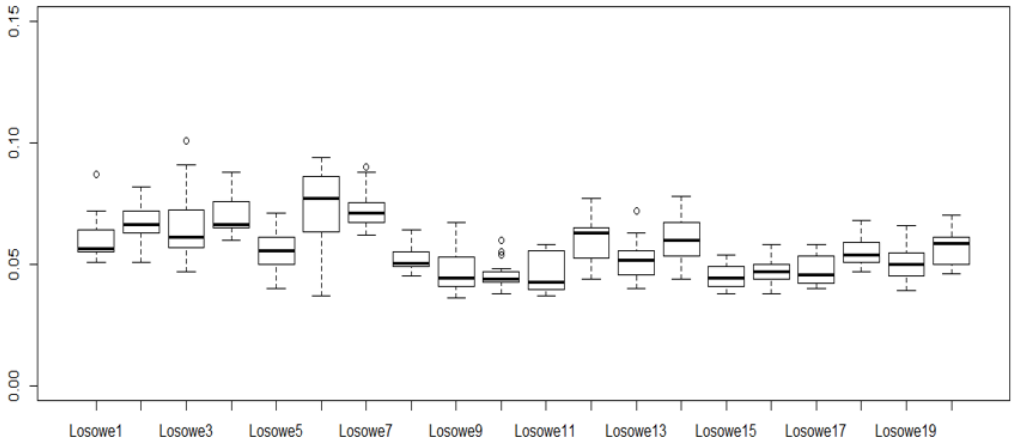
A1. 3 rundy DES (64-bitowy klucz)



Rysunek A1: Boxploty dla 3MiniSat



Rysunek A2: Boxploty dla 3Glu_vc



Rysunek A3: Boxploty dla 3Glucose_syrup

Losowe1	Losowe2	Losowe3	Losowe4
Min. :0.03300	Min. :0.06500	Min. :0.04200	Min. :0.04000
1st Qu.:0.04375	1st Qu.:0.07375	1st Qu.:0.05400	1st Qu.:0.05200
Median :0.05300	Median :0.07550	Median :0.06400	Median :0.05400
Mean :0.05130	Mean :0.07545	Mean :0.06115	Mean :0.05485
3rd Qu.:0.05600	3rd Qu.:0.07800	3rd Qu.:0.06500	3rd Qu.:0.05875
Max. :0.06800	Max. :0.08600	Max. :0.08300	Max. :0.06800

Losowe5	Losowe6	Losowe7	Losowe8
Min. :0.03600	Min. :0.03400	Min. :0.05800	Min. :0.04000
1st Qu.:0.04600	1st Qu.:0.03700	1st Qu.:0.06000	1st Qu.:0.04375
Median :0.05650	Median :0.03800	Median :0.06100	Median :0.04700
Mean :0.05465	Mean :0.03865	Mean :0.06635	Mean :0.04845
3rd Qu.:0.05800	3rd Qu.:0.03900	3rd Qu.:0.06725	3rd Qu.:0.05125
Max. :0.07200	Max. :0.05200	Max. :0.09900	Max. :0.06100

Losowe9	Losowe10	Losowe11	Losowe12
Min. :0.0580	Min. :0.05700	Min. :0.05400	Min. :0.0040
1st Qu.:0.0610	1st Qu.:0.05900	1st Qu.:0.06150	1st Qu.:0.0485
Median :0.0645	Median :0.06250	Median :0.06650	Median :0.0535
Mean :0.0645	Mean :0.06390	Mean :0.06700	Mean :0.0458
3rd Qu.:0.0660	3rd Qu.:0.06825	3rd Qu.:0.07025	3rd Qu.:0.0540
Max. :0.0750	Max. :0.07800	Max. :0.08400	Max. :0.0620

Losowe13	Losowe14	Losowe15	Losowe16
Min. :0.0350	Min. :0.04400	Min. :0.04000	Min. :0.07200
1st Qu.:0.0360	1st Qu.:0.04900	1st Qu.:0.04275	1st Qu.:0.07575
Median :0.0375	Median :0.05150	Median :0.04350	Median :0.07750
Mean :0.0397	Mean :0.05370	Mean :0.04630	Mean :0.07890
3rd Qu.:0.0410	3rd Qu.:0.05575	3rd Qu.:0.04725	3rd Qu.:0.07900
Max. :0.0580	Max. :0.06800	Max. :0.06500	Max. :0.10000

Losowe17	Losowe18	Losowe19	Losowe20
Min. :0.07200	Min. :0.05400	Min. :0.06200	Min. :0.07900
1st Qu.:0.07500	1st Qu.:0.05875	1st Qu.:0.06775	1st Qu.:0.09275
Median :0.07800	Median :0.06300	Median :0.07150	Median :0.09900
Mean :0.07915	Mean :0.06425	Mean :0.10470	Mean :0.09875
3rd Qu.:0.08100	3rd Qu.:0.06925	3rd Qu.:0.07300	3rd Qu.:0.10525
Max. :0.09700	Max. :0.07400	Max. :0.72400	Max. :0.12600

Tabela A1: **3MiniSat** (charakterystyki czasu złamania klucza dla poszczególnych prób losowych, min, 1stQu-pierwszy kwartyl, 3rdQu-trzeci kwartyl)

