Wydział Matematyki, Fizyki i Informatyki
Uniwersytet Marii Curie-Skłodowskiej w Lublinie

Stanisław Jakub Kotorowicz

Kryptograficzne algorytmy strumieniowe oparte na specjalnych grafach algebraicznych

Rozprawa doktorska napisana pod kierunkiem
prof. dr hab. Vasyla Ustimenko

LUBLIN 2014

Składam serdeczne podziękowania Profesorowi Vasylowi Ustimenko za opiekę, pomoc i czas poświęcony w toku realizacji przewodu doktorskiego.

Kryptograficzne algorytmy strumieniowe oparte na specjalnych grafach algebraicznych

Streszczenie

W niniejszej pracy zaprezentujemy dwie rodziny grafów algebraicznych bez krótkich cykli, posiadające ekstremalne własności. Nieskończone rodziny $D(n, \mathbb{K})$ oraz $A(n, \mathbb{K})$ są rodzinami grafów dwudzielnych, których strukturę incydencji określają równania algebraiczne nad pierścieniem przemiennym $\mathbb{K}$ lub ciałem skończonym $\mathbb{K} = \mathbb{F}_q$. Zaprezentujemy teoretyczne własności tych rodzin grafów oraz oszacowania, będące wynikami eksperymentów komputerowych, tam gdzie matematycznie własności jeszcze nie dowiedziono.

Pokażemy również w jaki sposób zbudować kryptograficzny system symetryczny, używając wprowadzonych rodzin grafów. Z każdym wierzchołkiem grafu kojarzymy krotkę $\mathbb{K}^n$. Podstawowa procedura szyfrowania polega na spacerze po wierzchołkach grafu ścieżką wyznaczoną przez klucz, zaczynając od wierzchołka, z którym skojarzona jest wiadomość otwarta (n wybieramy na podstawie długości tej wiadomości). Ostatni wierzchołek tej ścieżki ustalamy na szyfrogram.

W dalszej części pracy zaprezentujemy wyniki eksperymentów, polegających na zbadaniu zmian w szyfrogramie, przy pojedynczych zmianach klucza lub tekstu otwartego. Opiszemy problem, który odkryliśmy tymi testami oraz przedstawimy rozwiązanie problemu inspirowane ideą kryptografii wielu zmiennych (składanie przekształcenia bazującego na grafach z dwoma przekształceniami afinicznymi). Uzasadnimy również wybór konkretnych przekształceń afinicznych, szybkich i dopasowanych do naszych rodzin grafów oraz natury znalezionego problemu.

Wymagania stawiane dzisiejszym szyfrom symetrycznym to bezpieczeństwo i szybkość. W związku z tym pokażemy teoretyczne oszacowania szybkości naszego systemu oraz wyniki eksperymentów, polegających na porównaniu naszego szyfru, zbudowanego nad pierścieniami $\mathbb{Z}_{2^8}$, $\mathbb{Z}_{2^{16}}$, $\mathbb{Z}_{2^{32}}$, $\mathbb{Z}_{2^{64}}$ ze znanymi ze swojej szybkości (choć niedostatecznie bezpiecznymi) szyframi RC4 oraz DES oraz z dwoma implementacjami historycznymi. Skupienie się w naszej implementacji na rodzinach grafów budowanych nad pierścieniami $\mathbb{Z}_{2^i}$ uzasadniamy wymaganiami szybkości. Dzięki temu możemy bazować na działaniach $+, -, *, mod$ na typach całkowitych (naturalnych dla maszyn), co nie byłoby możliwe w przypadku mnożenia w ciele skończonym $\mathbb{F}_q$, gdzie q jest potęgą liczby pierwszej.

Zaprezentujemy dwie implementacje naszego kryptosystemu, gdzie w jednej skupiliśmy się na wydajności (każdy algorytm jest w całości pojedynczą

funkcją w języku programowia), natomiast druga zbudowana jest w zgodzie z nowymi trendami inżynierii oprogramowania i wykorzystuje znane wzorce projektowe. Dzięki tym wzorcom możliwe są modyfikacje algorytmów poprzez wprowadzanie nowych algorytmów tej samej klasy oraz łączenie kilku algorytmów w jeden. Poprawność rozwiązań uzasadniamy stosowaniem testów jednostkowych.

Druga z naszych implementacji wprowadza po raz pierwszy algorytmy zbudowane na naszych grafach nad pierścieniami boolowskimi, które łatwo zaimplementować w rozwiązaniach sprzętowych. Porównujemy prędkość działania tych algorytmów nad pierścieniami boolowskimi z algorytmami zbudowanymi nad pierścieniami arytmetycznymi.

Przeprowadzimy również kryptoanalizę naszego systemu dwoma różnymi metodami, wskazując jaki atak mógłby złamać ten system. W naszej opinii atak typu linearyzacji, będący zdecydowanie szybszy niż brutalne przeszukiwanie grafu, nie stanowi praktycznego zagrożenia dla szyfru symetrycznego ze względu na duże ilości liniowo niezależnych par (tekst otwarty, szyfrogram), które są potrzebne do jego przeprowadzenia (rzędu $O(n^3)$) oraz względnie duży czas ataku (rzędu $O(n^{10})$).

Wskażemy w jaki sposób można podnieść stopień przekształceń szyfrujących i deszyfrujących, aby uniemożliwić ataki typu linearyzacji. Jednak podniesienie stopnia przekształceń odbija się niekorzystnie na prędkości działania algorytmów, prowadzi do utraty niektórych korzystnych własności algorytmów.

Na koniec wspomnimy o innych możliwościach wykorzystania naszych rodzin grafów: systemie kryptografii publicznej oraz algorytmie wymiany kluczy, a także problemach wiążących się z tymi zagadnieniami.

Cryptographical stream ciphers based on the special algebraic graphs

Summary

In this paper we present two families of algebraic graphs without short cycles having extreme properties. Infinite families $D(n, \mathbb{K})$, and $A(n, \mathbb{K})$ are the families of bipartite graphs, which have the incidence structure defined via the algebraic equations over a commutative ring $\mathbb{K}$ or a finite field $\mathbb{K} = \mathbb{F}_q$. We present the theoretical properties of those families and some results of the computer experiments connected to properties, which were not proven mathematically.

We also show how to construct the symmetric cryptosystem, based on these two families of graphs. With each vertex of the graph we associate a tuple $\mathbb{K}^n$. A basic encryption procedure consists of a walk over the vertices of the graph, which uses the path defined by the key (key is also a tuple over $\mathbb{K}$) starting from the vertex associated with the plaintext. The tuple associated with the last vertex of this path is beeing treated as the cipthertext.

In the later parts of this manuscript we present the results of experiments consisting in the observation of the changes in the ciphertext when we change an element of the plaintext or an element of the key. We describe the problem, which has been found during these experiments and present a solution of the problem, through the multivariate cryptography (combining our graph based transformation with two affine transformations). We justify the choice of the concrete affine transformations, with the good speed of their generation ($O(n)$), and adjusted to the problem for our graph families.

Nowadays stream ciphers should be fast and secure. According to these requirements we present the theoretical estimation of the speed of our cryptosystem. We also show the results of an experiment of the comparison of our cipher constructed over commutative rings $\mathbb{Z}_{2^8}$, $\mathbb{Z}_{2^{16}}$, $\mathbb{Z}_{2^{32}}$, $\mathbb{Z}_{2^{64}}$ with the fast, but not very secure, well known algorithms RC4 and DES. We prefer to work with modulo rings instead of finite fields as the base of our operations, because of the performance parameters. Due to our choice, we work with integer numbers (with natural computer representation) and typical arithmetical operations: $+, -, *, mod$. If we decide to use finite fields the performance is much lower, because of the problems with multiplication in $\mathbb{F}_q$, where q is the power of a prime number.

We present two implementations of our cryptosystem. One of them is concentrated on achieving the maximum speed (each algorithms is fully closed in one computer function), where the second one is constructed according to new

trends in the software engineering, using well known design patterns and unit tests. The application of the design patterns allows us to modify the existing algorithms via building of the new algorithms in the same class or combining the existing algorithms into the new one.

Our second implementation introduces for the first time the families of graphs based algorithms over the boolean rings, which are convenient for the hardware implementation. We also compare the speed of our algorithms for families of graphs over arithmetical rings with the case of graphs over boolean rings.

We present some results of the cryptoanalysis of our system using two different methods. An attack using algebraic linearization is much faster than a brutal search over the passes in the graph, and much faster than the Dijkstra algorithm searching for the shortest path between plaintext and ciphertext. Nevertheless in our opinion this attack is not a practical treat to our system due to a large number of resources needed to perform attack: $O(n^3)$ linear independent pairs of type "plaintext-ciphertext" is necessary. The time complexity of this attack can be evaluated as $O(n^{10})$.

Additionally, we consider modifications of our algorithm, which lead to increase of the degree of the polynomial encryption and decryption maps. It makes linearization type attacks impossible, but leads to certain increase of the execution time, and the loss of some useful properties of the algorithm.

Last but not least, we mention the other possibilities of using our graph families in cryptography: public key cryptography and the key exchange protocol. We also mark the problems which appear it these directions.

Spis treści

Wstęp

W dzisiejszych czasach obserwujemy rozwój różnych nowych dziedzin kryptografii, powstają nowe funkcje skrótu, algorytmy wymiany kluczy, a najbardziej znany algorytm klucza publicznego RSA doczekał się setek modyfikacji oraz publikacji dotyczących rozkładu ogromnych liczb na czynniki pierwsze. Kryptografia symetryczna jest najstarszą dziedziną kryptografii (zapoczątkowały ją antyczne szyfry przestawieniowe). Jednak cały czas istnieje potrzeba jej stosowania i tworzenia nowych szybkich algorytmów symetrycznych.

Wielu z nas nie zdaje sobie sprawy jak często używa w codziennym życiu elementów kryptografii symetrycznej. Każde bezpieczne połączenie internetowe, każda praca z zabezpieczoną siecią Wi-Fi, każdy transfer szyfrowanych danych między dwoma elektronicznymi urządzeniami wymaga ustalenia klucza sesji (najczęściej mechanizmem klucza publicznego), po czym transfer szyfrowanych danych przebiega przy zastosowaniu szyfrów symetrycznych. Nie lubimy czekać i nie lubimy tracić poufności naszych danych. Istnieje więc cały czas zapotrzebowanie na nowe szybkie i bezpieczne algorytmy klucza prywatnego.

Badania nad rodzinami grafów ekstremalnych o dużej talii prowadzone przez V. Ustimenko, F. Lazebnika i A. J. Woldara [29, 30] poskutkowały powstaniem rodziny grafów zadanych przez równania algebraiczne. W jaki sposób można tę rodzinę wykorzystać w systemie symetrycznym po raz pierwszy pokazał Vasyl Ustimenko [45]. Pierwsze dwa kryptosystemy oparte o te grafy posiadały implementację z użyciem ciał skończonych jako podstawy budowy grafów [44, 46]. W kolejnych latach rozszerzono badania nad wspomnianą rodziną grafów i dokonano konstrukcji tej rodziny przy użyciu bardziej ogólnych pierścieni przemiennych. W naszej pracy przedstawiamy pierwszy kryptosystem, który wykorzystuje właśnie pierścienie przemienne (nie będące ciałem) jako podstawę obliczeń oraz konstrukcji grafów algebraicznych. Przedstawiamy też analizę bezpieczeństwa, analizę i testy własności kryptograficznych nie badanych do tej pory oraz możliwości rozszerzenia tego kryptosystemu w celu poprawy jego bezpieczeństwa.

Teza i cel pracy

Celem tej pracy jest:

- zbadanie algorytmów z klasy szyfrów strumieniowych, opartych na aproksymacji drzew, pod kątem szybkości i bezpieczeństwa,
- stworzenie nowych algorytmów w badanej klasie przez modyfikację algorytmów znanych do tej pory,
- wybór algorytmów z optymalnymi parametrami,
- porównanie właściwości kryptograficznych nowych algorytmów z poprzednimi (teoretyczne oraz symulacje).

Teza niniejszej rozprawy brzmi: *zaproponowane przez nas algorytmy klucza prywatnego oparte o grafy algebraiczne o dużej talii mają lepsze od znanych do tej pory algorytmów tej samej klasy właściwości kryptograficzne.*

Struktura pracy

Praca składa się z 7 rozdziałów. W części pierwszej (rozdziały 1–3) przypominamy wiedzę teoretyczną z kilku dziedzin nauk matematycznych, wprowadzamy oznaczenia oraz przytaczamy mniej znane fakty, które są wykorzystywane w drugiej części pracy.

Druga część pracy (rozdziały 4–7) przedstawia badany przez nas system kryptografii symetrycznej. Opisujemy w tej części konstrukcje rodzin grafów algebraicznych, własności wybranych rodzin grafów, a także sposób tworzenia algorytmów symetrycznych wykorzystujących dwie rodziny grafów. Przytaczamy tutaj wyniki matematycznie dowiedzione oraz wnioski z własnych eksperymentów komputerowych w miejscach, gdzie matematyka do tej pory milczy. Wskazujemy silne i słabe strony naszego systemu kryptograficznego oraz jak można poprawić niektóre słabe strony i z jakim dodatkowym narzutem się to wiąże. Na koniec porównujemy zaimplementowany przez nas system z dwoma systemami, które powstały wcześniej oraz przeprowadzamy wstępną kryptoanalizę zaproponowanego przez nas rozwiązania dwoma metodami. W każdym z rozdziałów 4–7 znajdują się oryginalne wyniki naszej pracy.

Rozdział 1 zawiera przypomnienie informacji z kilku dziedzin "nauk informatycznych. Najpierw opisujemy podstawe pojęcia kryptologiczne, podział nowoczesnej kryptografii oraz bardziej szczegółowy opis kryptografii symetrycznej. Prezentujemy tutaj również wymagania jakie stawia się systemom symetrycznym oraz możliwe ataki kryptologiczne.

W drugiej krótkiej części jest krótka sekcja, w której przytaczamy standardowe w informatyce teoretycznej notacje służące do szacowania złożoności algorytmów.

Na koniec rozdzialu opisujemy pewne aspekty praktyczne, związane z inżynierią oprogramowania. Opisujemy krótko ideę wzorców projektowych oraz te z nich, które zostały wykorzystane w budowanych przez nas systemów kryptograficznych.

Rozdział 2 jest krótkim przypomnieniem elementów algebry, które mają zastosowanie w kryptografii. Ostatnia sekcja przedstawia ogólny sposób konstruowania pewnej grupy algorytmów kryptograficznych opartych o obiekty algebraiczne — kryptografię wielu zmiennych.

Rozdział 3 zawiera wybrane elementy wiedzy z teorii grafów, opis idei problemów teorii grafów ekstremalnych, w szczególności problem konstrukcji rodziny grafów o rosnącym rzędzie, nie zawierających krótkich cykli. Na końcu rozdziału znajdziemy wyjaśnienie pojęcia grafów algebraicznych oraz opis konstrukcji grafów dwudzielnych ze strukturą incydencji zadaną równaniami algebraicznymi.

Rozdział 4 łączy wiedzę zawartą we wszystkich poprzednich rozdziałach. Znajdziemy tutaj schemat algorytmu symetrycznego, wykorzystującego spacer po wierzchołkach grafów algebraicznych. W dalszej części rozdziału prezentujemy dwie rodziny grafów algebraicznych, które można wybrać jako bazę szyfru symetrycznego. Przytaczamy własności tych rodzin grafów, z naciskiem na własności ważne z kryptograficznego punktu widzenia. Przedstawiamy też wyniki własnego eksperymentu, jako oszacowanie jednej z własności wprowadzonych rodzin grafów (rzędu grupy przekształcenia cyklicznego, którego do tej pory nie udało się oszacować teoretycznie).

Rozdział 5 zawiera krótki opis eksperymentów, polegających na badaniu zmian w wiadomościach zaszyfrowanych naszym algorytmem przy nieznacznych zmianach wiadomości wejściowej oraz hasła. Znajdziemy tu jeden z problemów jaki udało nam się wykryć w tych badaniach oraz opis rozszerzenia algorytmu sposobem analogicznym do konstrukcji algorytmów kryptografii wielu zmiennych. Przedstawiamy korzyści płynące z rozszerzenia algorytmu oraz wybrane przez nas parametry tego rozszerzenia z uzasadnieniem doboru tych parametrów.

Rozdział 6 poświęcony jest w całości praktycznym aspektom naszego algorytmu symetrycznego. Znajdziemy tutaj matematyczne oszacowania złożoności algorytmów, opis historycznych implementacji szyfrów opartych o grafy algebraiczne oraz opis naszego oryginalnego systemu. W rozdziale tym znajdują się również wyniki naszych eksperymentów i porównanie wydajności naszych

algorytmów z wcześniejszymi implementacjami oraz dwoma powszechnie znanymi szyframi symetrycznymi (RC4 i DES), uważanymi za szybkie.

Rozdział 7 zawiera dokonaną przez nas wstępną kryptoanalizę naszego szyfru symetrycznego. Pokazujemy w jaki sposób przy dużej ilości wygenerowanych par (wiadomość, szyfrogram) można zaatakować nasz system w czasie wielomianowym $O(n^{10})$, korzystając z wiedzy o stopniu przekształcenia oraz w jaki sposób można podnieść ten stopień przekształcenia.

Rozdział 1

Elementy matematycznych i inżynieryjnych nauk komputerowych

W tym rozdziale przytoczymy informację z kilku dziedzin naukowych związanych z szeroko rozumianą "informatyką", które później będziemy wykorzystywać w dalszej części pracy.

1.1 Kryptografia i kryptoanaliza

Na początek przypomnimy podstawowe pojęcia kryptologiczne oraz wprowadzimy oznaczenia używane w naszej rozprawie. Wykorzystamy w tym rozdziale wiedzę z pozycji [9, 22, 23, 24, 42].

1.1.1 Terminologia

Kryptologia jest dziedziną wiedzy o przekazywaniu informacji w sposób zabezpieczony przed niepowołanym dostępem. Kryptologię dzielimy na:

- *kryptografię* - gałąź wiedzy o utajnianiu wiadomości,
- *kryptoanalizę* - gałąź wiedzy o przełamywaniu zabezpieczeń, deszyfrowaniu zakodowanych wiadomości przy braku klucza (hasła) lub innego wymaganego elementu schematu szyfrowania.

Celem kryptografii jest stworzenie algorytmów, które pozwolą zaszyfrować wiadomość zwaną *tekstem jawnym* (ang. *plaintext*) w wiadomość nieczytelną zwaną *szyfrogramem* (ang. *ciphertext*) przy użyciu tajnego hasła, czyli *klucza*

(ang. *key*). Od wielu lat panuje przekonanie, że algorytm (czyli funkcja szyfrująca) powinnien być otwarty (znany), natomiast całe jego bezpieczeństwo powinno opierać się na tajnym kluczu.

Jeżeli oznaczymy wiadomość jawną jako M (ang. *message*), funkcję szyfrującą jako E (ang. *encryption function*), a szyfrogram jako C, to podstawową operację szyfrowania możemy przedstawić jako

$$E(M) = C.$$

Wiadomość zaszyfrowana C, przesyłana jest do odbiorcy tzw. kanałem otwartym. Następnie odbiorca po otrzymaniu wiadomości może ją zdeszyfrować, używając funkcji deszyfrującej (ang. *decryption funcion*), oznaczanej przeważnie jako D. W zapisie symbolicznym możemy to przedstawić jako

$$D(C) = M.$$

Oczywiście cała idea kryptografii polega na tym, aby odzyskać pierwotną wiadomość, więc musi zachodzić

$$D(E(M)) = M.$$

Sposób szyfrowania i deszyfrowania wiadomości zależy od klucza K, więc powyższe schematy postępowania w praktyce przyjmą postać

$$E_K(M) = C,$$
$$D_K(C) = M.$$

Atak kryptologiczny w najogólniejszym przypadku polega na przechwyceniu wiadomości C w kanale otwartym i próbie odzyskania wiadomości M bez znajomości klucza K.

1.1.2 Współczesny podział kryptografii

Schematy kryptograficzne na dzień dzisiejszy możemy podzielić na dwie podstawowe grupy:

- **Kryptografię symetryczną** (kryptografię klucza prywatnego) — gdzie zarówno do szyfrowania, jak i deszyfrowania używa się tego samego klucza. Podstawowym problemem jest w tym schemacie dystrybucja tajnego klucza. Często oczekuje się możliwości implementacji sprzętowej algorytmów symetrycznych. Jest to możliwe, gdy w algorytmie używa się operacji, które można zapisywać najprostszymi działaniami matematycznymi, którym odpowiadają w elektronice tzw. bramki logiczne.

- **Kryptografię asymetryczną** (kryptografię klucza publicznego) — gdzie wyróżniamy dwa klucze: prywatny oraz publiczny. Klucz prywatny posiada odbiorca wiadomości. Na podstawie tego klucza generuje on klucz publiczny, który następnie udostępnia wszystkim potencjalnym nadawcom. Każdy nadawca może zaszyfrować wiadomość przy użyciu klucza publicznego, natomiast tylko odbiorca może tak zaszyfrowaną wiadomość zdeszyfrować kluczem prywatnym i odzyskać tekst otwarty. Schematy szyfrowania i deszyfrowania są przeważnie różne, natomiast bezpieczeństwo kryptografii asymetrycznej opiera się na takim doborze kluczy, aby odzyskanie klucza prywatnego na podstawie publicznego oraz wielu zaszyfrowanych wiadomości było obliczalnie nieopłacalne. Oczekuje się, że złożoność obliczeniowa takiej kryptoanalizy powinna być na tyle wysoka, że przy dzisiejszym poziomie technologicznym, łamanie pojedynczej pary kluczy zajmować będzie całe lata (lub dłużej). Formalnie opiszemy to w sekcji 1.2 poświęconej złożoności obliczeniowej.

Większość algorytmów symetrycznych jest, przy tej samej długości hasła, o kilka rzędów wielkości szybsza niż standardowe algorytmy asymetryczne. Natomiast algorytmy asymetryczne charakteryzują się większym bezpieczeństwem i nie posiadają bariery związanej z dystrybucją tajnego klucza. W dawnych czasach nie było to problemem, ludzie spotykali się twarzą w twarz lub wysyłali posłańca z hasłem. Dziś jednak ogromne ilości danych wymieniane są jedynie za pomocą sieci komputerowych, a szybkość transmisji danych odgrywa ogromną rolę.

Kryptografowie wymyślili więc dla celów praktycznych tzw. *kryptografię mieszaną*. Bardzo często w przypadku transmisji danych między urządzeniami stosowany jest schemat dwuetapowy:

1. Przy pomocy algorytmu asymetrycznego dystrybuowany jest tzw. klucz sesji. Ilość danych jest niewielka, więc strata szybkości jest akceptowalna.

2. Po uzgodnieniu klucza sesji oraz wybraniu algorytmu (mogą być różne), transmisja faktycznych danych odbywa się przy pomocy szyfrowania symetrycznego. Bardzo ważna jest tu wydajność (szybkość działania), aby dane mogły być szyfrowane i deszyfrowane nawet na bieżąco w trakcie transmisji. Ilość danych może być bardzo duża (przesyłanie plików w bezpiecznych sieciach).

Po „rozłączeniu" urządzeń, czyli zakończeniu sesji, klucz sesji wykorzystany w algorytmie symetrycznym nie jest dłużej potrzebny i nie ma sensu jego łamanie. W następnej sesji zostanie uzgodniony nowy klucz sesji.