Losowe1	Losowe2	Losowe3	Losowe4
Min. :0.0590	Min. :0.04100	Min. :0.05000	Min. :0.04000
1st Qu.:0.0680	1st Qu.:0.04600	1st Qu.:0.05275	1st Qu.:0.04875
Median :0.0715	Median :0.05000	Median :0.05450	Median :0.05000
Mean :0.0707	Mean :0.05055	Mean :0.05840	Mean :0.05130
3rd Qu.:0.0740	3rd Qu.:0.05525	3rd Qu.:0.06325	3rd Qu.:0.05425
Max. :0.0840	Max. :0.06400	Max. :0.08300	Max. :0.06100

Losowe5	Losowe6	Losowe7	Losowe8
Min. :0.0620	Min. :0.0870	Min. :0.04000	Min. :0.0400
1st Qu.:0.0700	1st Qu.:0.0930	1st Qu.:0.04200	1st Qu.:0.0420
Median :0.0710	Median :0.0980	Median :0.04350	Median :0.0465
Mean :0.0719	Mean :0.0988	Mean :0.04565	Mean :0.0474
3rd Qu.:0.0730	3rd Qu.:0.1052	3rd Qu.:0.04850	3rd Qu.:0.0525
Max. :0.0900	Max. :0.1120	Max. :0.05700	Max. :0.0610

Losowe9	Losowe10	Losowe11	Losowe12
Min. :0.0510	Min. :0.07200	Min. :0.07400	Min. :0.05000
1st Qu.:0.0550	1st Qu.:0.07300	1st Qu.:0.08250	1st Qu.:0.05300
Median :0.0590	Median :0.07450	Median :0.09050	Median :0.05350
Mean :0.0605	Mean :0.07705	Mean :0.09165	Mean :0.05550
3rd Qu.:0.0655	3rd Qu.:0.07950	3rd Qu.:0.10150	3rd Qu.:0.05775
Max. :0.0760	Max. :0.08700	Max. :0.11400	Max. :0.06400

Losowe13	Losowe14	Losowe15	Losowe16
Min. :0.04100	Min. :0.0630	Min. :0.03000	Min. :0.0320
1st Qu.:0.04375	1st Qu.:0.0650	1st Qu.:0.03200	1st Qu.:0.0360
Median :0.04750	Median :0.0680	Median :0.03400	Median :0.0365
Mean :0.04890	Mean :0.0712	Mean :0.03680	Mean :0.0403
3rd Qu.:0.05275	3rd Qu.:0.0740	3rd Qu.:0.04225	3rd Qu.:0.0445
Max. :0.06500	Max. :0.0950	Max. :0.04900	Max. :0.0580

Losowe17	Losowe18	Losowe19	Losowe20
Min. :0.03300	Min. :0.05100	Min. :0.06900	Min. :0.0830
1st Qu.:0.03675	1st Qu.:0.05275	1st Qu.:0.07700	1st Qu.:0.0970
Median :0.04150	Median :0.05500	Median :0.07950	Median :0.1075
Mean :0.04220	Mean :0.05765	Mean :0.07940	Mean :0.1076
3rd Qu.:0.04725	3rd Qu.:0.06075	3rd Qu.:0.08225	3rd Qu.:0.1165
Max. :0.05300	Max. :0.07600	Max. :0.08800	Max. :0.1560

Tabela A2: **3Glu_vc** (charakterystyki czasu złamania klucza dla poszczególnych prób losowych, min, 1stQu-pierwszy kwartyl, 3rdQu-trzeci kwartyl)

Losowe1	Losowe2	Losowe3	Losowe4
Min. :0.0510	Min. :0.0510	Min. :0.04700	Min. :0.0600
1st Qu.:0.0550	1st Qu.:0.0635	1st Qu.:0.05700	1st Qu.:0.0650
Median :0.0565	Median :0.0665	Median :0.06100	Median :0.0665
Mean :0.0601	Mean :0.0674	Mean :0.06555	Mean :0.0701
3rd Qu.:0.0640	3rd Qu.:0.0705	3rd Qu.:0.07025	3rd Qu.:0.0755
Max. :0.0870	Max. :0.0820	Max. :0.10100	Max. :0.0880

Losowe5	Losowe6	Losowe7	Losowe8
Min. :0.04000	Min. :0.03700	Min. :0.06200	Min. :0.04500
1st Qu.:0.05050	1st Qu.:0.06375	1st Qu.:0.06700	1st Qu.:0.04950
Median :0.05550	Median :0.07700	Median :0.07100	Median :0.05050
Mean :0.05555	Mean :0.07440	Mean :0.07235	Mean :0.05195
3rd Qu.:0.06100	3rd Qu.:0.08600	3rd Qu.:0.07525	3rd Qu.:0.05500
Max. :0.07100	Max. :0.09400	Max. :0.09000	Max. :0.06400

Losowe9	Losowe10	Losowe11	Losowe12
Min. :0.0360	Min. :0.03800	Min. :0.03700	Min. :0.04400
1st Qu.:0.0415	1st Qu.:0.04275	1st Qu.:0.03975	1st Qu.:0.05275
Median :0.0445	Median :0.04400	Median :0.04250	Median :0.06300
Mean :0.0474	Mean :0.04525	Mean :0.04645	Mean :0.06050
3rd Qu.:0.0525	3rd Qu.:0.04650	3rd Qu.:0.05525	3rd Qu.:0.06450
Max. :0.0670	Max. :0.06000	Max. :0.05800	Max. :0.07700

Losowe13	Losowe14	Losowe15	Losowe16
Min. :0.04000	Min. :0.04400	Min. :0.03800	Min. :0.0380
1st Qu.:0.04675	1st Qu.:0.05375	1st Qu.:0.04100	1st Qu.:0.0440
Median :0.05150	Median :0.06000	Median :0.04450	Median :0.0470
Mean :0.05145	Mean :0.06010	Mean :0.04535	Mean :0.0475
3rd Qu.:0.05475	3rd Qu.:0.06600	3rd Qu.:0.04800	3rd Qu.:0.0500
Max. :0.07200	Max. :0.07800	Max. :0.05400	Max. :0.0580

Losowe17	Losowe18	Losowe19	Losowe20
Min. :0.04000	Min. :0.04700	Min. : 0.03900	Min. :0.04600
1st Qu.:0.04200	1st Qu.:0.05100	1st Qu.: 0.04550	1st Qu.:0.05000
Median :0.04550	Median :0.05400	Median : 0.05000	Median :0.05850
Mean :0.04750	Mean :0.05505	Mean : 2.09765	Mean :0.05705
3rd Qu.:0.05325	3rd Qu.:0.05850	3rd Qu.: 0.05425	3rd Qu.:0.06100
Max. :0.05800	Max. :0.06800	Max. :41.00000	Max. :0.07000

Tabela A3: **3Glucose_syrup** (charakterystyki czasu złamania klucza dla poszczególnych prób losowych, min, 1stQu-pierwszy kwartyl, 3rdQu-trzeci kwartyl)

Losowe1	Losowe2	Losowe3	Losowe4
Min. :0.07100	Min. :0.03500	Min. :0.0500	Min. :0.1330
1st Qu.:0.09175	1st Qu.:0.04000	1st Qu.:0.1037	1st Qu.:0.1520
Median :0.10100	Median :0.04150	Median :0.1165	Median :0.1805
Mean :0.10060	Mean :0.04340	Mean :0.1195	Mean :0.1825
3rd Qu.:0.11300	3rd Qu.:0.04625	3rd Qu.:0.1445	3rd Qu.:0.2008
Max. :0.13100	Max. :0.05600	Max. :0.1800	Max. :0.2800

Losowe5	Losowe6	Losowe7	Losowe8
Min. :0.0870	Min. :0.0810	Min. :0.1250	Min. :0.0970
1st Qu.:0.1015	1st Qu.:0.1060	1st Qu.:0.1665	1st Qu.:0.1118
Median :0.1185	Median :0.1315	Median :0.1815	Median :0.1170
Mean :0.1171	Mean :0.1434	Mean :0.1842	Mean :0.1202
3rd Qu.:0.1278	3rd Qu.:0.1630	3rd Qu.:0.2018	3rd Qu.:0.1315
Max. :0.1530	Max. :0.2860	Max. :0.2350	Max. :0.1460

Losowe9	Losowe10	Losowe11	Losowe12
Min. :0.1030	Min. :0.0920	Min. :0.07900	Min. :0.0890
1st Qu.:0.1067	1st Qu.:0.1060	1st Qu.:0.09675	1st Qu.:0.1197
Median :0.1185	Median :0.1305	Median :0.11050	Median :0.1365
Mean :0.1223	Mean :0.1283	Mean :0.11110	Mean :0.1400
3rd Qu.:0.1315	3rd Qu.:0.1452	3rd Qu.:0.11825	3rd Qu.:0.1570
Max. :0.1700	Max. :0.1670	Max. :0.15400	Max. :0.2120

Losowe13	Losowe14	Losowe15	Losowe16
Min. :0.08800	Min. :0.0930	Min. :0.1020	Min. :0.1260
1st Qu.:0.09875	1st Qu.:0.1035	1st Qu.:0.1108	1st Qu.:0.1435
Median :0.10900	Median :0.1100	Median :0.1170	Median :0.1475
Mean :0.11015	Mean :0.1116	Mean :0.3617	Mean :0.1522
3rd Qu.:0.12025	3rd Qu.:0.1187	3rd Qu.:0.1212	3rd Qu.:0.1550
Max. :0.13800	Max. :0.1420	Max. :5.0000	Max. :0.1890

Losowe17	Losowe18	Losowe19	Losowe20
Min. :0.1250	Min. :0.1410	Min. :0.1110	Min. :0.1070
1st Qu.:0.1370	1st Qu.:0.1573	1st Qu.:0.1225	1st Qu.:0.1168
Median :0.1520	Median :0.1880	Median :0.1375	Median :0.1255
Mean :0.1539	Mean :0.1897	Mean :0.1433	Mean :0.1280
3rd Qu.:0.1610	3rd Qu.:0.2275	3rd Qu.:0.1570	3rd Qu.:0.1353
Max. :0.2320	Max. :0.2420	Max. :0.2080	Max. :0.1580

Tabela A4: **3Plingeling** (charakterystyki czasu złamania klucza dla poszczególnych prób losowych, min, 1stQu-pierwszy kwartyl, 3rdQu-trzeci kwartyl)