Jak widać, dzięki podejściu mieszanemu, wykorzystane są zalety obydwu rodzajów kryptografii dla otrzymania systemu szybkiego, a zarazem bezpiecznego. Ten prosty przykład powinien również uzmysławiać nam potrzebę konstruowania algorytmów zarówno symetrycznych, jak i asymetrycznych.

W niniejszej pracy skupimy się jedynie na kryptografii symetrycznej. Warto w tym miejscu wspomnieć, iż istnieje możliwość wykorzystania podejścia, które zaprezentujemy w systemie asymetrycznym i taką tematyką zajmuje się inny członek naszego zespołu mgr Michał Klisowski.

1.1.3 Kryptografia symetryczna

Klasyczny podział kryptograficznych algorytmów symetrycznych jest następujący ([42]):

1. **Algorytmy blokowe**. Typowy algorytm blokowy ma klucz o ściśle określonej długości i operuje na danych o z góry określonej długości. Jeśli ilość danych jest większa, dzieli się je na bloki odpowiedniej długości i każdy blok danych szyfruje z osobna. W wyniku tych operacji otrzymuje się sekwencję zaszyfrowanych bloków, które ustawione obok siebie tworzą szyfrogram.

2. **Algorytmy strumieniowe**. Algorytmy strumieniowe przetwarzają tekst otwarty w szyfrogram operując za każdym razem na pojedynczym elemencie wiadomości (najczęściej na bicie danych). Wynik pojedynczej operacji zależy od trzech czynników: elementu tekstu otwartego, elementu klucza oraz stanu algorytmu.

Klasyczne algorytmy strumieniowe wykorzystywały działanie XOR (różnicę symetryczną) na pojedynczych bitach danych oraz bitach generowanego *strumienia klucza*. Bit strumienia klucza zależał od bitu hasła, poprzedniego bitu szyfrogramu oraz globalnego „stanu maszyny szyfrującej" (sekwencji N ostatnich bitów szyfrogramu). Często wykorzystywano tu operacje generowania liczb pseudolosowych. Większość klasycznych algorytmów można było bardzo łatwo przełożyć na schematy elektroniczne, więc miały szerokie zastosowanie w rozwiązaniach sprzętowych.

W naszych rozważaniach będziemy się posługiwać nieco szerszą definicją algorytmu strumieniowego ([39]): *szyfry strumieniowe operują na pojedynczych cyfrach (elementach) tekstu otwartego przekształceniami zmiennymi w czasie.*

Wymagania Madrygi

Warto w tym miejscu przytoczyć wytyczne jakie algorytmom symetrycznym zaproponował W. E. Madryga. W pracy opisującej swój algorytm blokowy

([32]) sformułował następujące własności, jakie powinien spełniać dobry projekt szyfru symetrycznego:

1. Tekst otwarty nie może być wyprowadzony z szyfrogramu bez użycia klucza (innymi słowy algorytm jest bezpieczny).

2. Liczba operacji potrzebnych do wyznaczenia klucza z próbki danych otwartych oraz szyfrogramów powinna być statystycznie równa iloczynowi operacji potrzebnych do szyfrowania oraz możliwych kluczy (co oznacza, że żaden atak na tekst otwarty nie powinien być lepszy niż *atak brutalny*).

3. Wiedza o algorytmie nie powinna zmniejszać jego bezpieczeństwa (czyli całe bezpieczeństwo opiera się na kluczu).

4. Zmiana jednego bitu klucza powinna powodować znaczne zmiany szyfrogramu przy ustalonym tekście otwartym.

5. Podobnie, zmiana jednego bitu tekstu otwartego przy ustalonym kluczu powinna powodować znaczne zmiany szyfrogramu.

6. Redundantne grupy bitów tekstu otwartego powinny być zupełnie ukryte w szyfrogramie.

7. Długość szyfrogramu i tekstu otwartego powinny być jednakowe.

8. Nie powinno być żadnych prostych związków między dowolnymi możliwymi kluczami a szyfrogramami.

9. Każdy możliwy klucz powinien produkować silny szyfr (brak słabych kluczy).

10. Długość klucza oraz wiadomości otwartej powinny być regulowane, aby móc sprostać różnym wymogom bezpieczeństwa.

11. Algorytm powinien mieć własność możliwej efektywnej implementacji na różnych komputerach, rozwiązaniach sprzętowych oraz w logice dyskretnej.

W powyższym wykazie pominęliśmy punkty charakterystyczne jedynie dla algorytmów blokowych, dotyczące wykorzystania permutacji i podstawień jako podstawowych operacji szyfrowania.

1.1.4 Rodzaje ataków kryptologicznych

W kryptografii wyróżniamy cztery podstawowe typy ataków kryptoanalitycznych [42]. W każdym z nich zakłada się, że kryptoanalityk ma pełną wiedzę o zastosowanym algorytmie szyfrującym.

1. **Atak ze znanym szyfrogramem** (ang. *ciphertext-only attack*). Kryptoanalityk posiada kilka wiadomości, zaszyfrowanych tym samym algorytmem. Zadaniem kryptoanalityka jest odkrycie jak największej ilości tekstów jawnych lub nawet wydedukowanie klucza (kluczy) użytego przy szyfrowaniu.

   - Mając: $C_1 = E_K(P_1), C_2 = E_K(P_2), \ldots, C_i = E_K(P_i)$
   - Znaleźć: $P_1, P_2, \ldots P_i, K$ lub algorytm uzyskania P_{i+1} z $C_{i+1} = E_K(P_{i+1})$

2. **Atak ze znanym tekstem jawnym** (ang. *known-plaintext attack*). Kryptoanalityk posiada kilka wiadomości zaszyfrowanych oraz odpowiadających im tekstów jawnych. Jego zadaniem jest wydedukowanie klucza (kluczy) użytych przy szyfrowaniu lub wymyślenie algorytmu, który pozwoli odszyfrować każdą następną wiadomość, zaszyfrowaną tym samym kluczem (kluczami).

   - Mając: $P_1, C_1 = E_K(P_1), P_2, C_2 = E_K(P_2), \ldots, P_i, C_i = E_K(P_i)$
   - Znaleźć: K lub algorytm uzyskania P_{i+1} z $C_{i+1} = E_K(P_{i+1})$

3. **Atak z wybranym tekstem jawnym** (ang. *chosen-plaintext attack*). Kryptoanalityk posiada nie tylko pary wiadomości (tekst jawny, szyfrogram), ale może on dowolnie wybrać zbiór tekstów jawnych do zaszyfrowania. Jest to atak silniejszy niż poprzedni, ponieważ kryptoanalityk może wybrać specyficzne wiadomości, których zaszyfrowana postać może zdradzać pewne charakterystyczne elementy klucza. Zadaniem kryptoanalityka jest znalezienie klucza (kluczy) lub algorytmu, który pozwoli odszyfrować każdą następną wiadomość, zaszyfrowaną tym samym kluczem (kluczami)

   - Mając: $P_1, C_1 = E_K(P_1), P_2, C_2 = E_K(P_2), \ldots, P_i, C_i = E_K(P_i)$, gdzie kryptoanalityk wybiera $P_1, P_2, \ldots, P_i$
   - Znaleźć: K lub algorytm uzyskania P_{i+1} z $C_{i+1} = E_K(P_{i+1})$

4. **Atak z adaptacyjnie wybieranym tekstem jawnym** (ang. *adaptive-chosen-plaintext attack*). Jest to specjalny przypadek ataku z wybranym tekstem jawnym. Kryptoanalityk może nie tylko wybrać teksty jawne

do zaszyfrowania, ale może również zmieniać te teksty na podstawie poprzednich prób szyfrowania. W podstawowym ataku z wybranym tekstem jawnym kryptoanalityk może jedynie wybrać jednorazowo duży zbiór tekstów jawnych do zaszyfrowania. Przy podejściu adaptacyjnym na początku wybiera się mały zbiór tekstów, a następnie kolejne w oparciu o przewidywania i wnioski wyciągnięte z analizy poprzednich par.

W praktyce najczęściej występującymi atakami są ataki ze znanym tekstem jawnym oraz z wybranym tekstem jawnym.

1.2 Złożoność obliczeniowa

W tej sekcji przypomnimy podstawy złożoności obliczeniowej oraz kilka oznaczeń związanych z szacowaniem złożoności. Wykorzystamy wiedzę z pozycji [10, 18].

Złożoność algorytmu można rozumieć jako ilość zasobów niezbędnych do wykonania algorytmu. W zależności od rodzaju potrzebnych zasobów wyróżniamy:

1. **złożoność czasową** - czyli ilość czasu lub pojedynczych operacji arytmetycznych potrzebnych do wykonania algorytmu. Najczęściej wyraża się ją jako funkcję zależną od rozmiaru danych wejściowych, a jednostką są pojedyncze operacje;

2. **złożoność pamięciową** - czyli ilość pamięci komputerowej, jaka jest potrzebna w trakcie działania algorytmu. Tutaj również można stosować do oceny złożoności funkcję rozmiaru wejścia, choć czasem bywają również użyteczne rozmiary bezwzględne wyrażane w bajtach i bitach.

Złożoność obliczeniową algorytmu można rozpatrywać w kryteriach pozytywnych (ile zasobów potrzeba, aby wykonać algorytm) lub negatywnych (dowodzenie, że pewnych rzeczy nie da się zrobić przy zadanym kryterium złożoności).

Należy również wspomnieć o tym, że złożoność często zależy nie tylko od rozmiaru danych, ale jeszcze od ich struktury. W takim przypadku można rozpatrywać dwie różne złożoności: złożoność pesymistyczną (czyli najgorszy możliwy przypadek) oraz złożoność oczekiwaną (czyli uśrednienie wszystkich przypadków). Dobrym przykładem mogą tu być algorytmy sortowania, z których spora ilość działa średnio w czasie $O(n \log n)$, natomiast najgorsze przypadki danych powodują wydłużenie czasu do $O(n^2)$.

W kolejnej sekcji zdefiniujemy formalnie najczęściej wykorzystywaną do szacowania złożoności czasowej w informatyce notację asymptotyczną „wielkie O”.

Wspomnimy też o analogicznych notacjach, służących do szacowania algorytmu „od dołu" oraz dokładnego „rzędu algorytmu".

1.2.1 Notacje $O(f(n))$, $\Omega(f(n))$, $\Theta(f(n))$

Niech $f, g : \mathbb{N} \to \mathbb{R}^+$.

Definicja 1.1. Mówimy, że funkcja f jest co najwyżej rzędu g, jeśli

$$\exists_{C>0}\ \exists_{n_0\in\mathbb{N}}\ \ \forall_{n\geqslant n_0}\ f(n) \leqslant C \cdot g(n).$$

Własność tę zapisujemy $f(n) = O(g(n))$ lub krócej $f = O(g)$.

Należy pamiętać, że pomimo zastosowania znaku "=", powinniśmy ten zapis traktować jako nierówność i myśleć o nim, jako o oszacowaniu funkcji f od góry.

Definicja 1.2. Mówimy, że funkcja f jest co najmniej rzędu g, jeśli

$$\exists_{c>0}\ \exists_{n_0\in\mathbb{N}}\ \ \forall_{n\geqslant n_0}\ f(n) \geqslant c \cdot g(n).$$

Własność tę zapisujemy $f(n) = \Omega(g(n))$ lub krócej $f = \Omega(g)$.

Definicja 1.3. Mówimy, że funkcja f jest dokładnie rzędu g, jeśli

$$\exists_{c_1,c_2>0}\ \exists_{n_0\in\mathbb{N}}\ \forall_{n\geqslant n_0}\ \ c_1 \cdot g(n) \leqslant f(n) \leqslant c_2 \cdot g(n).$$

Własność tę zapisujemy $f(n) = \Theta(g(n))$ lub krócej $f = \Theta(g)$.

Zauważmy, że

$$f(n) = \Theta(g(n)),\ \text{gdy}\ f(n) = O(g(n))\ \text{i}\ f(n) = \Omega(g(n)).$$

1.3 Inżynieria oprogramowania

Inżynieria oprogramowania jest dziedziną inżynierii systemów zajmującą się wszystkimi praktycznymi aspektami produkcji oprogramowania. W wyniku dyskusji i publikacji z zakresu inżynierii oprogramowania powstał standard diagramów UML, a także rozwinęły się i zyskały popularność obiektowe języki programowania, które zdominowały dzisiejszy rynek usług programistycznych.

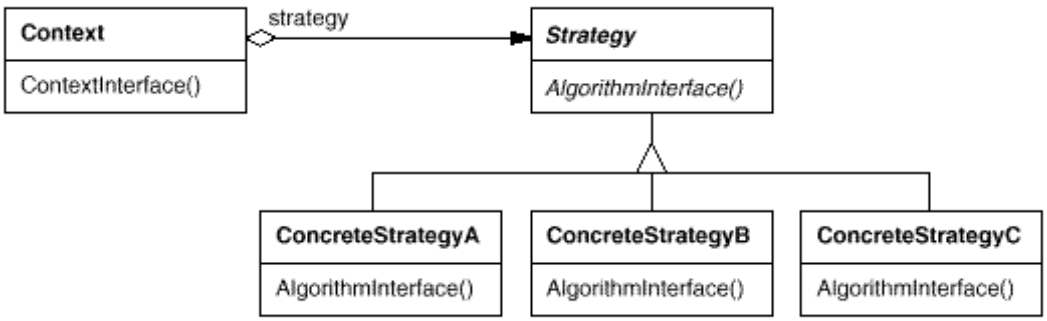
Inżynieria zajmuje się wszystkimi etapami wytwarzania oprogramowania, natomiast w tej sekcji przytoczymy jedynie kilka elementów z zakresu projektowania i implementacji.

1.3.1 Wybrane wzorce projektowe

Termin *wzorce projektowe* (ang. *design patterns*) został zapoczątkowany przez tzw. „bandę czworga", czyli autorów książki [17]. Termin został zaczerpnięty z architektury. Wzorce projektowe przedstawiają pewne uniwersalne rozwiązania wybranych zagadnień praktycznych programowania i projektowania obiektowego. Jak podkreślają autorzy wzorce nie były wymyślane, lecz odkrywane poprzez obserwację najlepiej sprawdzających się, powtarzalnych rozwiązań i późniejsze ich standaryzowanie. Przedstawimy tutaj trzy z puli ponad dwudziestu klasycznych wzorców projektowych.

Strategia

Wzorzec strategii ma bardzo prostą strukturę (rys. 1.1). Składa się z *klasy abstrakcyjnej* (czy też *interfejsu*), w którym zdefiniowany jest nagłówek jednej lub kilku funkcji wykonujących jakieś zadanie (na rys. 1.1 jest to klasa ***Strategy*** z metodą *AlgorithmInterface*). Zadanie to można zrealizować różnymi algorytmami, a każdy z tych algorytmów implementowany jest w oddzielnej klasie dziedziczącej ze ***Strategy*** (na rys. 1.1 **ConcreteStrategyA/B/C**) w swojej wersji metody AlgorithmInterface.



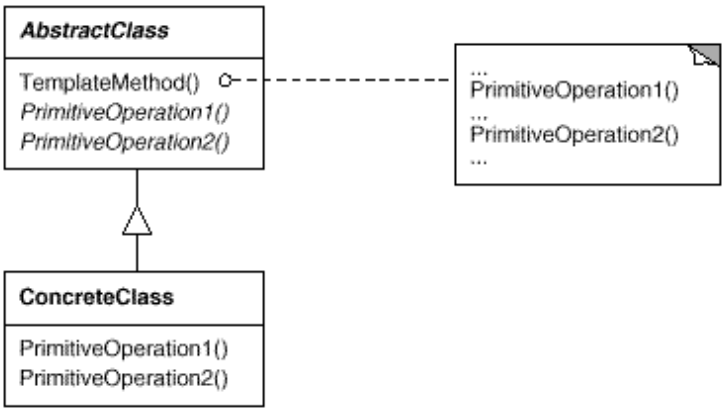
Rysunek 1.1: Diagram UML obrazujący klasyczny wzorzec Strategii

Wzorzec zawiera jeszcze klasę **Context**, która ma obiektowy „uchwyt" (realizowany przez zmienną wskaźnikową) do klasy reprezentującej strategię. W trakcie działania programu możliwe jest dynamiczne zmienienie strategii rozwiązywania danego problemu, poprzez ustawienie obiektu innej klasy dziedziczącej ze Strategy w tym uchwycie. Natomiast dzięki zastosowaniu abstrakcyjnego nagłówka metody i pracy w programie na wywołaniach tego abstrakcyjnego nagłówka, w żadnym innym miejscu programu nie ma potrzeby wprowadzania zmian w kodzie przy zmianie strategii.

Dodatkową zaletą tego wzorca jest bezbolesne wprowadzanie do projektu kolejnych (nowych) implementacji algorytmu realizującego jakieś zadanie. Przykładów zastosowania tego wzorca w praktyce jest mnóstwo. Możemy sobie wyobrazić różne algorytmy sortowania (jedna funkcja w klasie abstrakcyjnej i każdej pochodnej), czy parę algorytmów kryptograficznych *szyfruj–deszyfruj* realizującą jakiś konkretny rodzaj szyfru (dwie funkcje w klasie abstrakcyjnej i każdej pochodnej).

Metoda Szablonowa

Metoda Szablonowa (ang. *Template Method*) jest wzorcem projektowym z kategorii wzorców behawioralnych. Idea tego rozwiązania polega na zdefiniowaniu dosyć ogólnego algorytmu zamkniętego w metodzie klasy abstrakcyjnej (metoda TemplateMethod() w klasie ***AbstractClass*** z rys. 1.2). Algorytm ten składa się z pewnych abstrakcyjnych kroków (metody *PrimitiveOperation1* i *PrimitiveOperation2* na rysunku), które można realizować na różne sposoby.



Rysunek 1.2: Diagram UML obrazujący klasyczny wzorzec Metody Szablonowej

Każde konkretne rozwiązanie problemu (klasa **ConcreteClass** z rys. 1.2) musi dostarczyć implementacji tym abstrakcyjnym krokom i dzięki temu zrealizuje algorytm ogólny na swój specyficzny sposób. Dwa różne algorytmy będą się zatem różnić jedynie realizacją abstrakcyjnych kroków.

Najprostszym i dobrze przemawiającym do wyobraźni przykładem jest tutaj znowu algorytm sortowania. Na przykład można napisać ogólny algorytm sortowania szybkiego (lub dowolnego innego), w którym operacją abstrakcyjną będzie porównywanie dwóch elementów typu obiektowego. Każdy kto chciałby, aby elementy tworzonej przez niego klasy (konkretnego typu obiektowego)

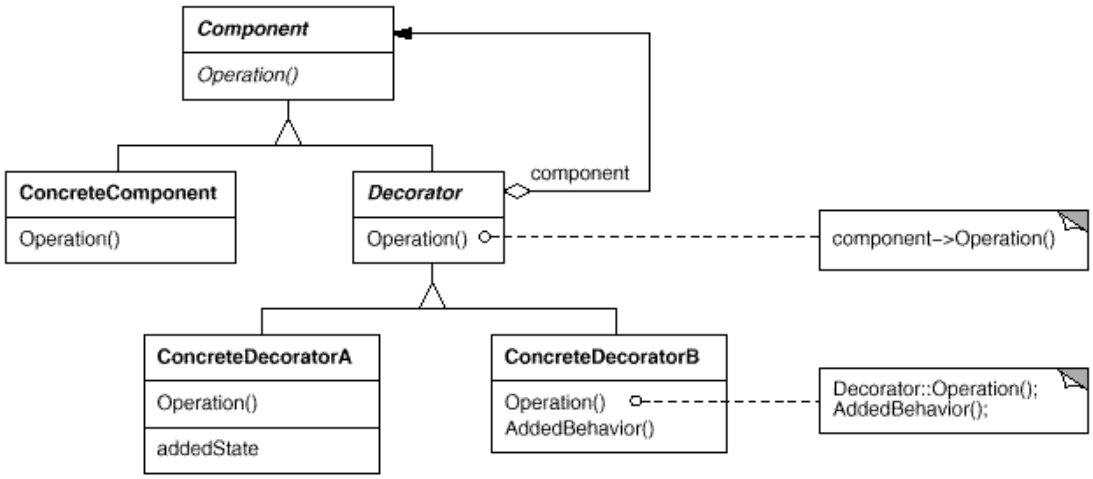
mogły być sortowane, musi zdefiniować operację porównania dwóch obiektów w swojej klasie zgodnie z nagłówkiem abstrakcyjnej metody, wykorzystywanej w operacji sortowania szybkiego.

Standardowe biblioteki języków obiektowych takich jak Java czy C# oferują właśnie takie rozwiązanie algorytmu sortowania. Operacja porównania używanego w sortowaniu powinna być zaznaczona poprzez implementację odpowiedniego interfejsu (w Javie *Comparable*). Ciekawostką jest możliwość deklarowania różnych sposobów porównywania obiektów tej samej klasy (w Javie poprzez różne implementację interfejsu *Comparator*) co jest oczywistym udogodnieniem, jeśli pomyślimy o klasach przechowujących dane osobowe i możliwościach sortowania ich alfabetycznie po nazwisku, po PESELu (dacie urodzenia) lub po jakimś identyfikatorze systemowym.

Dekorator

Wzorzec dekoratora na diagramie UML (rys. 1.3) wydaje się mieć bardziej złożoną strukturę niż przedstawione wcześniej wzorce strategii i metody szablonowej. Idea wzorca nie jest jednak skomplikowana.

Mamy klasę **ConcreteComponent** z metodą Operation, do której chcemy dodawać nowe zachowania. Aby nie trzeba było nadpisywać wielokrotnie tej klasy, zostaje wyciągnięta abstrakcja ***Component***, która jest implementowana między innymi przez naszą istniejącą klasę.



Rysunek 1.3: Diagram UML obrazujący klasyczny wzorzec Dekoratora

Kolejnym składnikiem tego systemu jest klasa abstrakcyjna ***Decorator*** (choć nie zawsze w rozwiązaniach przeznacza się na to klasę), która zawiera „uchwyt" do klasy dekorowanej Component, a z drugiej strony dziedziczy

po tej klasie. Mechanizm dekorowania polega na tym, że wywołanie metody Operation z konkretnego dekoratora skutkuje wywołaniem tej samej metody na obiekcie dekorowanym oraz dodaniem jakiegoś konkretnego pojedynczego zachowania (wywołanie funkcji lub zmiana stanu obiektu), charakterystycznego dla danego dekoratora.

Dzięki takiej strukturze klas konkretną realizację klasy ***Component*** możemy wielokrotnie dekorować w prosty sposób. Aby uzyskać dodatkowe zachowanie jakie oferuje pojednyczy dekorator, wystarczy stworzyć obiekt tego dekoratora, podając w parametrze konstruktora obiekt dekorowany. Następnie wszędzie tam, gdzie używany był obiekt dekorowany, należy używać tego stworzonego obiektu dekoratora. W tym celu najczęściej wystarczy w jednym miejscu podstawić nowo powstały obiekt pod zmienną typu abstrakcyjnego Component (pod którą można też podstawić ConcreteComponent) i dalej wykonywać operacje na tej zmiennej abstrakcyjnej bez rozpatrywania w programie czy jest to komponent konkretny czy udekorowany. Zauważmy, że taki schemat organizacji klas pozwala pojedynczy obiekt dekorować wielokrotnie (składać dekoratory ze sobą) w różnych konfiguracjach i kolejnościach, które mozna ustawiać nawet na etapie wykonywania programu.