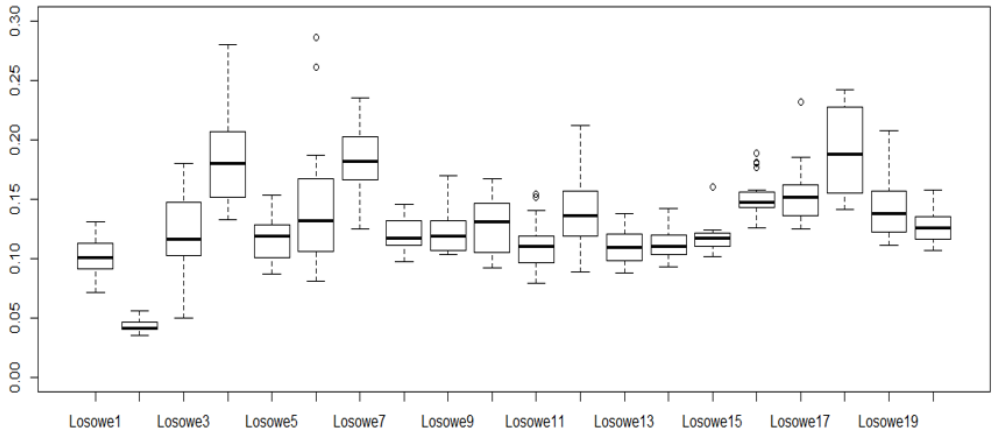
3MiniSat				
Losowe1	Losowe2	Losowe3	Losowe4	Losowe5
0.17	0.06	0.15	0.15	0.16
Losowe6	Losowe7	Losowe8	Losowe9	Losowe10
0.09	0.16	0.13	0.07	0.10
Losowe11	Losowe12	Losowe13	Losowe14	Losowe15
0.11	0.39	0.14	0.12	0.14
Losowe16	Losowe17	Losowe18	Losowe19	Losowe20
0.08	0.08	0.09	1.39	0.11

3Glu_vc				
Losowe1	Losowe2	Losowe3	Losowe4	Losowe5
0.08	0.12	0.14	0.10	0.09
Losowe6	Losowe7	Losowe8	Losowe9	Losowe10
0.07	0.12	0.14	0.11	0.06
Losowe11	Losowe12	Losowe13	Losowe14	Losowe15
0.14	0.09	0.13	0.12	0.17
Losowe16	Losowe17	Losowe18	Losowe19	Losowe20
0.18	0.14	0.12	0.06	0.16

3Glucose_syrup				
Losowe1	Losowe2	Losowe3	Losowe4	Losowe5
0.14	0.12	0.21	0.11	0.14
Losowe6	Losowe7	Losowe8	Losowe9	Losowe10
0.19	0.11	0.09	0.18	0.12
Losowe11	Losowe12	Losowe13	Losowe14	Losowe15
0.17	0.15	0.16	0.15	0.11
Losowe16	Losowe17	Losowe18	Losowe19	Losowe20
0.11	0.14	0.10	4.37	0.13

3Plingeling				
Losowe1	Losowe2	Losowe3	Losowe4	Losowe5
0.14	0.12	0.26	0.22	0.16
Losowe6	Losowe7	Losowe8	Losowe9	Losowe10
0.38	0.15	0.12	0.16	0.18
Losowe11	Losowe12	Losowe13	Losowe14	Losowe15
0.19	0.23	0.14	0.11	3.02
Losowe16	Losowe17	Losowe18	Losowe19	Losowe20
0.11	0.16	0.19	0.19	0.11

Tabela A5: Współczynniki zmienności (odchylenie/średnia)