Klasycznym przykładem wykorzystania wzorca dekorator jest implementacja strumieni w standardowej bibliotece Javy w pakiecie **java.io**. Dzięki temu uzyskujemy ten sam schemat postępowania przy zapisie danych tekstowych na ekran (konsolowy), do pliku, lub do połączenia sieciowego wykorzystującego gniazda (różne implementacje konkretne klasy abstrakcyjnej ***Writer*** lub ***OutputStream***). Jeśli chcemy mieć dodatkowo nasze dane w strumieniu buforowane, dokładamy konstruktor klasy odpowiadający za buforowanie (klasa dekorująca **BufferedWriter**). Jeśli chcemy je mieć spakowane czy zaszyfrowane (jakimś prostym szyfrem), dokładamy odpowiednie konstruktory przy tworzeniu obiektu. Każdy dekorator dokłada swoją operację, a my pracujemy na zmiennej typu ogólnego „strumień" bez dalszego wnikania czym został ten strumień udekorowany. Jeśli zależy nam na używaniu metody charakterystycznej dla któregoś dekoratora, zmieniamy jedynie typ deklarowanej zmiennej obiektowej (częsty przykład praktyczny to klasa **BufferedReader** z biblioteki Javy, która oferuje metodę **readLine()**, pozwalają odczytać jedną linię tekstu ze strumienia, czego nie oferuje klasa ***Reader***).

Rozdział 2

Algebraiczne aspekty kryptografii

W tym rozdziale przytoczymy fakty z algebry, które mają istotne znaczenie w kryptografii. Wiedza ta jest potrzebna dla celów konstrukcji algorytmów, dowodzenia ich poprawności i bezpieczeństwa, a także dla konstruowania przemyślanych ataków kryptologicznych. Wykorzystaliśmy wiedzę z pozycji [8, 27, 31, 49].

2.1 Oznaczenia obiektów algebraicznych

2.1.1 Grupy

Definicja 2.1. Niech G będzie zbiorem i niech $\bullet$ będzie działaniem na elementach tego zbioru. System algebraiczny $\langle G, \bullet \rangle$, nazywamy *grupą*, jeśli spełnione są następujące warunki:

1. działanie $\bullet$ jest łączne

$$\forall_{a,b,c \in G} \quad [(a \bullet b) \bullet c = a \bullet (b \bullet c)];$$

2. istnieje w grupie *element neutralny* e działania $\bullet$

$$\exists_{e \in G} \quad \forall_{a \in G} \quad [a \bullet e = e \bullet a = a];$$

3. każdemu elementowi grupy można przyporządkować w grupie *element odwrotny* działania:

$$\forall_{a \in G} \quad \exists_{\tilde{a} \in G} \quad [a \bullet \tilde{a} = \tilde{a} \bullet a = e].$$

4. Jeśli dodatkowo

$$\forall_{a,b\in G} \ \ [a \bullet b = b \bullet a],$$

to grupa nazywa się *grupą przemienną* lub *abelową*.

Jeśli operacja w grupie ma symbol + i nazywa się dodawaniem, to grupę nazywa się *grupą addytywną*. Wtedy element $\tilde{a}$ oznaczany jest jako $-a$ i nazywany jest *elementem przeciwnym* do a. Element neutralny grupy addytywnej oznacza się *0* i nazywa *zerem grupy*. W grupie multyplikatywnej operacja grupowa nazywa się mnożeniem i oznaczana jest przeważnie symbolem $\cdot$. Element $\tilde{a}$ nazywa się *elementem odwrotnym* do a i oznacza symbolem a^{-1}. Element neutralny grupy multyplikatywnej zwykle oznaczany jest symbolem *1* i nosi nazwę *jedności grupy*.

Definicja 2.2. Jeśli grupa składa się ze skończonej liczby elementów, wówczas nosi ona nazwę *grupy skończonej*. Liczba elementów grupy skończonej nazywa się *rzędem grupy*. Rząd grupy oznacza się symbolem $|G|$ lub $card\ G$.

Definicja 2.3. Mówimy, że grupa multyplikatywna jest grupą *cykliczną*, jeśli

$$\exists_{g\in G} \ \ \forall_{a\in G} \ \ \exists_{j\in\mathbb{N}} \ [a = g^j].$$

Element g nazywa się wtedy *generatorem* grupy cyklicznej i zapisuje $G = \langle g \rangle$. Wprost z definicji wynika przemienność każdej grupy cyklicznej oraz $g^{|G|} = 1$.

Definicja 2.4. *Rząd multyplikatywny* s elementu a skończonej grupy multyplikatywnej definiuje się następująco:

$$s = min\{m \in \mathbb{N} : a^m = 1\}.$$

Liczba s jest dzielnikiem rzędu grupy, czyli $s \mid card\ G$.

Definicja 2.5. System algebraiczny $\langle H, \bullet \rangle$, złożony ze zbioru $H \subset G$ i operacji zdefiniowanej w grupie $\langle G, \bullet \rangle$ nazywamy *podgrupą* grupy G, jeśli spełnia wszystkie aksjomaty grupy.

Trywialnymi podgrupami każdej grupy są: $\langle e, \bullet \rangle$ i sama grupa G.

Definicja 2.6. Niech będą dane dwie grupy $(G, \bullet), (H, \star)$. *Homomorfizmem grup* nazywamy funkcję $h : G \to H$, taką, że

$$\forall_{a,b\in G} \ \ [h(a \bullet b) = h(a) \star h(b)].$$

Jako oczywisty wniosek z definicji mamy przekształcanie przez h elementów neutralnych na siebie oraz przekształcanie elementów przeciwnych na siebie

$$h(u^{-1}) = h(u)^{-1}.$$

2.1.2 Pierścienie i ciała

Definicja 2.7. System algebraiczny $\langle \mathbb{K}, +, \cdot \rangle$ nazywa się *pierścieniem*, jeśli spełnione są aksjomaty

1. system $\langle \mathbb{K}, + \rangle$ jest grupą abelową;
2. działanie $\cdot$ jest łączne;
3. działanie $\cdot$ jest rozdzielne względem działania $+$, tzn.

$$\forall_{a,b,c \in \mathbb{K}} \ \ [a \cdot (b + c) = a \cdot b + a \cdot c] \wedge [(b + c) \cdot a = b \cdot a + c \cdot a].$$

Definicja 2.8. Mówimy, że element a pierścienia $\mathbb{K}$ jest *dzielnikiem zera*, jeśli

$$\exists_{b \in \mathbb{K}, b \neq 0} \ \ [a \cdot b = 0 \vee b \cdot a = 0].$$

W zależności od własności mnożenia pierścienie kwalifikuje się następująco:

(a) Jeśli $\exists_{e \in \mathbb{K}} \ \forall_{a \in \mathbb{K}} \ \ [a \cdot e = e \cdot a]$, to pierścień nosi nazwę *pierścienia z jedynką*.

(b) Jeśli mnożenie w pierścieniu jest przemienne, pierścień nazywamy *pierścieniem przemiennym*.

(c) Jeśli pierścień

   (a) jest przemienny;
   (b) posiada element neutralny mnożenia $e \neq 0$;
   (c) jeśli $\forall_{a,b \in \mathbb{K}} \ \ [a \cdot b = 0 \implies (a = 0) \vee (b = 0)]$,

   wówczas pierścień nazywamy *dziedziną całkowitości* lub *pierścieniem całkowitym*. Element neutralny mnożenia zwykle oznacza się symbolem *1*. W pierścieniu całkowitym nie istnieją *dzielniki zera*.

(d) Pierścień całkowity nazywa się *pierścieniem euklidesowym*, jeśli określona jest funkcja $v : \mathbb{K} \backslash \{0\} \to \mathbb{N} \cup \{0\}$ o własności:

$$\forall_{a,b \in \mathbb{K}, a,b \neq 0} \ \ \exists q, r \in \mathbb{K} \ \ \Big[\Big(a = b \cdot q + r\Big) \wedge \Big(v(r) < v(b) \vee r = 0\Big)\Big].$$

   Funkcję v nazywamy *normą* pierścienia, natomiast element pierścienia r nazywamy *resztą*.

(e) Jeśli system $\langle \mathbb{K} \backslash \{0\}, \cdot \rangle$ jest grupą, to pierścień nazywa się *pierścieniem z dzieleniem*.

(f) Pierścień przemienny z dzieleniem nazywa się *ciałem*.

Ciało można też zdefiniować w sposób następujący:

Definicja 2.9. System $\langle \mathbb{F}, +, \cdot \rangle$ nazywa się ciałem, jeśli

1. system $\langle \mathbb{F}, + \rangle$ jest grupą przemienną z elementem neutralnym oznaczonym symbolem *0*;
2. system $\langle \mathbb{F} \backslash \{0\}, \cdot \rangle$ jest grupą przemienną z elementem neutralnym oznaczonym symbolem *1*;
3. mnożenie jest rozdzielne względem dodawania, czyli spełniona jest zależność

$$\forall_{a,b,c \in \mathbb{F}} \quad [a \cdot (b + c) = a \cdot b + a \cdot c] \wedge [(b + c) \cdot a = b \cdot a + c \cdot a].$$

Liczba elementów ciała może być nieskończona lub równa potędze liczby pierwszej. Ciało o skończonej liczbie elementów nazywa się *ciałem skończonym* albo *ciałem Galois* i oznacza $\mathbb{GF}(q)$, gdzie q oznacza ilość elementów ciała.

Definicja 2.10. Niezerowy element e pierścienia $\mathbb{K}$ nazywamy *regularnym*, jeśli nie jest lewostronnym, ani prawostronnym dzielnikiem zera. Zbiór wszystkich elementów regularnych pierścienia $\mathbb{K}$ oznaczamy $Reg(\mathbb{K})$.

Definicja 2.11. System algebraiczny $\langle S, +, \cdot \rangle$, gdzie $S \subset \mathbb{K}$, nazywa się *podpierścieniem* pierścienia $\langle \mathbb{K}, +, \cdot \rangle$, jeśli spełnia wszystkie aksjomaty pierścienia.

Definicja 2.12. System algebraiczny $\langle I, +, \cdot \rangle$, gdzie $I \subset \mathbb{K}$, nazywa się *ideałem* pierścienia $\langle \mathbb{K}, +, \cdot \rangle$, jeśli system ten jest podpierścieniem rozpatrywanego pierścienia i zachodzi:

$$\forall_{a \in I} \quad \forall_{r \in \mathbb{K}} \; [(a \cdot r \in I) \wedge (r \cdot a \in I)].$$

Definicja 2.13. Ideał $\langle I, +, \cdot \rangle$ nazywa się *ideałem głównym*, jeśli istnieje element $a \in \mathbb{K}$ taki, że $I = (a)$, gdzie (a) oznacza zbiór $\mathbb{K} \cdot a$. Mówi się, że element a jest *generatorem* ideału (a). Jeśli każdy ideał $\langle I, +, \cdot \rangle$ w pierścieniu $\langle \mathbb{K}, +, \cdot \rangle$ jest ideałem głównym, to pierścień $\langle \mathbb{K}, +, \cdot \rangle$ nazywa się *pierścieniem głównym*.

Przykład 2.1. *Jako przykłady pierścieni rozpatrzmy następujące systemy:*

- *Zbiór liczb całkowitych $\mathbb{Z}$ z arytmetycznymi operacjami dodawania i mnożenia jest pierścieniem.*

- *Niech* (5) *oznacza zbiór wszystkich wielokrotności liczby 5:*

$$(5) = \{\ldots, -20, -15, -10, -5, 0, 5, 10, 15, 20, \ldots\}.$$

Można sprawdzić, że system $\langle (5), +, \cdot \rangle$ *jest ideałem w pierścieniu* $\langle \mathbb{Z}, +, \cdot \rangle$. *Istnieje następujący rozkład tego pierścienia na klasy reszt względem ideału (5):*

$$0 = 0 + (5)$$
$$1 = 1 + (5)$$
$$2 = 2 + (5)$$
$$3 = 3 + (5)$$
$$4 = 4 + (5)$$

System $\langle \mathbb{Z}/(5), \oplus, \odot \rangle$ *jest pierścieniem.*

- *Podobnie system* $\langle \mathbb{Z}/(4), \oplus, \odot \rangle$ *jest pierścieniem. W tym pierścieniu* $\{2\} \odot \{2\} = \{0\}$*, co oznacza, że klasa reszta* $\{2\}$ *jest dzielnikiem zera.*

Pierścień $\langle \mathbb{Z}/(n), \oplus, \odot \rangle$ istnieje dla każdej liczby naturalnej n i jest izomorficzny z pierścieniem $\langle \mathbb{Z}_n, \oplus, \odot \rangle$, czyli zbiorem liczb $\{0, 1, \ldots, n-1\}$ z działaniami dodawania i mnożenia modulo n.

Twierdzenie 2.1. *Pierścień* $\langle \mathbb{Z}/(p), \oplus, \odot \rangle$ *jest ciałem skończonym wtedy i tylko wtedy, gdy* p *jest liczbą pierwszą.*

Definicja 2.14. Niech p oznacza liczbę pierwszą, niech

$$\mathbb{GF}(p) = \langle \mathbb{F}/(p), \oplus, \odot \rangle, \quad \mathbb{F}(p) = \{0, 1, \ldots, p-1\},$$

i niech β będzie takim odwzorowaniem, że:

$$\forall_{a \in \mathbb{F}(p)} \ [\beta(\{a\}) = a].$$

Wówczas $\mathbb{GF}(p)$ jest ciałem skończonym nazywanym *ciałem Galois* o p elementach. Można zatem napisać $\mathbb{GF}(p) = \langle \mathbb{Z}_p, \oplus, \odot \rangle$. Jest to jedyne liczbowe ciało Galois. Działaniami w tym ciele są dodawanie i mnożenie liczb ze zbioru $\mathbb{Z}_p$ modulo p.

Definicja 2.15. Jeśli $\langle \mathbb{K}, +, \cdot \rangle$ jest dowolnym pierścieniem i jeśli

$$\exists_{n \in \mathbb{N}} \ \ \forall_{r \in \mathbb{K}} \ \ [n \cdot r = 0],$$

to najmniejsza taka liczba n nazywa się *charakterystyką pierścienia*. Mówi się wtedy, że pierścień posiada charakterystykę n. Jeśli liczba n o podanej tu własności nie istnieje, pierścień posiada charakterystykę 0.

Warto w tym miejscu przytoczyć dwa klasyczne twierdzenia dotyczące charakterystyk wybranych obiektów algebraicznych.

Twierdzenie 2.2. *Charakterystyka pierścienia $\langle \mathbb{K}, +, \cdot \rangle$ z jednością, w którym $|\mathbb{K}| \neq 0$, nieposiadającego dzielników zera równa jest liczbie pierwszej p. W takim pierścieniu zachodzi:*

$$\forall_{a,b \in \mathbb{K}} \ \forall_{n \in \mathbb{N}} \ \ [(a \pm b)^{p^n} = a^{p^n} \pm b^{p^n}]$$

Twierdzenie 2.3. *Charakterystyka p ciała skończonego jest liczbą pierwszą.*

Twierdzenie 2.4. *Niech a będzie niezerowym elementem rzędu n ciała skończonego $\mathbb{GF}(q)$. Wtedy liczba n jest podzielnikiem $(q-1)$.*

2.1.3 Pierścień wielomianów

Niech $\langle \mathbb{K}, +, \cdot \rangle$ będzie dowolnym pierścieniem. *Wielomianem* nad tym pierścieniem nazywa się wyrażenie postaci

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + \ldots + a_n x^n, \tag{2.1}$$

gdzie $n \in \mathbb{N}, a_i \in \mathbb{K}, 0 \leqslant i \leqslant n$. Symbol $x \notin \mathbb{K}$ nazywa się *zmienną niezależną*. Niech

$$f(x) = \sum_{i=0}^{n} a_i x^i, g(x) = \sum_{i=0}^{m} b_i x^i, m \leqslant n.$$

Gdy $n = m$ to warunek równości wielomianów zapisujemy następująco:

$$f(x) = g(x) \iff a_i = b_i, 0 \leqslant i \leqslant n.$$

Można też zdefiniować w oczywisty sposób sumę i iloczyn wielomianów:

$$f(x) + g(x) = \sum_{i=0}^{n} (a_i + b_i) x^i,$$

$$f(x)g(x) = \sum_{k=0}^{n+m} c_k x^k, c_k = \sum_{i+j=k} a_i b_j, \ 0 \leqslant i \leqslant n, \ 0 \leqslant j \leqslant m.$$

Definicja 2.16. Zbiór wszystkich wielomianów nad pierścieniem $\langle \mathbb{K}, +, \cdot \rangle$ wraz z operacjami dodawania i mnożenia wielomianów nazywa się pierścieniem wielomianów i oznacza symbolem $\mathbb{K}[x]$.

Jeśli we wzorze (2.1) mamy $a_n \neq 0$, to liczbę n nazywa się *stopniem* wielomianu $f(x)$, co zapisujemy jako $\deg(f(x)) = n$. Jeśli pierścień $\mathbb{K}$ posiada element neutralny mnożenia, oznaczony jako 1, i jeśli najwyższy współczynnik wielomianu równy jest 1, to taki wielomian nazywa się wielomianem *monicznym* lub *unormowanym*. Jeśli $\deg(f(x)) = 0$, to wielomian $f(x)$ nazywa się wielomianem *stałym*. Współczynnik a_0 nazywa się *wyrazem wolnym*. Przyjmuje się umownie, że stopień wielomianu zerowego $f(x) = 0$ wynosi $-\infty$.

Wielomiany stałe są elementami pierścienia $\mathbb{K}$ należącymi do pierścienia $\mathbb{K}[x]$. Stąd pierścień $\mathbb{K}$ jest podpierścieniem $\mathbb{K}[x]$.

Nieraz rozpatruje się pierścienie wielomianów nad ciałem $\mathbb{F}$, czyli systemy oznaczane $\mathbb{F}[x]$. Każde ciało jest dziedziną całkowitości. Jeśli więc $f(x), g(x) \in \mathbb{F}[x]$, to

$$\deg(f(x) + g(x)) \leqslant \max[\deg(f(x)), \deg(g(x))]$$

oraz

$$\deg(f(x)g(x)) = \deg(f(x)) + \deg(g(x)).$$

Podobnie jak w przypadku dzielenia w pierścieniu $\langle \mathbb{Z}, +, \cdot \rangle$, mówi się że wielomian $g(x) \in \mathbb{F}[x]$ dzieli wielomian $f(x) \in \mathbb{F}[x]$, jeśli istnieje taki wielomian $h(x) \in \mathbb{F}[x]$, że $f(x) = g(x)h(x)$.

Definicja 2.17. Element $b \in \mathbb{F}[x]$ nazywa się *pierwiastkiem* lub *zerem* wielomianu $f(x) \in \mathbb{F}[x]$, jeśli $f(b) = 0$.

Twierdzenie 2.5. *Element $b \in \mathbb{F}[x]$ jest pierwiastkiem wielomianu $f(x) \in \mathbb{F}[x]$ wtedy i tylko wtedy, gdy $(x - b)|f(x)$.*

Definicja 2.18. Wielomian $f(x) \in \mathbb{F}[x]$ nazywa się *nieprzywiedlnym* (lub *nierozkładalnym*) nad ciałem $\mathbb{F}[x]$, jeśli spełnione są warunki

1. $\deg(f(x)) > 0$
2. jeśli $f(x) = a(x)b(x),\ a(x), b(x) \in \mathbb{F}[x]$, to $a(x)$ lub $b(x)$ jest wielomianem stałym.

Twierdzenie 2.6. *Niech $f(x) \in \mathbb{F}[x]$. Pierścień klas reszt $\mathbb{F}[x]/(f)$ jest ciałem wtedy i tylko wtedy, gdy f(x) jest wielomianem nierozkładalnym nad $\mathbb{F}[x]$.*

Pokażemy teraz w jaki sposób wykorzystując ostatnie twierdzenie skonstruować ciało skończone $\mathbb{GF}(q)$, gdzie q będzie pewną potęgą liczby pierwszej.

Przykład 2.2. *Chcemy utworzyć ciało skończone $\mathbb{GF}(8) = \mathbb{GF}(2^3)$. Rozpatrzmy pierścień wielomianów nad $\mathbb{GF}(2)$, czyli o współczynnikach ze zbioru $\{0, 1\}$. Wybieramy wielomian nierozkładalny stopnia 3 nad $\mathbb{GF}(2)$: $f(x) = x^3 + x + 1$. Następnie dokonujemy rozkładu pierścienia $\mathbb{GF}(2)[x]$ na klasy reszt względem*

ideału generowanego przez $f(x)$. Ideałem będzie zbiór wielomianów podzielnych przez $f(x)$. Można więc utworzyć następujące klasy reszt:

$$\begin{aligned}
\{0\} &= 0 + (f), \\
\{1\} &= 1 + (f), \\
\{\alpha\} &= x + (f), \\
\{\beta\} &= 1 + x + (f), \\
\{\gamma\} &= x^2 + (f), \\
\{\delta\} &= 1 + x^2 (f), \\
\{\eta\} &= x + x^2 (f), \\
\{\kappa\} &= 1 + x + x^2 (f).
\end{aligned}$$

Wielomian $f(x)$ jest nieprzywiedlny nad $\mathbb{GF}(2)$, ponieważ $f(0) \neq 0$ oraz $f(1) \neq 0$, więc pierścień klas reszt $\langle \mathbb{F}_8, +, \cdot \rangle$, gdzie

$$\mathbb{F}_8 = \{\{0\}, \{1\}, \{\alpha\}, \{\beta\}, \{\gamma\}, \{\delta\}, \{\eta\}, \{\kappa\}\}$$

jest zgodnie z twierdzeniem 2.6, ciałem $\mathbb{GF}(8)$.

Działanie dodawania w tym ciele wykonujemy naturalnie, tak jak w pierścieniu $\mathbb{GF}(2)[x]$. Natomiast mnożenie wymaga wykonania mnożenia wielomianów w $\mathbb{GF}(2)[x]$, a następnie podzielenia wyniku przez $f(x)$. Wynikiem mnożenia w $\mathbb{GF}(8)$ jest reszta z ostatniego dzielenia.

Ciała skończone $\mathbb{GF}(2^n)$ mają szerokie zastosowanie w kryptografii. Przeważnie wykorzystuje się właśnie ich wielomianową reprezentację. Należy pamiętać, że szyfry symetryczne mają być szybkie, a wykonywanie dzielenia wielomianów w dowolnej reprezentacji będzie bardzo niekorzystnie wpływać na wydajność działania mnożenia. Aby nie zmagać się z tym problemem, w praktyce generuje się tabele działań + oraz $\cdot$, wykonując działania na wielomianach tylko raz dla każdej operacji lub sczytuje te tabele z pliku. W trakcie działania algorytmów kryptograficznych każda operacja arytmetyczna w ciele skończonym wymaga jedynie wyszukania wartości w tabeli. Jest to operacja o niewielkim czasie stałym, lecz dłuższa niż działanie w pierścieniu modulo $\langle \mathbb{Z}_{2^n}, \oplus, \odot \rangle$, który utożsamiamy w reprezentacji komputerowej z działaniami na liczbach całkowitych bez znaku.