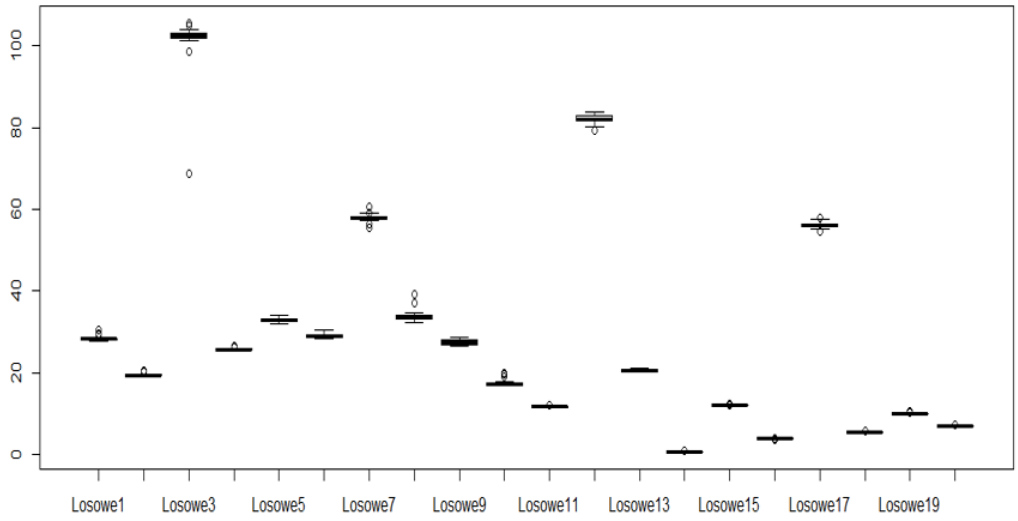


Rysunek A4: Boxploty dla 3Plingeling

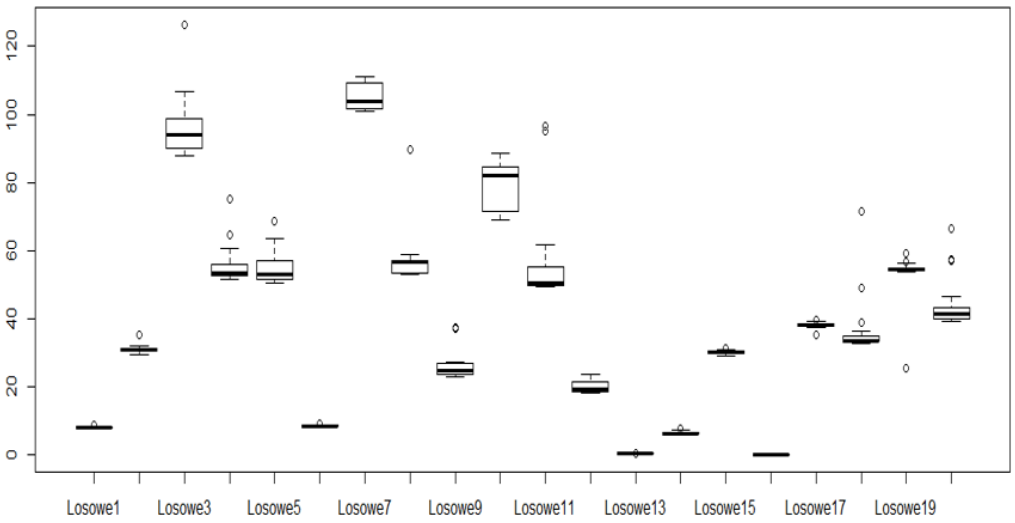
3MiniSat					
Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.0600	0.0900	0.1250	0.1945	0.1525	1.3900
3Glu_vc					
Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.060	0.090	0.120	0.117	0.140	0.180
3Glucose_syrup					
Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.0900	0.1100	0.1400	0.3500	0.1625	4.3700
3Plingeling					
Min	1st Qu.	Median	Mean	3rd Qu.	Max.
0.1100	0.1350	0.1600	0.3170	0.1975	3.0200

Tabela A6: Współczynniki zmienności (odchylenie/średnia)

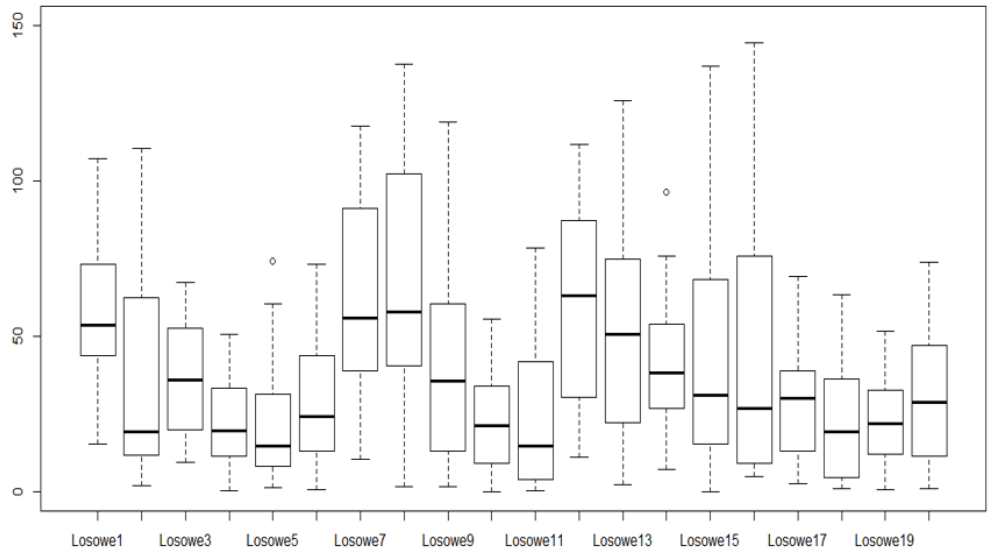
A2. 4 rundy DES (64-bitowy klucz)



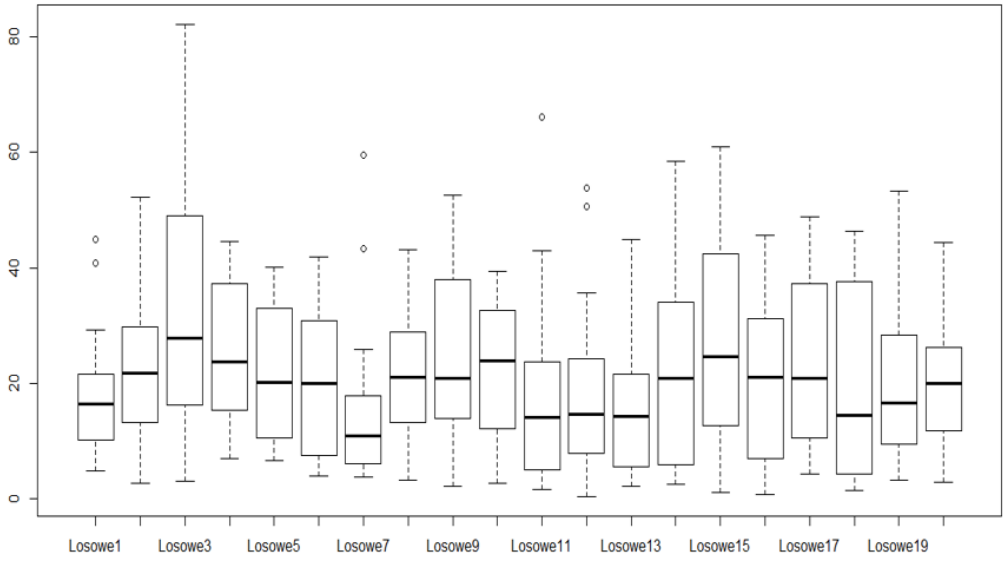
Rysunek A5: Boxploty dla 4MiniSat



Rysunek A6: Boxploty dla 4Glu_vc



Rysunek A7: Boxploty dla 4Glucose_syrup



Rysunek A8: Boxploty dla 4Plingeling

Losowe1	Losowe2	Losowe3	Losowe4
Min. :27.92	Min. :19.17	Min. : 68.86	Min. :25.54
1st Qu.:28.20	1st Qu.:19.38	1st Qu.:102.01	1st Qu.:25.71
Median :28.27	Median :19.47	Median :102.53	Median :25.80
Mean :28.47	Mean :19.53	Mean :100.95	Mean :25.85
3rd Qu.:28.43	3rd Qu.:19.53	3rd Qu.:103.09	3rd Qu.:25.95
Max. :30.42	Max. :20.50	Max. :105.51	Max. :26.60

Losowe5	Losowe6	Losowe7	Losowe8
Min. :32.16	Min. :28.31	Min. :55.65	Min. :32.36
1st Qu.:32.58	1st Qu.:28.60	1st Qu.:57.59	1st Qu.:33.20
Median :32.87	Median :29.00	Median :58.02	Median :33.72
Mean :32.90	Mean :29.07	Mean :58.01	Mean :34.01
3rd Qu.:33.12	3rd Qu.:29.23	3rd Qu.:58.21	3rd Qu.:34.22
Max. :33.98	Max. :30.38	Max. :60.55	Max. :39.22

Losowe9	Losowe10	Losowe11	Losowe12
Min. :26.68	Min. :16.83	Min. :11.55	Min. :79.37
1st Qu.:26.93	1st Qu.:16.96	1st Qu.:11.65	1st Qu.:81.66
Median :27.38	Median :17.19	Median :11.74	Median :82.05
Mean :27.47	Mean :17.54	Mean :11.78	Mean :82.12
3rd Qu.:27.94	3rd Qu.:17.56	3rd Qu.:11.80	3rd Qu.:82.76
Max. :28.83	Max. :19.91	Max. :12.19	Max. :83.97

Losowe13	Losowe14	Losowe15	Losowe16
Min. :20.34	Min. :0.7660	Min. :11.90	Min. :3.811
1st Qu.:20.38	1st Qu.:0.7748	1st Qu.:11.99	1st Qu.:3.863
Median :20.45	Median :0.7855	Median :12.04	Median :3.869
Mean :20.56	Mean :0.7898	Mean :12.07	Mean :3.898
3rd Qu.:20.65	3rd Qu.:0.7987	3rd Qu.:12.08	3rd Qu.:3.889
Max. :21.13	Max. :0.8870	Max. :12.46	Max. :4.105

Losowe17	Losowe18	Losowe19	Losowe20
Min. :54.56	Min. :5.407	Min. : 9.945	Min. :6.826
1st Qu.:55.92	1st Qu.:5.480	1st Qu.:10.038	1st Qu.:6.857
Median :56.27	Median :5.494	Median :10.063	Median :6.872
Mean :56.25	Mean :5.519	Mean :10.112	Mean :6.929
3rd Qu.:56.54	3rd Qu.:5.534	3rd Qu.:10.139	3rd Qu.:6.935
Max. :57.93	Max. :5.689	Max. :10.612	Max. :7.444

Tabela A7: **4MiniSat** (charakterystyki czasu złamania klucza dla poszczególnych prób losowych, min, 1stQu-pierwszy kwartyl, 3rdQu-trzeci kwartyl)

Losowe1	Losowe2	Losowe3	Losowe4
Min. :7.826	Min. :29.53	Min. : 87.76	Min. :51.44
1st Qu.:7.947	1st Qu.:30.34	1st Qu.: 89.98	1st Qu.:52.55
Median :7.994	Median :30.91	Median : 94.13	Median :53.23
Mean :8.038	Mean :31.04	Mean : 96.28	Mean :55.40
3rd Qu.:8.071	3rd Qu.:31.31	3rd Qu.: 98.50	3rd Qu.:55.85
Max. :8.836	Max. :35.15	Max. :126.31	Max. :75.16

Losowe5	Losowe6	Losowe7	Losowe8
Min. :50.58	Min. :8.033	Min. :101.0	Min. :52.99
1st Qu.:51.59	1st Qu.:8.311	1st Qu.:101.7	1st Qu.:53.55
Median :53.07	Median :8.421	Median :103.8	Median :56.76
Mean :55.21	Mean :8.435	Mean :105.2	Mean :57.25
3rd Qu.:56.77	3rd Qu.:8.544	3rd Qu.:109.3	3rd Qu.:57.18
Max. :68.46	Max. :8.965	Max. :111.0	Max. :89.59

Losowe9	Losowe10	Losowe11	Losowe12
Min. :22.69	Min. :69.08	Min. :49.37	Min. :18.28
1st Qu.:23.71	1st Qu.:71.58	1st Qu.:49.85	1st Qu.:18.59
Median :24.55	Median :82.14	Median :50.44	Median :19.13
Mean :26.02	Mean :79.64	Mean :56.21	Mean :19.78
3rd Qu.:26.74	3rd Qu.:84.39	3rd Qu.:53.38	3rd Qu.:21.25
Max. :37.23	Max. :88.60	Max. :96.55	Max. :23.66

Losowe13	Losowe14	Losowe15	Losowe16
Min. :0.2950	Min. :6.126	Min. :29.08	Min. :0.1130
1st Qu.:0.3058	1st Qu.:6.171	1st Qu.:29.89	1st Qu.:0.1200
Median :0.3075	Median :6.242	Median :30.20	Median :0.1300
Mean :0.3106	Mean :6.508	Mean :30.06	Mean :0.1271
3rd Qu.:0.3137	3rd Qu.:6.536	3rd Qu.:30.32	3rd Qu.:0.1330
Max. :0.3330	Max. :7.776	Max. :31.32	Max. :0.1430