Warto w tym miejscu wspomnieć, iż tabelaryzacja działań ma ograniczenie związane z pamięcią. Chcąc trzymać w pamięci tabele działań dla $\mathbb{GF}(256)$ potrzebujemy dwóch tabel o rozmiarze 256×256 bajtów

$$2 * 256 * 256 \ B = 2 * 2^8 * 2^8 \ B = 2^{17} \ B = 128 \ KB.$$

Co jest rozmiarem niedużym i jak najbardziej akceptowalnym. Dla liczb o reprezentacji 2-bajtowej ($\mathbb{GF}(2^{16})$) będziemy potrzebowali

$$2 * 2^{16} * 2^{16} * 2\ B = 2^{34}\ B = 4\ GB$$

na same tabele działań. A 4 Gigabajty pamięci RAM to na dzień dzisiejszy często cała pamięć operacyjna średniej jakości komputerów. Dla liczb czteroi ośmiobajtowych o tabelaryzacji działań nie ma sensu mówić.

Wniosek jaki płynie z tych rozważań jest następujący: na komputerach dzisiejszej klasy możemy sprawnie wykorzystywać w algorytmach działania w ciałach skończonych $\mathbb{GF}(2^8)$. Natomiast jeśli wystarcza nam pierścień przemienny z jedynką, mamy do dyspozycji $\mathbb{Z}_{2^8}$, $\mathbb{Z}_{2^{16}}$, $\mathbb{Z}_{2^{32}}$, $\mathbb{Z}_{2^{64}}$, czyli arytmetykę 1, 2, 4 lub 8-bajtową.

2.1.4 Przestrzenie wektorowe, przestrzenie afiniczne, moduł

Zacznijmy od definicji przestrzeni wektorowej oraz kilku struktur algebraicznych będących uogólnieniem przestrzeni wektorowej.

Definicja 2.19. Niech $\mathbb{F}$ będzie dowolnym ciałem. *Przestrzenią wektorową* (lub liniową) nad ciałem $\mathbb{F}$ nazywamy zbiór V, będący grupą abelową addytywną, z działaniami

- dodawania wektorów: $V \times V \to V$, oznaczanym $\mathbf{v} + \mathbf{w}$, gdzie $\mathbf{v}, \mathbf{w} \in V$,
- mnożeniem przez skalar: $\mathbb{F} \times V \to V$, oznaczanym $a\,\mathbf{v}$, gdzie $a \in \mathbb{F},\ \mathbf{v} \in V$,

które dla dowolnych *skalarów* $a, b \in \mathbb{F}$, oraz *wektorów* $\mathbf{v}, \mathbf{w} \in V$ spełniają poniższe aksjomaty:

1. $a(\mathbf{v} + \mathbf{w}) = a\mathbf{v} + a\mathbf{w}$,
2. $(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$,
3. $a(b\mathbf{v}) = (a \cdot b)\mathbf{v}$,
4. $\exists 1 \in \mathbb{F}\ \ (1\mathbf{v} = \mathbf{v})$, gdzie 1 jest elementem neutralnym mnożenia w ciele $\mathbb{F}$.

Definicja 2.20. Mówimy, że podzbiór wektorów $S = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ przestrzeni wektorowej V nad ciałem $\mathbb{K}$ jest *liniowo niezależny*, jeśli

$$a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \ldots + a_n\mathbf{v}_n = \mathbf{0} \ \Rightarrow\ a_1 = a_2 = \ldots = a_n = 0,$$

gdzie $a_i, 0 \in \mathbb{K}, \mathbf{0} \in V$. Jeśli istnieje inne rozwiązanie powyższej równości, wektory nazywamy *liniowo zależnymi*.

Równania, które można sprowadzić do postaci:

$$\sum_i a_i x_i = b, \quad \text{gdzie } a_i, b \in \mathbb{K}, x_i \in V$$

nazywamy *równaniami liniowymi*. Układy takich równań, nazywamy układami liniowymi.

Przestrzeń afiniczna może być intuicyjnie rozumiana jako przestrzeń wektorowa bez punktu początkowego. Natomiast moduł jest uogólnieniem przestrzeni liniowej nad dowolnym pierścieniem. Wolny moduł jest modułem, który posiada bazę, czyli niezależny liniowo zbiór generujący.

Definicja 2.21. Odwzorowanie $F = (F_1, F_2, \ldots, F_n) : \mathbb{K}^n \to \mathbb{K}^n$ jest *wielomianowe*, jeśli $F_1, \ldots, F_n \in \mathbb{K}[x_1, \ldots x_n]$.

Definicja 2.22. *Stopniem* odwzorowania wielomianowego $F = (F_1, F_2, \ldots, F_n) : \mathbb{K}^n \to \mathbb{K}^n$ nazywamy liczbę

$$\deg(F) = \max_{1 \leqslant i \leqslant n} (\deg(F_i)).$$

Przykład 2.3. *Odwzorowanie wielomianowe stopnia 1 nazywamy* przekształceniem afinicznym. *Można je macierzowo zapisać*

$$\mathbf{y} = \mathbf{A}\ \mathbf{x}^T + \mathbf{b},$$

gdzie $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{b} \in \mathbb{K}^n$, $\boldsymbol{A} \in \mathbb{K}^n \times \mathbb{K}^n$. *Jeśli wektor* $\boldsymbol{b}$ *redukuje się do 0 mówimy o* przekształceniu liniowym.

Definicja 2.23. Odwzorowanie wielomianowe $F : \mathbb{K}^n \to \mathbb{K}^n$ jest *automorfizmem wielomianowym* przestrzeni $\mathbb{K}^n$, jeśli jest bijekcją, a odwzorowanie odwrotne F^{-1} jest wielomianowe.

2.2 Kryptografia wielu zmiennych

Kryptografia wielu zmiennych (ang. *Multivariate Cryptography*) jest stosunkowo nową dziedziną w kryptografii asymetrycznej. W naszych badaniach skupiamy się na algorytmach symetrycznych, narzuca się więc pytanie: w jakim celu opisywać elementy tej kryptografii wielu zmiennych?

Odpowiedź znajdziemy w dalszej części pracy, gdy zaprezentujemy rozszerzony algorytm kryptograficzny, wykorzystujący schemat podobny do idei Imai-Matsumoto, a także, gdy przeprowadzimy wstępną analizę kryptologiczną naszego algorytmu. Przyjrzyjmy się typowemu schematowi kryptografii wielu zmiennych ([13, 41]).

Niech $\mathbb{K}$ będzie ciałem skończonym lub pierścień przemiennym z jedynką. Tekst otwarty $x = (x_1, \ldots, x_n)$ jest wektorem (krotką) nad $\mathbb{K}$. Funkcja szyfrująca P jest dana układem równań wielomianowych m zmiennych nad $\mathbb{K}$ o niewielkim stopniu (najczęściej 2 lub 3). Na przykład:

$$\begin{aligned} P_1(x) : y_1 &= x_1x_2 + x_1x_3 \\ P_2(x) : y_2 &= x_2x_4 + x_2x_5 \\ &\vdots \\ P_m(x) : y_m &= x_nx_1 + x_nx_{n-1} + x_1x_3 \end{aligned}$$

W ten sposób otrzymujemy szyfrogram, będący wektorem $y = (y_1, \ldots, y_m)$.

Przekształcenie publiczne P najczęściej buduje się z trzech odwracalnych przekształceń ukrytych, stanowiących klucz prywatny: przekształcenia wielomianowego P' (stopnia 2 lub 3) oraz dwóch przekształceń liniowych lub afinicznych S i T (stopnia 1) według schematu:

1. $S(x) = w$;
2. $P'(w) = u$;
3. $T(u) = y$.

W rzeczywistości $P = S \circ P' \circ T$, gdzie $\circ$ jest operatorem składania przekształceń, a pierwsze oddziałuje na wektorze przekształcenie po lewej stronie (S).

Ponieważ przekształcenia S, P', T są odwracalne, to schemat deszyfrowania polega na zastosowaniu przekształceń odwrotnych w odwrotnej kolejności:

1. $T^{-1}(y) = u$;
2. $P'^{-1}(u) = w$;
3. $S^{-1}(w) = x$.

Bezpieczeństwo kryptosystemu tego typu polega na trudności znalezienia przekształcenia odwrotnego do P bez znajomości przekształceń składowych P, gdy podajemy klucz publiczny P w formie równań wielu zmiennych. W praktyce łamanie szyfru przekłada się na rozwiązanie układu równań

$$P \begin{cases} p_1(x_1, \ldots, x_n) &= y_1 \\ \vdots & \\ p_m(x_1, \ldots, x_n) &= y_m \end{cases}$$

i znalezieniu rozwiązania $x = (x_1, \ldots, x_n)$, gdy znane jest P oraz $y = (y_1, \ldots, y_m)$.

Jednym z pierwszych przykładów kryptosystemu wielu zmiennych był kryptosystem Imai-Matsumoto przedstawiony w roku 1988 ([33]). Kryptoanalizę tego systemu zaprezentował J. Patarin ([36]), proponując w kolejnych latach swoje systemy kryptograficzne ([37, 38]), używając w jednym z nich terminu „multivariate cryptography".

Rozdział 3

Elementy teorii grafów oraz grafów algebraicznych

3.1 Podstawy teorii grafów

W tej sekcji zdefiniujemy podstawowe elementy teorii grafów, przytoczymy kilka klasycznych twierdzeń i przygotujemy materiał do dalszych rozważań. Wykorzystamy tu wiadomości z pozycji [1, 35]. Zacznijmy od definicji.

Definicja 3.1. *Grafem* lub *grafem nieskierowanym* nazywamy parę uporządkowaną $G = (V, E)$, gdzie

1. $V \neq \emptyset$,
2. $E \subseteq \{\{v, w\}, v, w \in V\}$.

Elementy zbioru V nazywamy *wierzchołkami* grafu, elementy zbioru E nazywamy *krawędziami* grafu. Można też używać oznaczenia $V(G)$ dla zbioru wierzchołków grafu G, $E(G)$ dla zbioru jego krawędzi.

Dla krawędzi $\{v, w\}$, wierzchołki v i w nazywamy *końcami krawędzi*.

Rozmiarem grafu $|E(G)|$ nazywamy liczbę jego krawędzi, natomiast *rzędem grafu* $|V(G)|$ liczbę wierzchołków w grafie.

Definicja 3.2. Grafem skierowanym nazywamy parę $H = (V, E)$, gdzie

1. $V \neq \emptyset$,
2. $E \subseteq \{(v, w), v, w \in V)$.

W kontekście tego rodzaju grafów możemy mówić, że wierzchołek v jest *głową* krawędzi (v, w), a w jej *ogonem*.

Znając definicję grafu, możemy mówić o relacjach, które zachodzą między wierzchołkami i krawędziami w tym grafie.

Definicja 3.3. Niech $G = (V, E)$ będzie grafem nieskierowanym.

- Wierzchołki $v, w \in V$ nazywamy *sąsiadami* lub *wierzchołkami sąsiadującymi*, jeśli istnieje krawędź łącząca te wierzchołki:

  $$\exists e \in E \ [e = \{v, w\}].$$

  Sąsiedztwo oznaczamy: $v \sim w$.

- Liczbę sąsiadów wierzchołka $v \in V$ nazywamy *stopniem wierzchołka* i oznaczamy $n(v)$.

- Dwie różne krawędzie $e, f \in E$ nazywamy *sąsiednimi* jeśli mają wspólny wierzchołek: $e \cap f \neq \emptyset$.

- Wierzchołek $v \in V$ i krawędź $e \in E$ są *incydentne* jeśli v jest jednym z końców krawędzi:

  $$\exists u \in V \ [e = \{v, u\}].$$

- Pętlą nazywamy krawędź $e \in E$ zaczynającą się i kończącą w tym samym wierzchołku: $e = \{v, v\}$.

- Mówimy, że E jest zbiorem *multikrawędzi*, jeśli istnieją w niej co najmniej dwie różne krawędzie o tych samych końcach. W tym przypadku często G nazywa się *multigrafem*.

- Wierzchołek $v \in V$ nazywamy *izolowanym*, jeśli nie istnieje krawędź incydentna z v.

Najczęściej rozważanymi grafami w praktyce są grafy proste, które można zdefiniować następująco.

Definicja 3.4. *Graf prosty* jest grafem nieskierowanym bez pętli oraz bez multikrawędzi.

W dalszej części, o ile nie napiszemy inaczej, wszystkie grafy, które będziemy rozważać będą grafami prostymi.

Definicja 3.5. Graf prosty $G = (V, E)$ jest nazywany *k-regularnym* jeśli $n(v) = k$ dla każdego $v \in V$.

Definicja 3.6. *Ścieżką* w grafie $G = (V, E)$ od wierzchołka $v \in V$ do wierzchołka $w \in V$ nazywamy ciąg wierzchołków

$$v = v_0, v_1, v_2, \ldots, v_n = w, \text{ gdzie } v_i \in V, v_i \neq v_j \text{ dla } i \neq j, \ i, j = 0, \ldots, n$$

takich, że dla każdego $i = 1, 2, \ldots, n$ istnieje krawędź $\{v_{i-1}, v_i\}$. Długością ścieżki jest liczba krawędzi ją tworzących, czyli n. Ścieżka zamknięta, czyli taka, w której $v_0 = v_n$ nazywana jest *cyklem*.

Analogiczny do ścieżki ciąg krawędzi $e_1 = \{v_0, v_1\}, e_2 = \{v_1, v_2\}, \ldots, e_n = \{v_{n-1}, v_n\}$, gdzie każda następna krawędź ma z poprzednią wspólny wierzchołek, nazywamy *drogą*.

Definicja 3.7. Niech $G = (V, E)$ będzie grafem prostym. *Odległością* $\rho(v, w)$ między wierzchołkami $v, w \in V$ nazywamy długość najkrótszej ścieżki między tymi wierzchołkami w grafie G. Jeżeli nie istnieje ścieżka między wierzchołkami v i w, to przyjmujemy $\rho(v, w) = \infty$.

Definicja 3.8. Średnicą $diam(G)$ grafu $G = (V, E)$ nazywamy maksymalną odległość między wierzchołkami w tym grafie

$$diam(G) = max_{v,w \in V} \rho(v, w).$$

Definicja 3.9. Graf G jest *spójny*, jeśli dla każdej pary różnych wierzchołków $v, w \in V$ graf G zawiera ścieżkę między nimi. W przeciwnym wypadku graf jest *niespójny*.

Definicja 3.10. *Talią* grafu prostego nazywamy długość najkrótszego cyklu w tym grafie.

Definicja 3.11. Graf spójny bez cykli nazywamy *drzewem*.

Definicja 3.12. Mówimy, że graf prosty $G' = (V', E')$ jest *podgrafem* grafu $G = (V, E)$, jeśli spełnione są warunki:

1. $V' \subseteq V$,
2. $E' \subseteq E$.

Definicja 3.13. Niech G będzie grafem prostym. Niech $H_1, H_2, \ldots, H_k$ będą pografami spójnymi G, takimi, że ich zbiory krawędzi oraz zbiory wierzchołków są parami rozłączne oraz pokrywają cały graf G, tzn

$$V(G) = V(H_1) \cup V(H_2) \cup \ldots \cup V(H_k);$$

$$E(G) = E(H_1) \cup E(H_2) \cup \ldots \cup E(H_k);$$

$$V(H_i) \cap V(h_j) = \emptyset = E(H_i) \cap E(H_j), \quad \text{dla } i \neq j.$$

Wtedy każdy podgraf $H_i, i = 1, \ldots, k$ nazywamy *składową spójności* grafu G.

Definicja 3.14. Graf prosty $G = (V, E)$ nazywamy *dwudzielnym* (ang. *bipartite*), jeśli zbiór wierzchołków tego grafu można podzielić na dwa podzbiory $X, Y \subseteq V$ takie, że

1. $V = X \cup Y$;
2. $X \cap Y = \emptyset$;
3. $\forall e \in E \;\; \exists v \in X \; \exists w \in Y \;\; [e = \{v, w\}]$.

Innymi słowy każda krawędź w grafie G ma jeden koniec w X, a drugi w Y.

Jednym z oczywistych intuicyjnie faktów dwudzielności grafu jest istnienie w nim jedynie parzystych cykli. Następujące twierdzenie precyzuje tę własność.

Twierdzenie 3.1. *Dla dowolnego grafu G następujące fakty są równoważne:*

1. *G jest grafem dwudzielnym.*
2. *Każdy cykl w G ma parzystą długość.*

Na koniec tej części przypominamy definicje przekształceń na grafach.

Definicja 3.15. Niech $G = (V, E)$ i $G' = (V', E')$ będą grafami prostymi. Homomorfizmem $f : G \to G'$ nazywamy odwzorowanie zbioru wierzchołków V w zbiór V' takie, że

$$\forall v, w \in V \;\; v \sim w \Rightarrow f(v) \sim f(w).$$

Jeżeli przekształcenie f jest bijekcją a odwzorowanie odwrotne również zachowuje relację incydencji, wówczas f nazywamy *izomorfizmem*. Izomorfizm grafu G w ten sam graf nazywany jest *automorfizmem*.

Definicja 3.16. Graf $G = (V, E)$ nazywamy wierzchołkowo tranzytywnym, jeśli dla dowolnych dwóch wierzchołków $v, w \in V$ istnieje automorfizm $f : V \to V$ taki, że

$$f(v) = w.$$

3.2 Elementy teorii grafów ekstremalnych

Teoria grafów ekstremalnych ([5, 6]) zajmuje się zwykle problemami grafów o maksymalnej lub minimalnej liczbie krawędzi lub wierzchołków, posiadających pewną własność. Na przykład: mamy graf „zabroniony" G oraz ustaloną liczbę wierzchołków n i pytamy jaka jest maksymalna liczba krawędzi grafu

posiadającego n wierzchołków, który nie posiada podgrafu G? Lub odwrotnie: jaka liczba krawędzi gwarantuje w grafie o n wierzchołkach zawieranie G? Ile krawędzi zapewni ścieżkę długości co najmniej k, ile krawędzi cykl długości co najmniej k lub cykl długości co najwyżej k? Badane są różne zbiory (czyli rodziny) grafów i ich ekstrema. Przyjrzyjmy się dokładniej jednej z takich własności.

Niech $\mathcal{F}$ oznacza rodzinę grafów. Przez

$$ex(n, \mathcal{F})$$

oznaczamy maksymalną ilość krawędzi w rodzinie grafów o n wierzchołkach, nie zawierających podgrafów z rodziny $\mathcal{F}$. Będziemy teraz rozpatrywać rodziny grafów nie zawierających coraz dłuższych cykli.

3.2.1 Grafy o dużej talii

Niech C_n oznacza cykl w grafie długości $n \geqslant 3$. Światowej sławy matematyk P. Erdos postawił w pracy [15], które zostało udowodnione później przez jego następców [7, 16]. W twierdzeniu tym dokonano oszacowania i pokazano że każdy graf rzędu n posiadający więcej niż $90kn^{1+\frac{1}{k}}$ krawędzi musi posiadać cykl długości $2k$. Stąd

$$ex(n, \{C_3, C_4, \ldots, C_{2k}\}) \leqslant 90kn^{1+\frac{1}{k}}.$$

Dolna granica tej liczby krawędzi szacowana jest na $\Theta(n^{1+\frac{1}{n-1}})$.

Niech $G = G(V, E)$ będzie grafem prostym. Przypomnijmy, że talia grafu ($g = g(G)$) oznacza długość najkrótszego cyklu w grafie. Za pracą [4] wprowadzamy:

Definicja 3.17. Niech $\{G_{i,i\geqslant 1}\}$,, będzie rodziną grafów, taką że każde G_i jest grafem r-regularnym o rosnącym rzędzie v_i oraz rosnącej talii g_i. Mówimy, że $\{G_i\}$ jest *rodziną grafów o dużej talii* jeśli

$$\exists \gamma \in \mathbb{R} \quad \left[g_i \geqslant \gamma \log_{r-1}(v_i) \right],$$

dla pewnej stałej γ.

Zostało dowiedzione, że $\gamma \leqslant 2$ i matematycy szukali różnych rodzin grafów jak najbliższych tej granicy.

3.3 Grafy algebraiczne

Znalezienie rodzin grafów o dużej talii oraz jak największej wartości γ z definicji 3.17 przyświecało powstaniu rodziny grafów, którą wykorzystamy w naszych algorytmach kryptograficznych. Rodzina ta została wprowadzona metodą niestandardową dla teorii grafów: poprzez równania algebraiczne. Zanim zaprezentujemy rzeczoną rodzinę, przyjrzyjmy się ogólnej idei konstruowania grafów algebraicznych ([3, 19]).

Niech $G(V, E)$ będzie grafem dwudzielnym, w którym zbiór wierzchołków dzielimy na dwa podzbiory, nazywając je tradycyjnie liniami L oraz punktami P. Punkt $x \in P$ może być połączony krawędzią jedynie z linią $y \in L$ i odwrotnie — linia jedynie z punktem. Wybierzmy teraz pierścień $\mathbb{K}$ i ustalmy $n \in \mathbb{N}$. Z każdym wierzchołkiem grafu kojarzymy teraz krotkę (lub wektor jeśli $\mathbb{K}$ będzie ciałem) n współczynników nad $\mathbb{K}$. Dodatkowo wprowadzamy rozróżnienie nawiasów między punktami, dla których użyjemy nawiasów zwykłych, a liniami, dla których użyjemy nawiasów kwadratowych. Mamy więc:

$$\forall x \in P, \quad x = (x_1, x_2, \ldots, x_n), \quad x_i \in \mathbb{K}, 1 \leqslant i \leqslant n,$$

$$\forall y \in L, \quad y = [y_1, y_2, \ldots, y_n], \quad y_j \in \mathbb{K}, 1 \leqslant j \leqslant n,$$

oraz z dwudzielności:

$$V = P \cup L, \quad P \cap L = \emptyset.$$

Brakuje jeszcze struktury incydencji grafu, czyli sposobu łączenia elementów ze zbiorów P i L krawędziami.

W naszych dalszych rozważaniach strukturę incydencji grafu tworzyć będzie $(n-1)$ równań algebraicznych, wiążących współczynniki x_i oraz y_j, $1 \leqslant i, j \leqslant n$ ze sobą. Używać będziemy równań, których ogólną postać można wyrazić następująco:

$$x_i \pm y_i = \epsilon_i \; x_1 \; y_{k(i)} + (1 - \epsilon_i) \; y_1 \; x_{k(i)}, \tag{3.1}$$

gdzie $\epsilon_i \in \{0, 1\}$ dla $2 \leqslant i \leqslant n$, natomiast funkcja $k : \mathbb{N} \to \mathbb{N}$ spełnia zawsze zależność $k(i) < i$. Wartości ϵ_i można zebrać w jedną krotkę

$$\xi = (\epsilon_1, \epsilon_2, \ldots, \epsilon_{n-1})$$

i finalną konfigurację struktury incydencji stanowi para (k, ξ).