Losowe17	Losowe18	Losowe19	Losowe20
Min. :35.17	Min. :32.56	Min. :25.55	Min. :39.14
1st Qu.:37.95	1st Qu.:33.30	1st Qu.:53.96	1st Qu.:39.80
Median :38.19	Median :33.39	Median :54.57	Median :41.43
Mean :38.13	Mean :36.63	Mean :53.41	Mean :44.08
3rd Qu.:38.38	3rd Qu.:34.45	3rd Qu.:54.83	3rd Qu.:43.04
Max. :39.48	Max. :71.69	Max. :59.15	Max. :66.26

Tabela A8: **4Glu_vc** (charakterystyki czasu złamania klucza dla poszczególnych prób losowych, min, 1stQu-pierwszy kwartyl, 3rdQu-trzeci kwartyl)

Losowe1	Losowe2	Losowe3	Losowe4
Min. : 15.51	Min. : 2.156	Min. : 9.534	Min. : 0.545
1st Qu.: 45.52	1st Qu.: 12.591	1st Qu.:21.009	1st Qu.:11.664
Median : 53.63	Median : 19.177	Median :36.050	Median :19.523
Mean : 57.82	Mean : 43.524	Mean :37.006	Mean :23.294
3rd Qu.: 72.62	3rd Qu.: 60.150	3rd Qu.:49.688	3rd Qu.:32.932
Max. :107.16	Max. :168.359	Max. :67.324	Max. :50.792

Losowe5	Losowe6	Losowe7	Losowe8
Min. : 1.41	Min. : 0.589	Min. : 10.35	Min. : 1.67
1st Qu.: 8.42	1st Qu.:13.678	1st Qu.: 39.26	1st Qu.: 41.25
Median :14.65	Median :24.346	Median : 55.73	Median : 57.94
Mean :23.34	Mean :29.172	Mean : 63.90	Mean : 65.89
3rd Qu.:31.34	3rd Qu.:43.158	3rd Qu.: 90.25	3rd Qu.:100.97
Max. :74.30	Max. :73.066	Max. :117.44	Max. :137.39

Losowe9	Losowe10	Losowe11	Losowe12
Min. : 1.854	Min. : 0.209	Min. : 0.389	Min. : 11.19
1st Qu.: 14.221	1st Qu.: 9.350	1st Qu.: 4.142	1st Qu.: 30.74
Median : 35.547	Median :21.267	Median :14.704	Median : 62.93
Mean : 43.855	Mean :22.968	Mean :24.120	Mean : 70.19
3rd Qu.: 58.260	3rd Qu.:33.977	3rd Qu.:41.078	3rd Qu.: 82.88
Max. :118.865	Max. :55.500	Max. :78.311	Max. :292.76

Losowe13	Losowe14	Losowe15	Losowe16
Min. : 2.334	Min. : 7.264	Min. : 0.081	Min. : 4.864
1st Qu.: 22.910	1st Qu.:28.827	1st Qu.: 18.110	1st Qu.: 10.492
Median : 50.549	Median :38.274	Median : 31.108	Median : 26.851
Mean : 53.680	Mean :40.932	Mean : 43.773	Mean : 46.410
3rd Qu.: 74.218	3rd Qu.:53.270	3rd Qu.: 67.852	3rd Qu.: 66.870
Max. :125.734	Max. :96.467	Max. :136.876	Max. :144.437

Losowe17	Losowe18	Losowe19	Losowe20
Min. : 2.60	Min. : 0.935	Min. : 0.755	Min. : 1.145
1st Qu.:13.39	1st Qu.: 4.624	1st Qu.:12.334	1st Qu.:13.492
Median :30.09	Median :19.175	Median :21.978	Median :28.816
Mean :30.15	Mean :22.142	Mean :22.967	Mean :31.168
3rd Qu.:38.68	3rd Qu.:34.786	3rd Qu.:32.645	3rd Qu.:46.011
Max. :69.30	Max. :63.350	Max. :51.706	Max. :73.963

Tabela A9: **4Glucose_syrup** (charakterystyki czasu złamania klucza dla poszczególnych prób losowych, min, 1stQu-pierwszy kwartyl, 3rdQu-trzeci kwartyl)

Losowe1	Losowe2	Losowe3	Losowe4
Min. : 4.794	Min. : 2.659	Min. : 3.018	Min. : 7.049
1st Qu.:10.663	1st Qu.:13.640	1st Qu.:19.081	1st Qu.:15.478
Median :16.455	Median :21.767	Median :27.816	Median :23.784
Mean :18.316	Mean :22.773	Mean :34.212	Mean :25.411
3rd Qu.:21.225	3rd Qu.:29.645	3rd Qu.:48.787	3rd Qu.:35.725
Max. :44.910	Max. :52.252	Max. :82.192	Max. :44.478

Losowe5	Losowe6	Losowe7	Losowe8
Min. : 6.709	Min. : 3.922	Min. : 3.858	Min. : 3.269
1st Qu.:10.703	1st Qu.: 7.537	1st Qu.: 6.198	1st Qu.:13.700
Median :20.215	Median :20.066	Median :10.870	Median :21.027
Mean :21.741	Mean :20.582	Mean :15.351	Mean :21.775
3rd Qu.:32.693	3rd Qu.:30.672	3rd Qu.:17.762	3rd Qu.:28.803
Max. :40.188	Max. :41.821	Max. :59.477	Max. :43.162