Dla konkretnego grafu mamy więc $(n-1)$ równań między współczynnikami krotek x i y. Stąd dla konkretnego wierzchołka $x \in P$ mamy przepis ogólny na sposób obliczenia każdego jego sąsiada. Wystarczy rozwiązać ten układ

postaci (przyjmijmy + po lewej stronie równań):

$$\begin{aligned} x_2 + y_2 &= x_1\ y_1 \\ x_3 + y_3 &= \epsilon_3\ x_1\ y_{k(3)} + (1-\epsilon_3)\ y_1\ x_{k(3)} \\ x_4 + y_4 &= \epsilon_4\ x_1\ y_{k(4)} + (1-\epsilon_4)\ y_1\ x_{k(4)} \\ &\vdots \\ x_n + y_n &= \epsilon_n\ x_1\ y_{k(n)} + (1-\epsilon_n)\ y_1\ x_{k(n)}, \end{aligned}$$

rozwiązując równania po kolei, traktując $y_2, y_3, \ldots, y_n$ jako niewiadome, natomiast zaczynając od konkretnej wartości y_1. Różnych wartości y_1 jest tyle, ile jest elementów w pierścieniu (lub ciele) $\mathbb{K}$. Analogicznie, mając krotkę (lub wektor) y, wystarczy powyższe równania rozwiązać ze względu na krotkę x, wybierając konkretny współczynnik x_1.

Jeśli teraz przez r oznaczymy ilość elementów w pierścieniu (lub ciele) $\mathbb{K}$, mamy natychmiastowy wniosek, że nasze grafy są grafami r-regularnymi. Ostatecznie więc jednoznacznie strukturę konkretnego grafu określać będzie trójka $(\mathbb{K}, k, \xi)$.

Przykład 3.1. *Jako prosty przykład grafu należącego do wprowadzonej przez nas klasy niech posłuży rodzina Grafów Wengera $W_n(\mathbb{F})$ nad ciałem skończonym $\mathbb{F}$ ([50]). Przy wszystkich oznaczeniach wprowadzonych wcześniej w tej sekcji równania 3.1 przybierają tutaj postać*

$$x_i + y_i = x_1\ y_{i-1},$$

dla $2 \leqslant i \leqslant n$. Graf Wengera ma małą talię $g(W_n(\mathbb{F})) = 8$, w związku z tym nie nadaje się do naszych zastosowań.

Rozdział 4

Kryptografia symetryczna z wykorzystaniem grafów o dużej talii. Rodziny grafów $D_n(\mathbb{K})$ oraz $A_n(\mathbb{K})$

W tym rozdziale zaprezentujemy schemat kryptografii symetrycznej z wykorzystaniem grafów o dużej talii. Uzasadnimy potrzebę posiadania przez grafy dużej talii. Następnie wprowadzimy dwie rodziny grafów, które będą podstawą wszystkich stosowanych przez nas w praktyce algorytmów.

4.1 Algorytm symetryczny wykorzystujący grafy

Wróćmy do podstawowego problemu kryptografii. Mamy pewien tekst otwarty M, będący ciągiem bitów, który chcemy zaszyfrować w tajny szyfrogram E.

Wybierzmy zatem jakieś ciało skończone $\mathbb{F}_q$ (lub pierścień przemienny $\mathbb{Z}_q$), który potraktujemy jako alfabet dla tekstu otwartego oraz szyfrogramu. Tekst otwarty, a także zaszyfrowany będzie krotką nad tym alfabetem o n składowych:

$$M = (x_1, x_2, \ldots, x_n), \quad x_i \in \mathbb{F}_q, 1 \leqslant i \leqslant n,$$

$$E = (e_1, e_2, \ldots, e_n), \quad e_i \in \mathbb{F}_q, 1 \leqslant i \leqslant n.$$

Bez straty ogólności rozważań możemy założyć, że hasło K jest również krotką zapisaną nad tym samym alfabetem:

$$K = (a_1, a_2, \ldots, a_k), \quad a_i \in \mathbb{F}_q, 1 \leqslant i \leqslant k.$$

Algorytm szyfrowania intuicyjnie polega na przejściu po grafie G ścieżką wyznaczoną przez hasło K. Dodatkowo przyjmujemy, że tekst jawny jest skojarzony z wierzchołkiem będącym punktem.

Jeden podstawowy krok algorytmu szyfrowania polega na wybraniu sąsiada bieżącego wierzchołka na podstawie elementu hasła $a_i, i = 1, \ldots, k$. Krok ten składa się z dwóch etapów (na przykładzie wierzchołka będącego punktem $x \in P$):

1. wyznaczenia pierwszej współrzędnej y_1 sąsiada wierzchołka x na podstawie współrzędnych x oraz a_1,

2. wyliczeniu pozostałych współrzędnych $y_2, \ldots, y_n$ z równań opisujących strukturę incydencji grafu.

W ten sposób otrzymujemy nowy wierzchołek $y \in L$, będący sąsiadem x.

W kolejnym kroku tą samą metodą, ale rozwiązując równania ze względu na $(x_1, x_2, \ldots, x_n)$, wyznaczymy przy pomocy y oraz a_{i+1} nowego sąsiada y należącego do zbioru P. Ostatni tak wyznaczony wierzchołek będziemy traktować jako szyfrogram E. W zależności od parzystości ilości elementów hasła, może to być punkt lub linia.

Procedurę szyfrowania możemy schematycznie opisać jako:

$$M = x^0 \xrightarrow{a_1} y^1 \xrightarrow{a_2} x^2 \xrightarrow{a_3} y^3 \rightarrow \quad \ldots \quad \xrightarrow{a_k} z^k = E,$$

gdzie $z^k \in P$ dla k parzystych lub $z^k \in L$ dla k nieparzystych. Ponadto $x^s \in P,$ dla $s = 0, 2, 4, \ldots; y^t \in L,$ dla $t = 1, 3, 5, \ldots$, a górny indeks oznacza numer wierzchołka grafu w procedurze szyfrowania. Procedura deszyfrowania polegać będzie na przejściu po grafie tą samą ścieżką, ale w odwrotnej kolejności: od wierzchołka z^k do wierzchołka x^0.

Nie wszystkie grafy dwudzielne wprowadzone równaniami algebraicznymi (3.1) są dobrymi kandydatami do przedstawionej powyżej procedury szyfrowania. W kolejnych uwagach uzasadnimy wybór grafów ekstremalnych o możliwie dużej talii. Przedstawimy również dodatkowe obostrzenia na dwa kolejne kroki szyfrowania.

Uwaga 1: W procedurze szyfrowania należy wybierać wierzchołki grafu w ten sposób, aby wierzchołki o numerach różniących się o 2 były różne od siebie. W przeciwnym wypadku występowałaby za każdym razem redukcja hasła o dwa symbole, ponieważ dwa kolejne kroki algorytmu nie zmieniałyby wierzchołka grafu.

Uwaga 2: W grafie nie powinny występować cykle małej długości, czyli graf powinien mieć jak największą talię. Występowanie krótkich cykli wiąże się z następującymi problemami:

1. przy cyklu długości m możliwa jest redukcja m kroków algorytmu przez problem analogiczny do problemu z Uwagi 1;

2. pomimo zaszyfrowania tekstu jawnego ścieżką wyznaczoną przez klucz $K = (a_1, \ldots, a_k)$, możliwe jest znalezienie innej ścieżki K' (być może nawet krótszej) między wierzchołkami M i E w grafie. Połączona ścieżka wyznaczona przez K i K' tworzyłaby wtedy cykl.

Wniosek 4.1. *Jeśli w grafie dwudzielnym ze strukturą incydencji zadaną przez równania algebraiczne postaci* (3.1) *nie występują cykle o długości s, to opisana powyżej procedura szyfrowania kluczami długości mniejszej niż $s/2$ jest „bezpieczna".*

Mamy tu na myśli „bezpieczeństwo" w sensie braku możliwości znalezienia innej, krótszej niż zadana hasłem, ścieżki w grafie między tekstem jawnym a szyfrogramem. Co się z tym wiąże — najszybszym atakiem kryptologicznym, wykorzystującym wiedzę o strukturze grafu, byłby *atak brutalny*, polegający na pełnej eksploracji grafu „w szerz". Oczywiście mamy świadomość bogactwa pomysłów i możliwości dzisiejszej analizy kryptologicznej. Dopuszczamy więc możliwość istnienia innego sposobu łamania naszego szyfru.

4.1.1 Kolorowanie wierzchołków grafu. Formalny zapis algorytmu szyfrowania

Kolorowanie grafu jest bardzo obszernym zagadnieniem teorii grafów. W naszej pracy posłuży ono głównie sformalizowaniu algorytmu kryptograficznego, więc nie będziemy tu przytaczać klasycznych twierdzeń i definicji związanych z samym kolorowaniem wierzchołków grafu.

Poprzez kolorowanie rozumie się przypisanie wybranym elementom grafu (najczęściej wierzchołkom, rzadziej krawędziom) pewnych liczb. W naszym przypadku przyjmujemy, że liczby te są nad alfabetem $\mathbb{F}_q$ (lub $\mathbb{Z}_q$, gdy użyjemy pierścienia modulo). Kolorem każdego wierzchołka określamy wartość pierwszej składowej wektora skojarzonego z tym wierzchołkiem. Wprowadzamy więc funkcję kolorowania wierzchołków, przy wszystkich dotychczasowych oznaczeniach grafów dwudzielnych algebraicznych, jako:

$$c : \mathbb{F}_q^n \to \mathbb{F}_q,$$
$$\forall_{v \in V, v=(v_1,v_2,\ldots,v_n)} \quad c(v) = v_1.$$

Jako bezpośredni wniosek z definicji funkcji c oraz faktu, że układ równań (3.1) można rozwiązać na q sposobów, otrzymujemy następujące

Twierdzenie 4.2. *Dla dowolnego wierzchołka $v \in V$ zbiór jego sąsiadów tworzy zbiór wierzchołków o różnych kolorach.*

Ponadto biorąc pod uwagę, że mamy graf q-regularny oraz q możliwych kolorów, możemy zdefiniować operator wyboru sąsiada dla wybranego wierzchołka grafu w sposób następujący:

$$\begin{gathered} N : \mathbb{F}_q^{n+1} \to \mathbb{F}_q^n, \\ \forall_{v \in V, v=(v_1,v_2,\ldots,v_n)}, \forall_{a \in \mathbb{F}_q} \quad [N_a(v) = (v_1 + a, w_2, w_3, \ldots, w_n)], \end{gathered} \tag{4.1}$$

gdzie parametry $w_2, w_3, \ldots, w_n$ wylicza się z równań struktury incydencji grafu, inaczej dla $x \in P$ a inaczej dla $y \in L$. Parametr zmiany koloru wierzchołka a możemy utożsamiać z pojedynczym elementem klucza szyfrowania.

Jeżeli więc klucz jest wektorem $K = (a_1, \ldots, a_k) \in \mathbb{F}_q^k$, to kolejne kroki algorytmu szyfrowania możemy zapisać następująco

$$\begin{aligned} N_{a_1}(x^0) &= y^1 \\ N_{a_2}(y^1) &= x^2 \\ &\vdots \\ N_{a_k}(x^{k-1}) &= y^k, \text{ jeśli } k \text{ było nieparzyste; lub} \\ N_{a_k}(y^{k-1}) &= x^k, \text{ jeśli } k \text{ było parzyste.} \end{aligned}$$

Zbierając to w jeden zapis mamy:

$$N_{a_k}(N_{a_{k-1}}(\ldots(N_{a_1}(M = x^0))\ldots) = v^k = E,$$

Jeśli teraz oznaczymy jako N_K operator powstały ze złożenia kroków algorytmu (dla wygody kolejność działania operacji będzie zaczynać się od lewej strony, tak jak kolejność kroków algorytmu):

$$N_K = N_{a_1} \circ N_{a_2} \circ \ldots \circ N_{a_k},$$

to szyfrowanie możemy przedstawić klasycznie jako:

$$N_K(M) = E.$$

Aby móc deszyfrować tak zakodowane wiadomości, wystarczy zauważyć, że:

$$(N_a)^{-1} = N_{-a},$$

gdzie $-a$ jest symbolem przeciwnym do a w ciele skończonym $\mathbb{F}_q$ (lub pierścieniu modulo $\mathbb{Z}_q$). Stąd całą procedurę deszyfrowania możemy oznaczyć jako:

$$(N_K)^{-1} = N_{-a_k} \circ N_{-a_{k-1}} \circ \ldots \circ N_{-a_1}.$$

Algorytm 4.1 Podstawowa procedura szyfrowania (rodzina grafów algebraicznych dwudzielnych)

Wejście: $K = (a_1, \ldots, a_k), a_i \in \mathbb{K}$ — klucz,
$x \in P$ — wierzchołek grafu, tekst jawny
Wyjście: $u = N_K(x)$ — wierzchołek grafu, szyfrogram

```
1: for j = 1, 2, ..., k
2:     if j ≡ 0 (mod 2)
3:         x := N_{a_j}(y)
4:     else
5:         y := N_{a_j}(x)
6: if k ≡ 0 (mod 2)
7:     u := x
8: else
9:     u := y
```

Dodatkowy krok w procedurze deszyfrowania symetrycznego wymaga sprawdzenia parzystości k, aby wiedzieć czy zastosować złożony operator $(N_K)^{-1}$ dla punktu czy dla linii i, co się z tym wiąże, rozwiązywać równania w pierwszym kroku ze względu na wektor niewiadomych x czy y.

Algorytm 4.1 przedstawia procedurę szyfrowania zapisaną w pseudokodzie dla dowolnego grafu algebraicznego dwudzielnego regularnego z operatorem wyboru sąsiada $N_a(x)$ (gdy wyliczamy współrzędne wierzchołka będącego linią) lub $N_a(y)$ (gdy wyliczamy współrzędne punktu). Deszyfrowanie opisane jest w algortymie 4.2. Obie procedury używają w kodzie operatora N_a jako pojedynczej instrukcji, a w praktyce jest to algorytm, którego postać zależy od struktury incydencji grafu.

4.2 Rodzina grafów $D_n(\mathbb{K})$

Rodzinę grafów $D_n(\mathbb{K})$ wprowadzili profesorowie Lazebnik i Ustimenko w pracach [28, 29]. Pierwsza definicja zakładała budowanie tej rodziny grafów nad ciałem skończonym $\mathbb{F}_q$, a oznaczenie rodziny parametryzowano zmiennymi n i q. Na dzień dzisiejszy wiemy, że można tę rodzinę grafów budować również nad ogólniejszymi pierścieniami przemiennymi, choć niektóre własności grafów ulegają wtedy zmianie.

Używając tych samych oznaczeń, niech $\mathbb{K}$ będzie pierścieniem przemiennym z jedynką, $D(V, E)$ grafem dwudzielnym, gdzie $V = P \cup L$ jest podziałem na punkty i linie. Niech n oznacza liczbę składowych wektora (krotki) skojarzonego z każdym wierzchołkiem. Dla zachowania spójności z oryginal-

Algorytm 4.2 Podstawowa procedura deszyfrowania (rodzina grafów algebraicznych dwudzielnych)

Wejście: $K = (a_1, \ldots, a_k), a_i \in \mathbb{K}$ — klucz,
$u \in P$ — wierzchołek grafu, szyfrogram
Wyjście: $x = N_K^{-1}(u)$ — wierzchołek grafu, tekst jawny

```
1: if k ≡ 0 (mod 2)
2:     x := u
3: else
4:     y := u
5: for j = k, k − 1 . . . , 2, 1
6:     if j ≡ 0 (mod 2)
7:         y := N_{−a_j}(x)
8:     else
9:         x := N_{−a_j}(y)
```

ną konstrukcją definiujemy następujące numerowanie składowych dla każdego $x \in P, y \in L$:

$$\begin{aligned}(x) &= (x_1, x_{11}, x_{12}, x_{21}, x_{22}, x'_{22}, x_{23}, x_{32}, x_{33}, x'_{33}, \ldots,\\ &\quad x_{ii}, x'_{ii}, x_{i,i+1}, x_{i+1,i}, \ldots)\\ [y] &= [y_1, y_{11}, y_{12}, y_{21}, y_{22}, y'_{22}, y_{23}, y_{32}, y_{33}, y'_{33}, \ldots,\\ &\quad y_{ii}, y'_{ii}, y_{i,i+1}, y_{i+1,i}, \ldots].\end{aligned}$$

Rodzina $D_n(\mathbb{K}), n \in \mathbb{N}$ jest nieskończonym zbiorem grafów, jednak dla określonego n mamy konkretny graf, z dokładnie n składowymi w każdym punkcie x oraz w każdej linii y. Składowe te spełniają następujące równania algebraiczne ($n-1$ pierwszych równań, gdy składowych jest n):

$$\begin{aligned} y_{11} - x_{11} &= y_1\ x_1\\ y_{12} - x_{12} &= x_1\ y_{11}\\ y_{21} - x_{21} &= y_1\ x_{11}\\ y_{ii} - x_{ii} &= y_1\ x_{i-1,i}\\ y'_{ii} - x'_{ii} &= x_1\ y_{i,i-1}\\ y_{i,i+1} - x_{i,i+1} &= x_1\ y_{ii}\\ y_{i+1,i} - x_{i+1,i} &= y_1\ x'_{ii}, \end{aligned} \tag{4.2}$$

gdzie ostatnie cztery równania są zdefiniowane dla $i \geqslant 2$. Dla lepszej czytelności opuszczono przecinki między podwójnymi indeksami tam, gdzie nie prowadzi to do dwuznaczności.

Pewną prawidłowość wśród indeksów w powyższych równaniach można zaobserwować, jeśli przyjmiemy $x_1 = x_{01}$, $y_1 = y_{10}$: sumy „pierwszych" i „drugich" indeksów po obu stronach równości są te same. Symbolu „prim" użyto z uwagi na to, że wartości indeksów i, i można uzyskać na dwa sposoby: $x_{01}\ y_{i,i-1}$ oraz $y_{10}\ x_{i-1,i}$.

4.2.1 Algorytm obliczania sąsiada

Podwójne indeksy w równaniach (4.2) są wygodne w zapisie matematycznym. Jednak do zaprogramowania algorytmu należy przenumerować te indeksy do postaci krotek jednowymiarowych. Równania (4.2) należy naturalnie przekształcić przenosząc niewiadome na jedną stronę i rozwiązywać poczynając od zmiennej o najniższym indeksie. Algorytmy 4.3 oraz 4.4 przedstawiają sposób obliczania sąsiada o kolorze a dla ustalonego punktu lub linii.

Algorytm 4.3 Operator $N_a(x), x \in P$ (rodzina grafów $D_n(\mathbb{K})$)

Wejście: $a \in \mathbb{K}$,
$x \in P$ — wierzchołek grafu,
Wyjście: $y = N_a(x)$

```
 1: y_0 := x_0 + a
 2: y_1 := x_1 + x_0 · y_0
 3: y_2 := x_2 + x_0 · y_1
 4: r := 2                                          //r ≡ (i − 1) mod 4
 5: for i = 3, 4, ..., n − 1
 6:     if r < 2                                    //i ≡ 1 ∨ i ≡ 2 (mod 4)
 7:         y_i := x_i + x_0 · y_{i−2}
 8:     else                                        //i ≡ 0 ∨ i ≡ 3 (mod 4)
 9:         y_i := x_i + y_0 · x_{i−2}
10:     if r = 3
11:         r := 0
12:     else
13:         r := r + 1
```

Aby uzyskać symetryczny algorytm szyfrujący oparty na rodzinie $D_n(\mathbb{K})$ należy wykorzystać algortym ogólny 4.1 wstawiając w miejsce operatora N_a w krokach nieparzystych algorytm 4.4, a w parzystych algorytm 4.3. Analogicznie tworzymy algorytm deszyfrowania, używając jako bazy algorytmu 4.2.

Algorytm 4.4 Operator $N_a(y), y \in L$ (rodzina grafów $D_n(\mathbb{K})$)

Wejście: $a \in \mathbb{K}$,
$y \in L$ — wierzchołek grafu,
Wyjście: $x = N_a(y)$

1: $x_0 := y_0 + a$
2: $x_1 := y_1 - x_0 \cdot y_0$
3: $x_2 := y_2 - x_0 \cdot y_1$
4: $r := 2$ //$r \equiv (i-1) \bmod 4$
5: **for** $i = 3, 4, \ldots, n-1$
6: **if** $r < 2$ //$i \equiv 1 \vee i \equiv 2 \pmod 4$
7: $x_i := y_i - x_0 \cdot y_{i-2}$
8: **else** //$i \equiv 0 \vee i \equiv 3 \pmod 4$
9: $x_i := y_i - y_0 \cdot x_{i-2}$
10: **if** $r = 3$
11: $r := 0$
12: **else**
13: $r := r + 1$

4.2.2 Własności grafów z rodziny $D_n(\mathbb{F}_q)$

W tej sekcji przedstawimy własności rodziny grafów D_n budowanej nad ciałem skończonym $\mathbb{F}_q$. Większość tych własności zostało odkrytych i udowodnionych metodami czysto matematycznymi przez profesora Ustimenko oraz jego współpracowników z różnych uczelni w latach 1995-2010 [29, 30, 45, 47, 51].

Talia grafu

Zacznijmy od rzeczy podstawowej w naszych rozważaniach, czyli talii grafu.

Twierdzenie 4.3. *Niech q będzie pewną potęgą liczby pierwszej i niech $k \geqslant 2$. Wtedy zachodzą następujące fakty:*

1. *$D_n(\mathbb{F}_q)$ jest q-regularnym grafem dwudzielnym rzędu $|V| = 2q^n$;*
2. *dla n nieparzystych mamy $g(D_n(\mathbb{F}_q)) \geqslant n + 5$;*
3. *dla n nieparzystych oraz $q \equiv 1 (mod\ \frac{n+5}{2})$ zachodzi $g(D_n(\mathbb{F}_q)) = n + 5$.*

W kolejnych latach rozszerzono twierdzenie i uogólniono wynik z punktu 2 na n dowolne. **Uwaga:** Przełóżmy teraz powyższy wynik na interpretację kryptograficzną pomijając stałą 5. Dla wiadomości otwartej o długości n otrzymamy wówczas możliwość używania do szyfrowania haseł o maksymalnej długości $n/2$ nad tym samym alfabetem co tekst otwarty. Hasła tej długości gwarantują,

że nie będzie istniała w grafie krótsza ścieżka między wierzchołkami reprezentującymi tekst otwarty i szyfrogram, niż ścieżka użyta do szyfrowania.

Spójność

Przyjrzyjmy się teraz spójności naszych grafów.

Twierdzenie 4.4. *Niech $2 \leqslant n \leqslant 5$. Wtedy graf $D_n(\mathbb{F}_q)$ jest spójny.*

W analizie spójności grafów o większym rozmiarze ($n > 5$) przydatne będzie wprowadzenie dodatkowego oznaczenia (a_r). Poniższy pomysł wynika z algebraicznej interpretacji równań struktury incydencji grafu i przedstawia tak naprawdę formę kwadratową.

Niech $n \geqslant 6$, $t = \lfloor (n+2)/4 \rfloor$ i niech

$$u = (u_1, u_{11}, \ldots, u_{tt}, u'_{tt}, u_{t,t+1}, u_{t+1,t}, \ldots)$$

będzie wierzchołkiem grafu $D_n(\mathbb{F}_q)$ (u jest linią lub punktem). Dla każdego $r, 2 \leqslant r \leqslant t$, niech będzie dana funkcja $a_r : \mathbb{F}_q^n \to \mathbb{F}_q$ określona wzorem

$$a_r = a_r(u) = \sum_{i=0}^{r} (u_{ii}\ u'_{r-i,r-i} - u_{i,i+1}\ u_{r-i,r-i-1}).$$

Natomiast przez $a(u)$ oznaczmy następujący wektor funkcji

$$a(u) = (a_2, a_3, \ldots, a_t).$$

Twierdzenie 4.5. *Niech q będzie liczbą nieparzystą, $n \geqslant 6$. Wtedy następujące fakty są równoważne*

- *wierzchołki u oraz v należą do tej samej składowej spójnej grafu $D_n(\mathbb{F}_q)$;*
- *$a(u) = a(v)$.*

Oczywiście graf o kilku spójnych składowych jako całość jest niespójny. Ciekawy fakt związany ze składowymi spójnymi grafu przedstawia następujące

Twierdzenie 4.6. *Niech q będzie liczbą nieparzystą, $n \geqslant 6$. Dla każdych $t-1$ elementów x_i ciała skończonego $\mathbb{F}_q$, $2 \leqslant i \leqslant t = \lfloor (n+2)/4 \rfloor$, istnieje wierzchołek v grafu $D_n(\mathbb{F}_q)$, dla którego*

$$a(v) = (x_2, x_3, \ldots, x_t).$$

Uwaga: Powyższe własności zachodzą dla wszystkich ciał skończonych o nieparzystej liczbie elementów, czyli $q = p^s$, gdzie p jest nieparzystą liczbą pierwszą. W przypadku $p = 2$ mamy inne własności.

Stopień przekształcenia

Przejdźmy teraz do interpretacji algebraicznej przekształcenia wykorzystującego grafy $D_n(\mathbb{F}_q)$. Patrząc na definicję operatora N (def. 4.1) naturalnie rodzi się pytanie o stopień skomplikowania przekształcenia, będącego pojedynczym krokiem algorytmu (N_a), a także wielokrotnie złożonego przekształcenia (N_K), odpowiadającego całej procedurze szyfrowania. Gdyby się okazało, że dla ustalonego elementu hasła a mamy do czynienia z przekształceniem liniowym przestrzeni wektorowej $\mathbb{F}_q^n \to \mathbb{F}_q^n$, symetryczny algorytm szyfrowania byłby łatwy do złamania metodami algebry liniowej, bez wnikania w sposób jego konstrukcji i wykorzystane grafy. Poniższe twierdzenie jest głównym wynikiem teoretycznym pracy [51]

Twierdzenie 4.7. *Niech $n > 4$. Przy naszych oznaczeniach odwzorowanie N_K przestrzeni $\mathbb{F}_q^n$ na siebie, wykorzystujące strukturę incydencji grafu $D_n(\mathbb{F}_q)$ jest odwzorowaniem bijektywnym stopnia 3, niezależnie od wyboru oraz długości K (za wyjątkiem przekształcenia zdegenerowanego do identyczności).*

Dowód twierdzenia polega na wykorzystaniu zasady indukcji matematycznej ze względu na długość klucza k i analizie stopni każdego z przekształceń potrzebnych do obliczenia pojedynczej składowej wierzchołka. Dla wierzchołka szyfrogramu $u = (u_1, u_2, \ldots, u_n), u \in V$, wzory na współrzędne u_i o indeksach wyższych niż 2, były wyrażeniami wielomianowymi stopnia 2 lub 3 współrzędnych wierzchołka wejściowego $x = (x_1, x_2, \ldots, x_n), x \in P$.

4.3 Rodzina grafów $A_n(\mathbb{K})$

Rodzinę grafów $A_n(\mathbb{K})$ oryginalnie otrzymano w roku 2007. W pierwszej pracy [47] wykorzystywana była jedynie jako obiekt matematyczny, służący dowodzeniu własności grafów z rodziny $D_n(\mathbb{K})$. W następnych latach rodzinę $A_n(\mathbb{K})$ zbadano pod kątem własności analogicznych do znanej już rodziny $D_n(\mathbb{K})$ i okazało się, że jej własności pozwalają wykorzystywać te grafy w algorytmach kryptograficznych.

Strukturę rodziny $A_n(\mathbb{K})$ otrzymano poprzez usunięcie z układu równań (4.2) tych równań, w których występowały współrzędne z notacją „prim", a także usunięcie tych składowych z definicji współrzędnych. W związku z tym mamy teraz układ równań powtarzających się cyklicznie co dwa oraz wektory $x \in P, y \in L$, których indeksy współrzędnych przyjmują wartości:

$$x = (x_{01}, x_{11}, x_{12}, x_{22}, \ldots, x_{ii}, x_{i,i+1}, \ldots),$$

$$y = [y_{10}, y_{11}, y_{12}, y_{22}, \ldots, y_{ii}, y_{i,i+1}, \ldots].$$

Strukturę incydencji tych grafów opisują następujące równania:

$$\begin{aligned} y_{ii} - x_{ii} &= y_{10}\ x_{i-1,i} \\ y_{i,i+1} - x_{i,i+1} &= x_{01}\ y_{i,i} \\ & \quad i = 1, 2, \ldots \end{aligned} \tag{4.3}$$

Podobnie jak poprzednio, przyjmiemy oznaczenia $x_1 = x_{01},\ y_1 = y_{10}$.

4.3.1 Podobieństwa i różnice między rodzinami $A_n(\mathbb{K})$ oraz $D_n(\mathbb{K})$

Przyjrzyjmy się rodzinie grafów $A_n(\mathbb{K})$ i jej własnościom, porównując je z rodziną $D_n(\mathbb{K})$. Zacznijmy od podobieństw.

Talia

Twierdzenie 4.8. *[30] Niech $\mathbb{F}_q$ będzie ciałem skończonym. Wówczas talia grafu $A_n(\mathbb{F}_q)$ rośnie ze wzrostem n, czyli*

$$\lim_{n\to\infty} g(A_n(\mathbb{F}_q)) = \infty.$$

Nie zostało jednak do tej pory dowiedzione, czy grafy $A_n(\mathbb{F}_q)$ są grafami o dużej talii w myśl definicji 3.17. Problem pozostaje otwarty.

Spójność

Zacznijmy od przypadku $q = 2$. Wtedy zarówno $A_n(\mathbb{F}_q)$, jak i $D_n(\mathbb{F}_q)$ są rodzinami grafów 2-regularnych. Jak łatwo zauważyć każdy taki graf jest albo cyklem, albo zbiorem cykli i nie istnieją w takich grafach inne spójne składowe. Udowodnijmy twierdzenie

Twierdzenie 4.9. *Niech będzie dane ciało $\mathbb{F}_2$. Każdy graf rodziny $A_n(\mathbb{F}_2)$ dla $n > 2$ nie jest spójny.*

Dowód. Rozpatrzymy graf $A_3(\mathbb{F}_2)$. Wierzchołki tego grafu tworzą dwa cykle:

$$(0,0,0)-[1,0,0]-(1,1,0)-[0,1,1]-(0,1,1)-[1,1,1]-(1,0,0)-[0,0,0]-(0,0,0),$$

$$(0,0,1)-[0,0,1]-(1,0,1)-[1,1,0]-(0,1,0)-[0,1,0]-(1,1,1)-[1,0,1]-(0,0,1).$$

Mamy więc dwie składowe spójne, czyli graf $A_3(\mathbb{F}_2)$ nie jest spójny.

Weźmy teraz $n > 3$. Załóżmy, że graf $A_n(\mathbb{F}_2)$ jest spójny. Jego wierzchołki tworzą w takim razie jeden cykl. Niech f będzie homomorfizmem grafu $A_n(\mathbb{F}_2)$

na graf $A_3(\mathbb{F}_2)$, polegającym na rzutowaniu każdego wierzchołka na 3 pierwsze składowe wektora skojarzonego z tym wierzchołkiem. Homomorfizm zachowuje krawędzie, stąd obrazem grafu spójnego musi być graf spójny. Ale $A_3(\mathbb{F}_2)$ nie jest spójny. □

Mamy więc tutaj podobieństwo między rodzinami $A_n(\mathbb{F}_q)$ i $D_n(\mathbb{F}_q)$.

Weźmy teraz $q > 2$. W roku 2013 profesor Ustimenko ([48]) udowodnił, że w tym przypadku każdy graf rodziny $A_n(\mathbb{F}_q)$ jest grafem spójnym.

Granica projektywna

Zaczniemy od dwóch brakujących definicji ([8, 27]).

Definicja 4.1. Niech $(I, \leqslant)$ będzie niepustym zbiorem uporządkowanym, niech $(A_i)_{i\in I}$ będzie rodziną obiektów tej samej kategorii. Załóżmy, że mamy rodzinę morfizmów $f_{ij} : A_j \to A_i$, dla wszystkich $i \leqslant j$ z następującymi własnościami:

1. f_{ii} jest identycznością na A_i;
2. $f_{ik} = f_{ij} \circ f_{jk}$ dla wszystkich $i \leqslant j \leqslant k$.

Wtedy para $((A_i)_{i\in I}, (f_{ij})_{i\leqslant j\in I})$ jest nazywana *odwrotnym systemem* w kategorii nad I.

W myśl powyższej definicji możemy zdefiniować odwrotny system dla naszych rodzin grafów przyjmując:

1. $\mathbb{N}$ jako wymagany zbiór uporządkowany,
2. $D_n(\mathbb{K}), n \in \mathbb{N}$ (lub analogicznie dla $A_n(\mathbb{K})$) jako wymaganą rodzinę obiektów kategorii grafów,
3. rodzinę wymaganych morfizmów jako homomorfizmy $f_{ij} : D_j(\mathbb{K}) \to D_i(\mathbb{K})$, dla wszystkich $i \leqslant j$, gdzie f_{ij} rzutuje graf $D_j(\mathbb{K})$ na jego i pierwszych składowych z zachowaniem $i-1$ pierwszych równań incydencji.

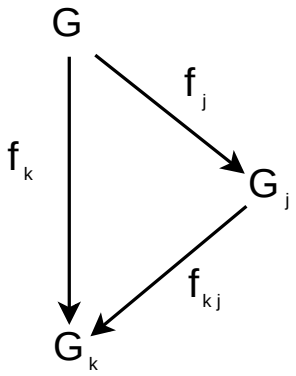
Mając odwrotny system pewnych obiektów tej samej kategorii, możemy zdefiniować dla tego systemu granicę rzutową. Nie wprowadzamy tej definicji dla struktury ogólnej, ale od razu granicę rzutową dla rodziny grafów indeksowanej liczbami naturalnymi.

Definicja 4.2. Niech $\{G_n\}, n \in \mathbb{N}$ będzie rodziną grafów. Niech $((G_n)_{n\in\mathbb{N}}, (f_{ij})_{i\leqslant j;\ i,j\in\mathbb{N}})$ będzie odwrotnym systemem grafów z odpowiednio zdefiniowanymi wszystkimi homomorfizmami f_{ij}. *Granica odwrotna* (lub *projektywna*) tego systemu istnieje jeśli istnieje jedyny graf G, wraz z rodziną homomorfizmów $f_i : G \to G_i, i \in \mathbb{N}$ taką, że

$$\forall_{j,k\in\mathbb{N};\ k\leqslant j}\ \ [f_k = f_j \circ f_{kj}].$$

Fakt ten oznaczamy

$$\varprojlim_{n\to\infty} G_n = G.$$



Rysunek 4.1: Diagram obrazujący warunek istnienia granicy projektywnej. Graf G_k ma być tym samym grafem, niezależnie, którą ścieżkę od G wybierzemy. $k \leqslant j$.

Jak łatwo zauważyć, zdefiniowany powyżej system odwrotny dla rodziny grafów $D_n(\mathbb{K})$ (analogicznie dla $A_n(\mathbb{K})$) będzie mieć dobrze określoną granicę projektywną gdy $n \to \infty$. Następujący wynik teoretyczny został opisany w pracy [30]:

Twierdzenie 4.10. *Niech $D_n(\mathbb{F}_q), n \in \mathbb{N}, q > 2$ będzie rodziną grafów o strukturze incydencji* (4.2) *nad ciałem skończonym $\mathbb{F}_q$. Wówczas*

$$\varprojlim_{n\to\infty} (D_n(\mathbb{F}_q)) \textit{ jest lasem.}$$

W przypadku grafów $A_n(\mathbb{F}_q), q > 2$ otrzymujemy [48]:

Twierdzenie 4.11. *Niech $A_n(\mathbb{F}_q), n \in \mathbb{N}, q > 2$ będzie rodziną grafów o strukturze incydencji* (4.3) *nad ciałem skończonym $\mathbb{F}_q$. Wówczas*

$$\varprojlim_{n\to\infty} (A_n(\mathbb{F}_q)) \textit{ jest drzewem.}$$

Zauważmy, że obie te rodziny grafów w nieskończonej granicy nie posiadają żadnych cykli, co potwierdza tezę, że talia tych grafów rośnie z n do nieskończoności.

Uwaga: Jeśli chcielibyśmy szukać innych rodzin grafów algebraicznych, które mogą mieć zastosowanie w przedstawionym przez nas algorytmie szyfrowania, dobrymi kandydatami są rodziny, które w granicy odwrotnej dążą właśnie do drzewa lub lasu.

Rząd przekształcenia cyklicznego

Załóżmy teraz, że mamy ustalone $n \in \mathbb{N}$, wybrany pierścień $\mathbb{K}$ o skończonej ilości elementów, oraz ustalone dwa elementy klucza a_1, a_2 w ten sposób, że $a_1 + a_2 \in Reg(\mathbb{K})$. Dla wybranego grafu dwudzielnego $A_n(\mathbb{K})$ (lub $D_n(\mathbb{K})$) operator $N_{(a_1,a_2)}$ przekształca punkt na punkt oraz linię na linię. Działanie składania operatora $N_{(a_1,a_2)}$ jest łączne i dla ustalonego elementu $x \in \mathbb{K}^n$ tworzy grupę cykliczną.

Warto zadać teraz pytanie: jak duże będą to cykle i jak szybko będą rosły ze wzrostem n (o ile w ogóle będą rosły)? Szybki wzrost rzędu każdej takiej grupy cyklicznej pozwalałby wierzyć w sensowność wykorzystania przekształcenia $N_{(a_1,a_2)}$, a także innych przekształceń N_K w problemie logarytmu dyskretnego. A problem logarytmu dyskretnego jest naturalną bazą problemu Diffiego-Hellmana wymiany kluczy ([11]) oraz algorytmu ElGamala klucza publicznego ([14]).

Na dzień dzisiejszy nie są znane oszacowania czysto matematyczne rzędów tych grup cyklicznych, natomiast autor niniejszej pracy przeprowadził szereg eksperymentów komputerowych, których wyniki były publikowane w [26] i które prezentujemy w tej sekcji.

Założenia eksperymentu

Najpierw ustalamy $\mathbb{K} = \mathbb{Z}_q$ dla konkretnego q. Następnie każdorazowo losujemy parę (a_1, a_2) oraz x należące do $\mathbb{Z}_q$ i przeprowadzamy operację $x := N_{(a_1,a_2)}(x)$ tak długo, aż dojdziemy do wektora początkowego x. Każdy z testów był przeprowadzany przynajmniej 20 razy.

q jest liczbą pierwszą

Patrząc na tabelę 4.1 łatwo zauważyć, że każdorazowo długość cyklu jest potęgą liczby pierwszej q. Powyższe wyniki były identyczne bez względu na wybór parametrów x, a_1, a_2. W przypadku $q = 2$ otrzymywaliśmy długość cyklu równą potędze 2, ale cykle te posiadały różne długości, w zależności od wyboru

$q \setminus n$	4	10	30	50	100	200	400	600	1000
3	9	27	81	81	243	243	729	729	2187
5	5	25	125	125	125	625	625	625	3125
7	7	49	49	343	343	343	2041	2041	2041
11	11	11	121	123	121	1331	1331	1331	1331

Tabela 4.1: Długości cykli dla $K = \mathbb{Z}_q$, gdzie q jest liczbą pierwszą, grafy $D_n(\mathbb{K})$

x, a_1, a_2.

q jest liczbą złożoną

$q \setminus n$	4	10	30	50	100	200	400	600
4	16	32	64	128	256	512	1024	2048
6	72	432	2592	5184	31104	62208	brak wyników	
8	32	64	128	256	512	1024	2048	4096
9	27	81	243	243	729	729	2187	2187
15	45	675	10125	10125	30375	151875	455625	455625

Tabela 4.2: Długości cykli dla małych liczb złożonych q, grafy $D_n(\mathbb{K})$, $K = \mathbb{Z}_q$

Tabela 4.2 prezentuje wyniki testu dla q będącego liczbą złożoną. Co ciekawe długość cyklu była każdorazowo liczbą, której rozkład na czynniki pierwsze składał się z potęg czynników pierwszych q. Znacznie szybsze tempo wzrostu można zaobserwować dla liczb, które nie są potęgami liczby pierwszej ($q = 6, 15$).

Porównanie rodzin $D_n(\mathbb{Z}_q)$ i $A_n(\mathbb{Z}_q)$

W ostatnim z testów ustaliliśmy przykładową wartość $q = 15$, aby zaobserwować tempo wzrostu długości cykli dla grafów $A_n(\mathbb{Z}_q)$ i $D_n(\mathbb{Z}_q)$ (tabele 4.3, 4.4). Jak widzimy długość cyklu rośnie zdecydowanie szybciej dla grafów z rodziny $A_n(\mathbb{Z}_q)$, co czyni tę rodzinę lepszą w zastosowaniach opartych o problem logarytmu dyskretnego.

Uwaga: wszystkie powyższe testy potwierdzają tezę, iż talia obu rodzin grafów dąży do nieskończoności przy $n \to \infty$.

n_{MIN}	n_{MAX}	dł. cyklu
4	4	45
5	8	225
9	24	675
25	26	3375
27	80	10125
81	120	30375
140	240	151875
260	620	455625
640	720	2278125
760		6834375

Tabela 4.3: Długość cykli dla $q = 15$, rodzina grafów $A_n(\mathbb{Z}_q)$

n_{MIN}	n_{MAX}	dł. cyklu
4	7	45
8	17	225
18	53	675
54	65	3375
150	249	10125
250	299	30375
300	649	151875
650	1000	455625

Tabela 4.4: Długości cykli dla $q = 15$, grafy $D_n(\mathbb{Z}_q)$

Rozdział 5

Własności Madrygi algorytmów. Rozszerzenie algorytmu

Jednym z pierwszych samodzielnych wyników autora tej pracy było zbadanie jak zachowuje się rodzina $D_n(\mathbb{K})$ w kontekście wymagań Madrygi, opisanych w sekcji 1.1.3. Szczególnie interesowały nas zmiany jakie można zaobserwować w korespondującym szyfrogramie przy zmianie jednego elementu klucza (z ustalonym tekstem jawnym) lub tekstu jawnego (z ustalonym kluczem).

Badaliśmy zmiany pojedynczych elementów pierścienia $\mathbb{K}$, choć w oryginale Madryga pisał o zmianach bitów. Wyniki eksperymentów komputerowych były następujące [25]:

1. przy zmianie pojedynczego elementu klucza i ustalonym tekście jawnym, otrzymywano od 92% (dla haseł 1-2 elementowych) do 96% zmian w korespondujących szyfrogramach;

2. przy zmianie pojedynczego elementu tekstu jawnego i ustalonym kluczu zmianie ulegało jedynie kilka elementów w korespondującym szyfrogramie, umiejscowionych w bliskim otoczeniu zmiany.

Drugi z powyższych punktów był bardzo nieporządanym efektem, a analiza matematyczna równań 4.2 wykazała, że problem rzeczywiście istnieje. Wystarczy wypisać 8 kolejnych równań w postaci

$$y_i = x_i + \epsilon_i(y_1\ x_{k(i)}) + (1 - \epsilon_i)(x_1\ y_{k(i)}),$$

gdzie $\epsilon \in \{0, 1\}$, a funkcja $k : \mathbb{N} \to \mathbb{N}$ bierze odpowiednią składową wcześniejszą niż i. Następnie wystarczy zakładać zmiany w kolejnych czterech równaniach i przeanalizować, które współrzędne ulegną zmianie.

5.1 Rozszerzenie algorytmu o odwzorowanie afiniczne

Aby zaradzić problemowi zbyt małych zmian szyfrogramu, postanowiliśmy wykorzystać pomysł z algorytmu Imai-Matsumoto opisany w sekcji 2.2. Oznaczając jak poprzednio nasze przekształcenie grafowe jako

$$N_K : \mathbb{K}^n \to \mathbb{K}^n,$$

dodajemy do procedury szyfrowania dwa kroki, będące odwracalnymi przekształceniami afinicznymi

$$S, T : \mathbb{K}^n \to \mathbb{K}^n, \deg(S) = \deg(T) = 1$$

i finalnie nasza procedura szyfrowania prezentuje się następująco:

$$S \circ N_K \circ T : \mathbb{K}^n \to \mathbb{K}^n.$$

W odróżnieniu od pomysłu stosowanego w kryptografii wielu zmiennych, nie tworzymy równań publicznych, lecz uważamy parametry przekształceń S oraz T za część hasła i generujemy je na podstawie klucza K.

Uzasadnienie stosowania przekształceń S, T stopnia 1 jest następujące: złożenie przekształcenia wyższego stopnia z przekształceniem stopnia 1 nie zmieni stopnia wynikowego przekształcenia. Czyli przekształcenie wynikowe $S \circ N_K \circ T$ będzie nadal przekształceniem 3 stopnia. Gdyby próbować w miejsce S lub T użyć przekształcenia wyższego stopnia, takiej pewności nie ma. Stopień wynikowego przekształcenia mógłby wzrosnąć, zmaleć lub pozostać bez zmian i za każdym razem pojawiałoby się pytanie: czy takie przekształcenia złożone razem będą miały odpowiedni stopień?

Zbiór przekształceń afinicznych jest bardzo liczny. Naturalne pytanie brzmi: jakie przekształcenia wybrać w naszym przypadku, aby poprawić własność „mieszania" szyfrogramu przy ustalonym kluczu?

Rzut oka na układy równań (4.2) oraz (4.3) pozwala zauważyć, że rozwiązując ten układ od początku w skład każdego rozwiązania wchodzi składowa x_1 lub y_1. W związku z tym chcielibyśmy, aby zmiana dowolnej składowej tekstu otwartego x_i powodowała zmianę pierwszej składowej x_1. Wtedy y_1 również się zmieni, ponieważ za każdym razem $y_1 = x_1 \pm a_j$ (minus gdy przechodzimy z y w x). Nasze próby użycia przekształcenia afinicznego wyglądały następująco (przekształcenia wg kolejności ich powstawania):

1. do składowej x_1 dodać wszystkie pozostałe składowe wektora x;

2. do składowej x_1 dodać kombinację liniową wszystkich pozostałych składowych wektora x;

3. ze względu na regularność równań i specyfikę pierwszej składowej – zastosować permutację kilku elementów wektora x (w tym x_1), a następnie zastosować punkt 2.

Ponowne testy wykazały, że system kryptograficzny używający przekształcenia $S \circ N_K \circ T$ przy ustalonym kluczu i zmianie pojedynczego elementu wiadomości otwartej powoduje zmiany ponad 90% elementów w korespondującym szyfrogramie.