Losowe9	Losowe10	Losowe11	Losowe12
Min. : 2.137	Min. : 2.743	Min. : 1.701	Min. : 0.371
1st Qu.:15.020	1st Qu.:12.662	1st Qu.: 5.169	1st Qu.: 8.953
Median :20.945	Median :23.986	Median :14.181	Median :14.659
Mean :25.252	Mean :22.313	Mean :17.560	Mean :18.304
3rd Qu.:37.799	3rd Qu.:31.497	3rd Qu.:22.888	3rd Qu.:23.435
Max. :52.554	Max. :39.455	Max. :66.145	Max. :53.872

Losowe13	Losowe14	Losowe15	Losowe16
Min. : 2.147	Min. : 2.563	Min. : 1.162	Min. : 0.784
1st Qu.: 5.728	1st Qu.: 6.358	1st Qu.:12.816	1st Qu.: 7.300
Median :14.361	Median :20.840	Median :24.648	Median :21.134
Mean :16.845	Mean :21.753	Mean :28.101	Mean :20.900
3rd Qu.:21.203	3rd Qu.:32.804	3rd Qu.:41.481	3rd Qu.:30.787
Max. :44.886	Max. :58.458	Max. :60.891	Max. :45.559

Losowe17	Losowe18	Losowe19	Losowe20
Min. : 4.401	Min. : 1.459	Min. : 3.228	Min. : 2.848
1st Qu.:11.090	1st Qu.: 4.295	1st Qu.: 9.602	1st Qu.:12.183
Median :20.956	Median :14.556	Median :16.670	Median :19.984
Mean :23.761	Mean :20.435	Mean :21.821	Mean :19.916
3rd Qu.:36.837	3rd Qu.:37.601	3rd Qu.:28.256	3rd Qu.:25.909
Max. :48.868	Max. :46.252	Max. :53.224	Max. :44.351

Tabela A10: **4Plingeling** (charakterystyki czasu złamania klucza dla poszczególnych prób losowych, min, 1stQu-pierwszy kwartyl, 3rdQu-trzeci kwartyl)

4MiniSat				
Losowe1	Losowe2	Losowe3	Losowe4	Losowe5
0.02	0.02	0.08	0.01	0.02
Losowe6	Losowe7	Losowe8	Losowe9	Losowe10
0.02	0.02	0.05	0.02	0.05
Losowe11	Losowe12	Losowe13	Losowe14	Losowe15
0.02	0.01	0.01	0.03	0.01
Losowe16	Losowe17	Losowe18	Losowe19	Losowe20
0.02	0.01	0.01	0.02	0.02
4Glu_vc				
Losowe1	Losowe2	Losowe3	Losowe4	Losowe5
0.03	0.04	0.09	0.10	0.09
Losowe6	Losowe7	Losowe8	Losowe9	Losowe10
0.02	0.04	0.14	0.16	0.09
Losowe11	Losowe12	Losowe13	Losowe14	Losowe15
0.25	0.08	0.03	0.08	0.02
Losowe16	Losowe17	Losowe18	Losowe19	Losowe20
0.06	0.02	0.25	0.12	0.17
4Glucose_syrup				
Losowe1	Losowe2	Losowe3	Losowe4	Losowe5
0.42	1.04	0.50	0.64	0.87
Losowe6	Losowe7	Losowe8	Losowe9	Losowe10
0.75	0.50	0.58 0.83	0.73	
Losowe11	Losowe12	Losowe13	Losowe14	Losowe15
0.97 0.86	0.68	0.55	0.87	
Losowe16	Losowe17	Losowe18	Losowe19	Losowe20
1.00	0.63	0.83	0.63	0.71
4Plingeling				
Losowe1	Losowe2	Losowe3	Losowe4	Losowe5
0.59	0.59	0.69	0.49	0.53
Losowe6	Losowe7	Losowe8	Losowe9	Losowe10
0.66	0.91	0.51	0.56	0.54
Losowe11	Losowe12	Losowe13	Losowe14	Losowe15
0.91	0.82	0.73	0.78	0.67
Losowe16	Losowe17	Losowe18	Losowe19	Losowe20
0.70	0.62	0.85	0.70	0.57

Tabela A11: Współczynniki zmienności (odchylenie/średnia)

4MiniSat					
Min.	1st Qu.	Median.	Mean	3rd Qu.	Max.
0.0100	0.0100	0.0200	0.0235	0.0200	0.0800
4Glu_vc					
Min.	1st Qu.	Median.	Mean	3rd Qu.	Max.
0.0200	0.0375	0.0850	0.0940	0.1250	0.2500
4Glucose_syrup					
Min.	1st Qu.	Median.	Mean	3rd Qu.	Max.
0.4200	0.6175	0.7200	0.7295	0.8625	1.0400
4Plingeling					
Min.	1st Qu.	Median.	Mean	3rd Qu.	Max.
0.4900	0.5675	0.6650	0.6710	0.7425	0.9100

Tabela A12: Współczynniki zmienności (odchylenie/średnia)

.

Dodatek B

B1. Przykład działania podstawienia S-box w AES

W tym dodatku przedstawiono sposób działania podstawienia `S-box` używanego w algorytmie AES.

Niech s_{21} = `0x46`. Wówczas wartość podstawienia `S-box` dla s_{21} znajduje się w komórce leżącej na przecięciu wiersza o indeksie 4 i kolumny o indeksie 6. Stąd s'_{21} ma wartość `0x5a`. Działanie S-boxa na powyższym przykładzie ilustruje tabela B1.

HEX		y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
x	**0**	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	**5a**	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Tabela B1: Przykład działania S-boxa.