Kolejnym argumentem za stosowaniem operatorów S i T jest utrudnienie pracy kryptoanalitykom poprzez „ukrycie" użytego do szyfrowania grafu „za" tymi przekształceniami. Nawet posiadając pełną wiedzę o strukturze incydencji grafu, użytego do szyfrowania (znajomość równań algebraicznych), nie można wykonywać spacerów po wierzchołkach grafu i testować różnych ścieżek, dopóki graf jest „ukryty".

Przykład algorytmu, polegającego na dodaniu do x_1 kombinacji liniowej pozostałych współrzędnych, wygenerowanej przy pomocy klucza (stosowanego k-elementowymi blokami) prezentuje algorytm 5.1.

Algorytm 5.1 Dodanie do x_1 kombinacji liniowej wszystkich następnych elementów, zależnej blokowo od klucza

Wejście: $K = (a_1, a_2, \ldots, a_k),\ a_i \in \mathbb{K}$,
$x = (x_1, x_2, \ldots, x_n) \in P$ — wierzchołek grafu,
$k < n$

Wyjście: $x = (x_1 + \sum_{j=2}^{n} x_j \cdot a_{f(j)}, x_2, x_3, \ldots, x_n)$

1: $d := (n-1)$ div k
2: $r := (n-1)$ mod k
3: $i := 2$
4: **for** $j = 1, 2, \ldots, d$
5: **for** $s = 1, 2, \ldots, k$
6: $x_1 := x_1 + a_s \cdot x_i$
7: $i := i + 1$
8: **for** $s = 1, 2, \ldots, r$
9: $x_1 := x_1 + a_s \cdot x_i$
10: $i := i + 1$

Rozdział 6

Wydajność naszego systemu kryptograficznego

W rozdziale tym przedstawimy krótką analizę złożoności algorytmów szyfrowania symetrycznego, opartych na naszych rodzinach grafów. Następnie opiszemy implementacje tych algorytmów w kolejności ich występowania. Na koniec pokażemy na ile nasze nowe implementacje są szybsze od poprzednich oraz opiszemy co nowego wprowadziliśmy w tych implementacjach.

6.1 Oszacowanie złożoności algorytmów

Zacznijmy od oszacowania złożoności obliczeniowej pojedynczego kroku szyfrowania naszym algorytmem N_a, gdzie a jest dowolnym symbolem hasła. Patrząc na układ równań (3.1) zostawiamy niewiadome po lewej stronie i rozwiązując je po kolei od pierwszego równania mamy:

- jedno dodawanie z wykorzystaniem a ($x_0 = y_0 + a$ lub $y_0 = x_0 + a$);
- jedno dodawanie (lub odejmowanie) i jedno mnożenie dla każdego równania ($n - 1$ równań);
- jedną operację na indeksie tablicy i – wybieranie jednego z poprzednich indeksów;
- podstawień oraz operacji wyszukania w tablicy jako zdecydowanie szybszych nie liczymy.

Ostatecznie możemy pojedynczy krok oszacować jako $O(3n) = O(n)$ działań obliczeniowych na liczbach całkowitych.

Dla klucza $K = (a_1, a_2, \ldots, a_k)$ takich kroków będzie k, więc podstawowy algorytm szyfrowania dla obu rodzin grafów będzie rzędu $O(nk)$. Dla ustalonej długości klucza k, otrzymamy wzrost liniowy względem rozmiaru danych i odwrotnie – przy ustalonym n będziemy mieli wzrost liniowy względem rozmiaru klucza.

W tym momencie mamy odpowiedź na pytanie „dlaczego stosując rozszerzenie algorytmu podstawowego o przekształcenia afiniczne S i T należy wybierać tylko niektóre przekształcenia?". Ogólnie przekształcenie afiniczne o gęstej macierzy kwadratowej posiada złożoność obliczeniową $O(n^2)$, więc taki dodatkowy krok spowodowałby spadek wydajności całego algorytmu o rzędy wielkości.

Przy stosowaniu przekształceń proponowanych przez nas mamy:

- $2n - 1$ działań przy dodaniu kombinacji liniowej wszystkich $n - 1$ następnych elementów wiadomości do elementu pierwszego;
- dowolną permutację można również wykonać w czasie liniowym $O(n)$.

Jeśli chodzi o złożoność pamięciową, to w naszej implementacji stosujemy dwie tablice n elementowe (na wektory x oraz y) i w zależności od kroku algorytmu jeden traktujemy jako dane, a drugi jako niewiadome. Dodatkowo potrzebna nam jest tablica k elementów klucza. Czyli potrzebujemy co najmniej $(2n + k)r$ pamięci, gdzie $r \in \{1, 2, 4, 8\}$ jest rozmiarem danych i odpowiada ilości bajtów potrzebnych do przechowania jednego elementu pierścienia K.

Parametry przekształceń S oraz T wyliczamy na podstawie k oraz ewentualnie generatora liczb pseudolosowych. W zależności od parametru, który chcemy optymalizować (szybkość czy pamięć) możemy: albo wyliczyć je na początku i zapisać w pamięci, albo wyliczać na bieżąco, albo nawet potraktować k jako wektor parametrów kombinacji liniowej stosowany blokowo (gdy k jest znacznie mniejsze od n powtarzamy bloki k elementowe, aby w kombinacji liniowej związać pierwszą współrzędną ze wszystkimi pozostałymi).

6.2 Implementacje historyczne

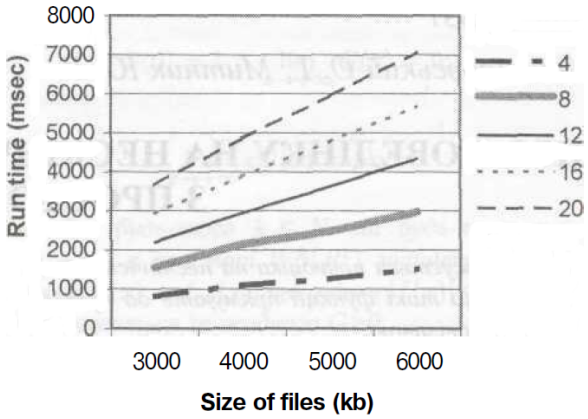
Pierwsza implementacja szyfru symetrycznego opartego na grafach algebraicznych powstała w 2001 roku [46] i nazywała się CRYPTIM. Wykorzystano w niej grafy $D_n(\mathbb{Z}_{127})$ i szyfrowano tylko tekst i grafikę. Liczba 127 jest pierwsza, więc pierścień $\mathbb{Z}_{127}$ jest ciałem skończonym. Wykorzystanie ciała skończonego jako bazy algorytmu było na ten moment jedyną opcją, ponieważ badanie rodziny grafów $D_n(\mathbb{K})$, dla $\mathbb{K}$ będącego pierścieniem przemiennym z jedynką rozpoczęto w latach późniejszych. Zastosowanie pierścienia modulo 127 wiązało się z dodatkowym narzutem czasowym na operację $mod\ 127$ po każdym

działaniu arytmetycznym. Mimo to system komunikacji oparty o ten algorytm szyfrowania powstał na Uniwersytecie Południowego Pacyfiku (ang. *University of the South Pacific*) na wyspach Fidżi i był wykorzystywany w praktyce do przesyłania danych między kilkoma wyspami. W tabeli 6.1 przytaczamy wyniki prędkości pracy tego systemu.

$n[kB]\backslash k$	9	13	17	21	25
1	1	1	1	2	2
2	2	3	3	4	4
3	4	6	8	9	11
4	16	16	23	30	33
5	22	27	35	44	52
6	38	54	64	88	105

Tabela 6.1: Cryptim: Czas w sekundach zaszyfrowania/deszyfrowania wiadomości o n kB długości hasłem o długości k elementów nad alfabetem $\mathbb{Z}_{127}$.

Kolejna implementacja nazywała się CRYPTALL, powstała w 2004 roku na Uniwersytecie Sułtańskim w Omanie (ang. *Sultan Qabos University*) [44]. Do szyfrowania wykorzystano ciała skończone $\mathbb{Z}_{257}$, z konsekwencjami pracy z operacjami modulo jak poprzednio. Eksperymentalne wyniki szybkości systemu przedstawia rysunek 6.1.



Rysunek 6.1: Wydajność systemu CRYPTALL, każda linia to szyfrowanie kluczem o innej długości.

6.3 Wydajność i szczegóły naszej implementacji

6.3.1 Dodatkowe wymagania dla pierścieni

W rozdziale 4 opisaliśmy różne własności rodzin grafów A_n oraz D_n. Większość z własności dowiedzionych matematycznie dotyczyło rodzin tworzonych nad ciałem skończonym $\mathbb{F}_q$. Co prawda w roku 1998 ([45]) został zaproponowany szyfr symetryczny oparty o grafy $D_n(\mathbb{K})$, analogiczny do szyfru opartego o grafy $D_n(\mathbb{F}_q)$, jednak stan wiedzy matematycznej nie był wystarczający, by uzasadnić poprawność i bezpieczeństwo tego systemu.

Dopiero w roku 2007 ([47]) profesor Ustimenko stworzył grafy "podwójne" skierowane $DD_n(\mathbb{K}, Reg(\mathbb{K}))$ i uzasadnił brak istnienia w tych grafach odpowiednio dużych *diagramów przemiennych* (analogia cyklu w grafach nieskierowanych). Kluczowym założeniem był dodatkowy wymóg na hasło $K = (a_1, a_2, \ldots, a_k)$:

$$\forall i \quad (a_i + a_{i+1}) \in Reg(\mathbb{K}), \quad 0 < i < k. \tag{6.1}$$

Przypominamy, że $Reg(\mathbb{K})$ to zbiór elementów odwracalnych w pierścieniu $\mathbb{K}$, który dla pierścienia skończonego jest równoważny ze zbiorem elementów, które nie są dzielnikami 0.

Pierścienie modulo, w których na maszynach wykonuje się naturalne działania na liczbach całkowitych są pierścieniami modulo $2^s, s \in \{8, 16, 32, 64\}$ (całkowity typ danych 1 Bajtowy odpowiada pierścieniowi $\mathbb{Z}_{2^8}$, 2 bajtowy pierścieniowi $\mathbb{Z}_{2^{16}}$, itd.). Wobec tego warunek (6.1) dla tych pierścieni przyjmuje postać:

$$\forall i \; \exists b \in \mathbb{Z}_{2^s} \quad [a_i + a_{i+1} = 2b + 1], \quad 0 < i < k,$$

gdzie $s \in \{8, 16, 32, 64\}$.

W naszej implementacji nieparzystość dwóch kolejnych elementów hasła zapewniliśmy prostym i przejrzystym trikiem: zanim przystąpimy do szyfrowania wykonujemy przekodowanie hasła, ustawiając ostatni bit „na sztywno" w jednym kroku na 0, a w następnym na 1, w kolejnym na 0, itd. Aby nie tracić bitów znaczących klucza, przesuwamy bity niewykorzystane na kolejne symbole klucza.

Przykład 6.1. *Załóżmy, że operujemy na liczbach 1-bajtowych i klucz szyfrowania składa się z dwóch bajtów, które w zapisie binarnym przedstawiają się następująco:*

`11010101  01100101`.

Po zakodowaniu klucza otrzymujemy 3 bajty zmodyfikowanego hasła:

`1101010` **0** `1011001` **1** `01` *00000* **0**,

gdzie elementy pogrubione są zawsze ustawiane na naprzemiennie 0 i 1, a elementy zapisane pochyłą czcionką są po prostu dopełnieniem zerami nieznaczącymi ostatniego bajtu. Czcionka normalna obrazuje bity znaczące klucza wejściowego.

Uwaga: Powyższy sposób przekodowania klucza ma jeszcze jedną zaletę. Mianowicie zapewnia własność $a_i \neq -a_{i+1}$ dla każdego i, czyli nigdy nie nastąpi redukcja hasła poprzez powrót do wierzchołka poprzedniego.

Oczywiście tak powstały, przekodowany klucz ma więcej znaków niż klucz K. Jeśli k było długością klucza, to zaproponowana zmiana bitów wymaga $O(k)$ operacji.

6.3.2 Wyniki eksperymentów

Wyniki przedstawione w tej sekcji przeprowadzone zostały w 2007 roku na przeciętnym jak na tamte czasy komputerze o parametrach:

- procesor AMDAthlon 1.46 GHz,
- 1 GB pamięci RAM,
- system Windows XP.

Całość programu została napisana w języku Java z myślą o jak najszybszym działaniu algorytmów. Powszechnie znane algorytmy RC4 oraz DES (oba opisywane między innymi w pozycji [42]), których użyliśmy w porównaniach szybkości wzięte zostały ze standardowej biblioteki Javy dla zastosowań kryptograficznych *javax.crypto*.

W naszej implementacji wykorzystaliśmy arytmetykę jedno, dwu oraz czterobajtową, oznaczając na wykresach i tabelach dla uproszczenia odpowiadające im algorytmy jako $G1$, $G2$ oraz $G4$. Każdy z testów był powtarzany co najmniej 20-krotnie, a wyniki zostały uśrednione.

Narzut na przekodowanie hasła nie był wliczany do testu, jednak można go uważać za pomijalny przy kluczu długości kilku lub kilkunastu bajtów, gdy dane są rzędu dziesiątek megabajtów.

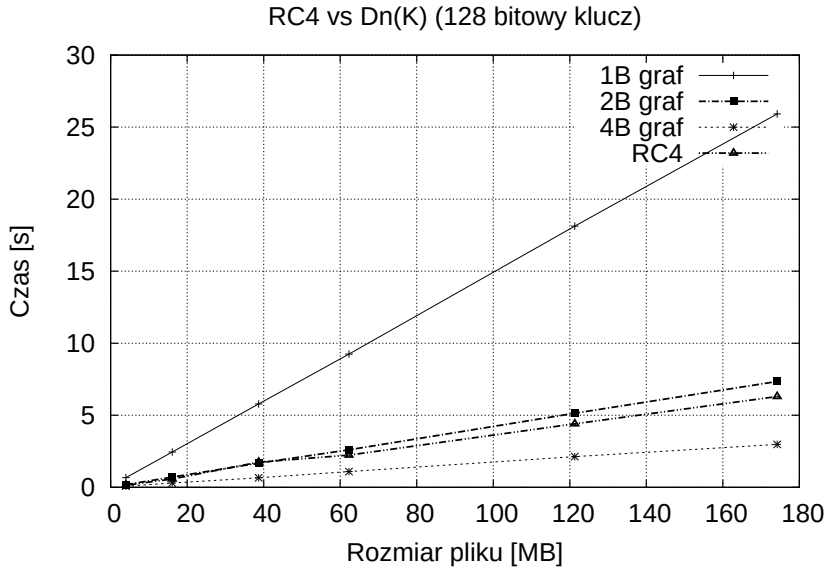
Porównanie z algorytmem RC4

Algorytm RC4 jest szyfrem strumieniowym zaproponowanym przez Rona Rivesta na konferencji w 1987 roku, szeroko znanym i dyskutowanym w literaturze. Szyfr ten został złamany i na dzień dzisiejszy nie gwarantuje odpowiedniego bezpieczeństwa. Wybraliśmy go jednak do porównania prędkości działania ze

Plik [MB]	RC4 [s]	G1 [s]	G2 [s]	G4 [s]
4	0.15	0.67	0.19	0.08
16.1	0.58	2.45	0.71	0.30
38.7	1.75	5.79	1.68	0.66
62.3	2.24	9.25	2.60	1.09
121.3	4.41	18.13	5.14	2.13
174.2	6.30	25.92	7.35	2.98

Tabela 6.2: Wydajność naszego systemu w porównaniu z algorytmem RC4. Grafy $D_n(\mathbb{K})$, klucz 128-bitowy.

względu na ogólnie znaną wysoką wydajność tego szyfru. Szybkość RC4 nie zależy od długości klucza, natomiast algorytm wykorzystujący grafy zależy liniowo od długości klucza. W związku z tym w porównaniu użyliśmy klucza o z góry ustalonej długości, jaką zalecała dokumentacja Javy dla tego algorytmu – 128 bitów.



Rysunek 6.2: Wydajność naszego systemu w porównaniu z algorytmem RC4. Grafy $D_n(\mathbb{K})$.

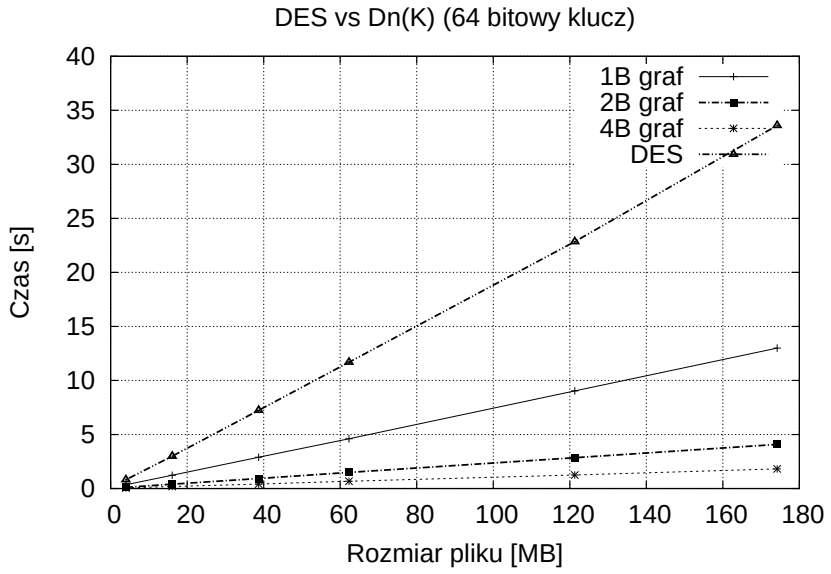
Jak widzimy z tabeli 6.2 oraz wykresu 6.2 algorytm wykorzystujący arytmetykę 2-bajtową (graf $D_n(\mathbb{Z}_{2^{16}})$) był nieznacznie wolniejszy od RC4, natomiast G4 (graf $D_n(\mathbb{Z}_{2^{32}})$) był ponad 2 razy szybszy.

Plik [MB]	DES [s]	G1 [s]	G2 [s]	G4 [s]
4	0.81	0.35	0.11	0.05
16.1	2.99	1.23	0.40	0.18
38.7	7.24	2.90	0.92	0.41
62.3	11.69	4.60	1.49	0.68
121.3	22.85	9.03	2.85	1.25
174.2	33.60	13.00	4.08	1.82

Tabela 6.3: Wydajność naszego systemu w porównaniu z algorytmem DES. Grafy $D_n(\mathbb{K})$, klucz 64-bitowy.

Porównanie z algorytmem DES

DES jest algorytmem blokowym, który przez pewien czas był standardem szyfrowania kryptografii symetrycznej. Został wybrany do porównania z podobnych powodów co RC4 – jest algorytmem szeroko znanym, wiadomo na jego temat praktycznie wszystko, a inne, nowsze, algorytmy (3DES, AES) są od niego wolniejsze. Szyfrowanie zostało wykonane kluczem 64-bitowym, zalecanym dla algorytmu DES (implementacja Javy DES nie przyjmuje kluczy o innej długości).



Rysunek 6.3: Wydajność naszego systemu w porównaniu z algorytmem DES. Grafy $D_n(\mathbb{K})$.

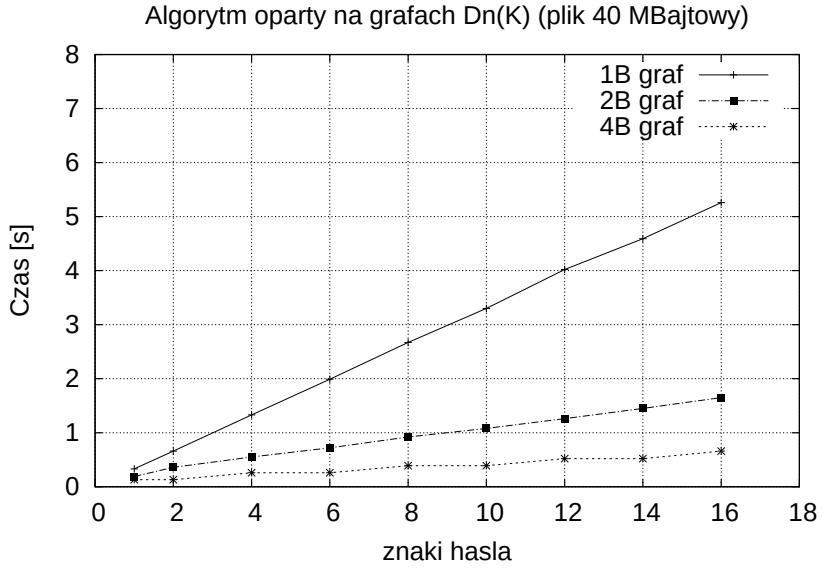
Porównując DES z naszym najszybszym algorytmem, widać przewagę szybkości algorytmu opartego o grafy o ponad rząd wielkości. Co za tym idzie, gdyby na przykład porównać nasz algorytm G4 z szyfrem AES, akceptującym klucze długości 128, 192 oraz 256 bitów, G4 byłby również znacznie szybszy.

Zależność liniowa od długości hasła

W ostatnim z testów (tabela 6.4, rysunek 6.4) ustaliliśmy na sztywno wielkość pliku wejściowego i szyfrowaliśmy dane kluczem różnej długości, aby sprawdzić i zobrazować zależność liniową od hasła przy ustalonym tekście otwartym.

Zauważmy, że zmiana pierścienia 1-bajtowego na 2-bajtowy powoduje dwukrotną redukcję liczby danych wejściowych liczonej w elementach odpowiedniego pierścienia ($\mathbb{Z}_{2^8}$ i $\mathbb{Z}_{2^{16}}$), a także prawie dwukrotne skrócenie liczby elementów hasła (a każdy z elementów hasła odpowiada jednej zmianie wierzchołka w grafie), „prawie" ponieważ hasło przekodowujemy. Z tego powodu zmiana arytmetyki na dwukrotnie większą (1B na 2B, 2B na 4B) powoduje każdorazowo blisko 4-krotne przyspieszenie działania naszych algorytmów. Może się to jednak wiązać z mniejszymi zmianami liczby pojedynczych bitów w kontekście wymagań Madrygi (problem poruszany w rozdziale 5).

Reprezentacji na liczbach 64-bitowych nie było sensu testować na komputerach 32-bitowych.



Rysunek 6.4: Szyfrowanie grafem $D_n(\mathbb{K})$ – funkcja liniowa długości hasła przy ustalonej wielkości pliku (40 MB).

Hasło [B]	G1 [s]	G2 [s]	G4 [s]
1	0.33	0.19	0.13
2	0.66	0.36	0.13
4	1.33	0.55	0.26
6	1.99	0.72	0.26
8	2.67	0.92	0.39
10	3.30	1.08	0.39
12	4.02	1.26	0.52
14	4.59	1.45	0.52
16	5.26	1.65	0.66

Tabela 6.4: Szyfrowanie grafem $D_n(\mathbb{K})$ – funkcja liniowa długości hasła przy ustalonej wielkości pliku (40 MB).

File [MB]	G1 [s]	G2 [s]	G4 [s]
4	0.04	0.02	0.01
16.1	0.12	0.10	0.08
38.7	0.32	0.24	0.20
62.3	0.50	0.40	0.30
121.3	0.96	0.76	0.60
174.2	1.39	0.96	0.74

Tabela 6.5: Narzut czasowy na operację S oraz T rozszerzającą algorytm N_K do algorytmu $S \circ N_K \circ T$.

Narzut przekształcenia afinicznego

Serię testów uzupełnia tabela 6.5, w której obrazujemy na tych samych plikach ile czasu wymaga pojedyncza operacja afiniczna, polegająca na dodaniu wszystkich następnych składowych wektora x do składowej x_1. W przypadku kombinacji liniowej o innych parametrach otrzymamy tutaj maksymalnie dodatkowe $n-1$ działań mnożenia, czyli dwu lub trzykrotny wzrost narzutu.

Podsumowanie

Systemy CRYPTIM i CRYPTALL wykorzystywały grafy $D_n(\mathbb{F}_q)$, gdzie q było liczbą pierwszą do szyfrowania symetrycznego w oparciu o wiedzę dotyczącą ciał skończonych. W naszym systemie dokonaliśmy szeregu usprawnień:

- wykorzystaliśmy pierścienie przemienne $\mathbb{Z}_{2^s}$ z naturalną dla maszyn arytmetyką 1, 2 i 4-bajtową;

- dokonaliśmy niezbędnej korekty hasła wymaganej dla pierścieni ogólnych, przy okazji zapewniając brak możliwości zredukowania hasła;
- odkryliśmy małą podatność szyfrogramu na niewielkie zmiany tekstu otwartego przy ustalonym kluczu;
- rozszerzyliśmy algorytm szyfrujący o złożenia z przekształceniami afinicznymi odpowiednio dobranymi do problemu;
- sprawdziliśmy, że wprowadzone rozszerzenie poprawia słabą stronę algorytmu oraz uzasadniliśmy, że nie wpływa negatywnie na jego silne strony;
- porównaliśmy wydajność naszego systemu z algorytmami, które przez wielu specjalistów uważane są za szybkie, obrazując sprawność naszych algorytmów.

Jeśli próbowalibyśmy porównać szybkość naszego systemu z systemem CRYPTALL, to dla klucza 8-elementowego nad alfabetem $\mathbb{F}_{257}$ wiadomość o długości $4000kB \simeq 4MB$ szyfrowano ponad 2 sekundy, a $6000kB \simeq 6MB$ około 3 sekund.

Najbliżej tej wielkości klucza w naszym teście mamy przy kluczu 64-bitowym (tabela 6.3, 8 elementów nad alfabetem $\mathbb{Z}_{256}$), gdzie najwolniejszy algorytm szyfrował średnio niecałe 3 sekundy plik rozmiaru $38.7MB$, natomiast G4 poniżej 2 sekund plik ponad $170MB$.

Aż tak duża różnica może wynikać z innej klasy komputera lub błędnego pomiaru czasu dla systemu CRYPTALL – system ten szyfrował pliki o różnych rozszerzeniach i być może autorzy odliczali czas "pracy systemu" razem z czasem potrzebnym na wczytanie i zapis pliku. W naszych doświadczeniach mierzyliśmy jedynie czas działania algorytmu szyfrowania i deszyfrowania (oczywiście czasy są takie same ze względu na symetrię odwzorowań).

6.4 Nowa implementacja

W roku 2013 rozpoczęliśmy pracę nad nową implementacją naszych algorytmów, tym razem w środowisku Microsoft Visual Studio i języku C#. System budowany jest w oparciu o kilka hierarchii klas i interfejsów, zgodnie z wzorcami projektowymi znanymi z inżynierii oprogramowania.

Rozbicie na szczegółowe klasy i interfejsy nastąpiło zgodnie ze wzorcem *strategii* (ang. *Strategy Design Pattern*) i ma na celu:

- możliwość łatwiejszego niż w poprzednim systemie dodawania kolejnych, kompatybilnych z już działającym systemem algorytmów,

- poprawienie możliwości składania algorytmów z tej samej grupy w nowe algorytmy przy zastosowaniu wzorca *dekorator*, dzięki czemu łatwo można zmieniać oba kroki afiniczne w algorytmie rozszerzonym,

- możliwość budowania meta-algorytmów z abstrakcyjnie zdefiniowanych kroków (wzorzec *metoda szablonowa*). Przykładem jest tutaj ogólny algorytm szyfrowania kluczem pewnej długości, składający się z abstrakcyjnych pojedynczych kroków wykorzystujących kolejne elementy klucza. Operacji innych dla kroków parzystych i nieparzystych. W naszych algorytmach kroki te odpowiadają przejściom w grafie od punktu do linii lub odwrotnie.

Mniejsze, łatwiej zarządzane kawałki kodu, pozwalają też na dokładniejsze przetestowanie tego kodu z wykorzystaniem kolejnej technologii inżynierii oprogramowania – testów automatycznych.

W naszej ostatniej implementacji dokonaliśmy niewielkiego progresu w stosunku do systemu w języku Java poprzez dodanie do puli algorytmów rodziny D_n algorytm działający na arytmetyce 64-bitowej ($D_n(\mathbb{Z}_{2^{64}})$) oraz wszystkie algorytmy dla rodziny $A_n(\mathbb{K}), \mathbb{K} \in \{\mathbb{Z}_{2^8}, \mathbb{Z}_{2^{16}}, \mathbb{Z}_{2^{32}}, \mathbb{Z}_{2^{64}}\}$.

Zupełnie nowym elementem systemu jest wprowadzenie algorytmów pracujących dla obu rodzin grafów nad pierścieniami boolowskimi.

6.4.1 Pierścień boolowski

Pierścień boolowski $\mathbb{B} = (B, \oplus, \otimes)$ w rozumieniu algebraicznym wprowadzamy poprzez zdefiniowanie na pewnym zbiorze B działań dodawania i mnożenia jako

- $x \otimes y := x \wedge y$,

- $x \oplus y := (x \wedge \neg y) \vee (\neg x \wedge y)$.

Zauważmy, że przedstawione tutaj operacje iloczynu logicznego oraz różnicy symetrycznej możemy w praktyce łatwo realizować zarówno programistycznie (operatory binarne na liczbach całkowitych), jak i sprzętowo (bramki logiczne). W związku z tym zastosowanie pierścienia boolowskiego jako podstawy budowy rodzin grafów D_n oraz A_n nabiera sensu, szczególnie jeśli chodzi o rozwiązania czysto sprzętowe.

Nasze testy pokazały, że na komputerze osobistym o architekturze x64 przekształcenie odpowiadające ścieżce w grafie $D_n(\mathbb{B}_{2^s})$ działa szybciej niż przekształcenie $D_n(\mathbb{K}_{2^s})$ jedynie dla przypadku $s = 64$. W pozostałych przypadkach ($s \in \{8, 16, 32\}$) algorytm w pierścieniu boolowskim był nieznacznie wolniejszy. Obserwacja ta pozwala wysnuć przypuszczenie, że przy wykonywaniu

działań w arytmetyce „naturalnej” dla architektury komputerów (np. na liczbach 32 bitowych w architekturze x86 lub na liczbach 64 bitowych w architekturze x64) zastosowanie pierścienia boolowskiego przyspiesza nasz algorytm.

Rozdział 7

Wstępna kryptoanaliza systemu. Możliwości poprawy bezpieczeństwa.

7.1 Atak brutalny

Zacznijmy od przypomnienia założeń. Mamy pierścień przemienny $\mathbb{K}$ o q elementach. Test otwarty składający się z n elementów tego pierścienia szyfrujemy kluczem k-elementowym nad $\mathbb{K}$. W ten sposób wektor

$$x^0 = (x_1, x_2, \ldots, x_n)$$

będący punktem, przekształcamy na pewien wektor

$$z^k = (z_1, z_2, \ldots, z_n)$$

procedurą

$$M = x^0 \xrightarrow{a_1} y^1 \xrightarrow{a_2} x^2 \xrightarrow{a_3} y^3 \rightarrow \quad \ldots \quad \xrightarrow{a_k} z^k = E,$$

gdzie kluczem jest

$$K = (a_1, a_2, \ldots, a_k).$$

Dodatkowo załóżmy, że $k < n/2$ (zgodnie z naszym zaleceniem na długość klucza).

Jeśli przeciwnik systemu spróbuje wykonać atak ze znanym tekstem jawnym, będzie dysponował parą (M, E) (lub większą ilością takich par), a jego celem będzie odgadnięcie K.

Wykorzystując wiedzę o strukturze incydencji grafu algebraicznego kryptoanalityk może próbować wyliczyć ścieżkę między wierzchołkami M i E w grafie

oraz odpowiadający tej ścieżce ciąg zmian kolorów wierzchołków, będący hasłem. Każdy wierzchołek posiada q sąsiadów. Zakładając, że procedura szyfrowania nie redukuje hasła poprzez powroty do wierzchołków już odwiedzonych mamy:

- q możliwości wyboru pierwszego sąsiada,
- $q-1$ możliwości wyboru każdego kolejnego sąsiada.

Aby przebadać wszystkie ścieżki długości k (procedura przeszukiwania grafu „w szerz"), należy wykonać $q\ (q-1)^{k-1}$ przejść między wierzchołkami w grafie. Każde przejście między dwoma sąsiednimi wierzchołkami wymaga wyliczenia n współrzędnych nowego wierzchołka, a złożoność tej operacji jest liniowa względem n, czyli wymaga $O(n)$ operacji arytmetycznych. Ostatecznie atak brutalny wymagałby

$$O(n\ q\ (q-1)^{k-1}) = O(n\ q^k),$$

gdzie q możemy uważać za stałe przy ustalonym pierścieniu, jednak całość rośnie wykładniczo względem długości hasła.

Przy hasłach długości większej niż $n/2$ przeszukiwane wierzchołki mogłyby się powtarzać i wtedy można próbować stosować algorytm Dijkstry ([12]) wyszukiwania najkrótszej ścieżki między dwoma ustalonymi wierzchołkami grafu. W zależności od implementacji złożoność czasową algorytmu szacuje się na $O(|V|^2)$, $O(|E|\log|V|)$ lub $O(|E|+|V|\log|V|)$. Ale ilość wierzchołków grafu zgodnie z twierdzeniem 4.3 wynosi

$$|V| = 2\ q^n,$$

natomiast ilość krawędzi każdej składowej spójnej zależy liniowo od $|V|$ ([30]).

Reasumując, niezależnie od implementacji algorytmu Dijkstry, za każdym razem otrzymamy zależność wykładniczą $O(q^n)$, a dodatkowym problemem byłaby pamięć potrzebna na zapisanie wierzchołków odwiedzonych w trakcie działania algorytmu. Zgodnie z przewidywaniami ataki typu „brutalnego" posiadają złożoność wykładniczą względem rozmiaru danych i ich stosowanie w praktyce nie ma sensu. Dodatkowym problemem, z którym musiałby sobie poradzić kryptoanalityk, byłoby wymyślenie sposobu wyszukiwania ścieżki w grafie, gdy graf zostanie „schowany" przez przekształcenia afiniczne, gdy stosujemy algorytm rozszerzony opisany w rozdziale 5.

7.2 Atak metodą linearyzacji

W jaki sposób można próbować zaatakować opisany przez nas system kryptografii symetrycznej zauważył członek naszego zespołu kryptologicznego mgr M.

Klisowski, który pracuje nad algorytmami klucza publicznego, wykorzystującymi grafy algebraiczne ([21]). Zaproponowany przez niego atak wykorzystuje wiedzę o ograniczonym stopniu przekształcenia, które stosujemy oraz tym samym stopniu przekształcenia odwrotnego oraz wiedzę z pozycji [2, 20]. Poniżej przedstawiamy ideę tego ataku.

Przypomnijmy, że przekształcenia typu N_K wykorzystujące rodziny grafów A_n i D_n są przekształceniami 3 stopnia przestrzeni wektorowej $\mathbb{K}^n$ na siebie. Stopień nie ulega zmianie przy rozszerzeniu algorytmu o kroki afiniczne i nie zależy od długości klucza K. Przekształcenie odwrotne N_K^{-1} jest również przekształceniem stopnia 3.

Potraktujmy teraz nasze przekształcenie odwrotne jako wektor funkcji $F = (f_1, f_2, \ldots, f_n)$

$$\begin{aligned} x_1 &= f_1(z_1, z_2, \ldots, z_n) \\ x_2 &= f_2(z_1, z_2, \ldots, z_n) \\ &\vdots \\ x_n &= f_n(z_1, z_2, \ldots, z_n), \end{aligned}$$

gdzie każde f_i składa się z sumy wielomianów zmiennych $z_1, \ldots, z_n$ stopnia co najwyżej 3.

Jeśli teraz wprowadzimy pomocnicze zmienne na każdy możliwy iloczyn postaci

$$z_{ijk} = z_i \; z_j \; z_l,$$

to zmiennych takich otrzymamy dokładnie

$$(n+1)(n+2)(n+3)/6 = d_n$$

co jest ilością $O(n^3)$.

Aby przeprowadzić atak, kryptoanalityk potrzebuje $O(n^3)$ liniowo niezależnych między sobą par (M, E). Posiadając te pary kryptoanalityk może rozważyć każdą funkcję f_i jako funkcję liniową wszystkich zmiennych z_{ijk} i stworzyć układ równań postaci

$$x_i = a_{000} + a_{001} \; z_1 + \ldots + a_{111} \; z_1^3 + a_{112} \; z_1^2 z_2 + \ldots + a_{nnn} \; z_n^3$$

każdorazowo wstawiając w miejsce x_i i-tą składową M, natomiast wszystkie jednomiany maksymalnie trzeciego stopnia od zmiennych z_i należy wyliczyć na podstawie $E = (z_1, z_2, \ldots, z_n)$. W ten sposób otrzymamy układ równań liniowych ze względu na współczynniki a_{ijk}, gdzie $i \leqslant j \leqslant k$ (aby nie powielać tych samych współczynników), który można zapisać macierzowo:

$$\mathbf{x} = \mathbf{A} \; \mathbf{z}, \tag{7.1}$$

gdzie $\mathbf{x}, \mathbf{z}$ są danymi wektorami długości d_n, natomiast $\mathbf{A}$ macierzą kwadratową niewiadomych rozmiaru $d_n \times d_n$.

Taki układ równań liniowych można łatwo rozwiązać metodami numerycznymi (np. metodą eliminacji Gausa). Czas wykonania ogólnej metody Gaussa wynosi $O(m^3)$, gdzie m jest rozmiarem danych. W naszym przypadku $m = O(n^3)$, co oznacza, że całkowity koszt rozwiązania układu równań (7.1) jest rzędu $O(n^9)$.

Jest to czas potrzebny na wyznaczenie wszystkich parametrów dokładnie jednej funkcji f_i (funkcji obliczającej i-tą składową tekstu jawnego na podstawie szyfrogramu). Aby złamać szyfr należy tę procedurę powtórzyć n razy. Ostatecznie otrzymujemy oszacowanie całkowitego kosztu łamania naszego szyfru atakiem typu linearyzacji jako $O(n^{10})$.

Jeśli weźmiemy pod uwagę ilość odpowiednich par (tekst jawny, szyfrogram), jakich potrzebuje kryptoanalityk do dokonania opisanego tu ataku, dochodzimy do wniosku, że nasze algorytmy są podatne na rodzaj ataku *z wybranym tekstem jawnym* przy bardzo dużej ilości tekstów jawnych ($O(n^3)$) oraz możliwości kryptoanalizy w czasie wielomianowym o dosyć wysokim współczynniku.

Naszym zdaniem ataki tego typu nie stanowią w praktyce zagrożenia dla kryptosystemu klucza prywatnego. Natomiast atak ten stanowi problem przy konstrukcji algorytmu klucza publicznego metodą analogiczną do systemu Imai-Matsumoto – poprzez opublikowanie układu równań jako schematu postępowania przy szyfrowaniu kluczem publicznym i zachowaniu w tajemnicy szczegółów trzech przekształceń S, T, N. Znając parametry przekształcenia publicznego, można przecież szyfrować dowolne duże liczby różnych tekstów jawnych i dobierać te teksty według uznania (w szczególności, aby były liniowo niezależne).

7.3 Podniesienie stopnia przekształcenia

Aby poprawić algorytm wykorzystujący rodziny A_n oraz D_n w kontekście ataku typu linearyzacji, należy zwiększyć stopień odwzorowania odpowiadający przejściu między wierzchołkami w grafie. Używany przez nas sposób kolorowania wierzchołków traktował pierwsze współrzędne wierzchołka jako jego kolor, natomiast wybór sąsiada następował przy użyciu operatora (krok $x \xrightarrow{a} y$)

$$N_a(x_1, x_2, \ldots, x_n) = [x_1 + a, y_2, y_3, \ldots, y_n]$$

i analogicznie przy przejściu z punktu na linię nowe $x_1 = y_1 + a$. Ciąg kolorów wierzchołków kodowanych kluczem $K = (a_1, a_2, \ldots, a_k)$ był postaci

$$x_1, x_1 + a_1, x_1 + a_1 + a_2, \ldots, x_1 + \sum_{i=1}^{k} a_i. \tag{7.2}$$

Rozwiązywanie układu równań typu (3.1) polegało na sprowadzeniu każdego równania do postaci (dla ustalenia uwagi przejście z wierzchołka $p = (p_1, p_2, \ldots, p_n) \in P$ na $l = [l_1, l_2, \ldots, l_n] \in L$)

$$l_i = \pm p_i \pm p_1 \; l_{k(i)}$$

lub

$$l_i = \pm p_i \pm l_1 \; p_{k(i)},$$

gdzie funkcja $k(i)$ wybiera, któryś z poprzednich indeksów, a p_1 i l_1 oznaczają pierwsze współrzędne w bieżącym układzie równań i przyjmują wartości wybierane odpowiednio z ciągu (7.2). Stopień odwzorowania definiującego każde l_i można było ograniczyć przez 3 z uwagi na to, że ciąg pierwszych składowych (7.2) zawiera tylko elementy liniowo zależne od x_1, a wyliczenie kolejnej składowej l_i dla grafów z rodzin D_n oraz A_n okazało się zależne od składowej z indeksem 1 oraz jednej lub dwóch składowych o indeksie mniejszym od i, ale większym od 1. Z tego samego powodu mieliśmy problem z jedną z własności "mieszających" Madrygi opisany w rozdziale 5.

Aby zwiększyć stopień przekształceń wyliczających nowe wartości składowych wektorów p i l bez zmieniania równań, definiujących strukturę incydencji w grafie, należy zastosować inny operator wyboru sąsiada. Na przykład można zastosować operator zależny liniowo od dwóch kolejnych elementów hasła i kolor następnego wierzchołka definiować jako:

$$M_{a,b}(x_1) = a \; x_1 + b.$$

Przy stosowaniu tego typu kolorowania traci się niektóre własności, które wpływały korzystnie na nasz kryptosystem symetryczny. Na przykład dla ustalonego $x_1 \in \mathbb{K}$ operator $M_{a,b}(x_1)$ nie jest różnowartościowy w pierścieniu skończonym $\mathbb{K}$ dla zmiennych $a, b \in \mathbb{K}$.

Problem ten wpłynąłby niekorzystnie na nasz symetryczny algorytm i wymagałby dodatkowych założeń na a i b. Jednak w przypadku algorytmu z kluczem publicznym, gdzie upubliczniane są współczynniki pewnego przekształcenia wykorzystującego grafy, złożonego z dwoma przekształceniami afinicznymi wspomniana różnowartościowość nie jest przeszkodą (wybiera się jedno, określone przekształcenie odpowiadające jakiejś ścieżce w grafie, składa z przekształceniami afinicznymi i publikuje w formie wielomianów n zmiennych).

Z drugiej strony przy algorytmach klucza publicznego typu kryptografii wielu zmiennych mamy inne problemy:

1. Stopień przekształcenia szyfrującego powinien być możliwie mały (raczej wielomianowy o niewielkim stopniu).

2. Stopień przekształcenia deszyfrującego powinien być jak największy.

3. Liczba jednomianów w wielomianach przekształcenia szyfrującego, stanowiącego klucz publiczny, powinna być na tyle mała, aby można było efektywnie realizować algorytm szyfrujący. Liczba ta stanowi rozmiar klucza publicznego i zależy od niej czas szyfrowania.

W praktyce należy zbalansować stopień przekształcenia szyfrującego oraz gęstość jednomianów składowych tego przekształcenia.

Podsumowanie

Na początku tej pracy postawiliśmy sobie pewne cele. Uzasadnimy teraz wykonanie każdego z nich.

Zbadanie algorytmów z klasy szyfrów strumieniowych, opartych na aproksymacji drzew, pod kątem szybkości i bezpieczeństwa

Na początek uzasadnimy fakt stworzenia szyfrów strumieniowych. Przede wszystkim szyfry blokowe dzielą tekst otwarty na bloki równego rozmiaru i każdy z nich szyfrują tym samym algorytmem niezależnie od pozostałych bloków. W naszych algorytmach nie ma podziału na bloki, a sposób zaszyfrowania poprzednich znaków wiadomości wpływa na szyfrowanie kolejnego. Mamy tu różnicę w stosunku do klasycznych szyfrów strumieniowych, ponieważ wykonujemy serię takich szyfrowań (każde przejście do następnego wierzchołka grafu wymaga wyliczenia wszystkich współrzędnych) i nie używamy operacji XOR jako dominującego działania.

W rozdziale opisującym własności rodzin grafów $A_n(\mathbb{K})$ i $D_n(\mathbb{K})$ przytoczyliśmy twierdzenia (4.10) oraz (4.11), wg których nasze rodziny grafów przy odpowiednich założeniach w nieskończonej granicy rzutowej dążą do drzewa lub lasu. Dodatkowo talia każdej z rodzin grafów dąży do nieskończoności przy $n \to \infty$, można więc śmiało nazwać te rodziny „aproksymującymi drzewa".

Rozdział 6 w całości poświęcony jest wydajności naszych algorytmów, natomiast ich bezpieczeństwo było badane w różnych miejscach pracy:

- po pierwsze przytaczając twierdzenie (4.7) uzasadniamy sens stosowania przekształceń opartych na naszych grafach algebraicznych (przekształcenia nieliniowe),
- w rozdziale 5 przytaczamy wyniki eksperymetów, badających zmiany szyfrogramu przy jednostkowych zmianach w tekście otwartym lub kluczu; pokazujemy też jak poprawić jedną z tych własności,

- w rozdziale 7 opisujemy dwa różne rodzaje kryptoanalizy naszego systemu i wspominamy o możliwości poprawy tego bezpieczeństwa kosztem wydajności i skomplikowania systemu.

Stworzenie nowych algorytmów w badanej klasie przez modyfikację algorytmów znanych do tej pory

W rozdziale 5 opisaliśmy w jaki sposób stworzyć bardziej bezpieczny algorytm szyfrowania symetrycznego oparty na grafach: poprzez złożenie podstawowego przekształcenia grafowego z dwoma przekształceniami afinicznymi. Z kolei w sekcji 6.4 przedstawiliśmy nowe podejście do algorytmów opartych o rodziny grafów $A_n\mathbb{K}$ i $D_n\mathbb{K}$ wybierając jako pierścień $\mathbb{K}$ pierścień boolowski.

Wybór algorytmów z optymalnymi parametrami

Przekształcenia afiniczne S i T systemu rozszerzonego $S \circ N_K \circ T$ można wybierać na wiele sposobów. W sekcji 5.1 zaproponowaliśmy trzy przekształcenia dostosowane do naszego problemu. Przekształcenia te „wiążą" pierwszą współrzędną tekstu otwartego z wszystkimi pozostałymi i mają liniową względem rozmiaru danych liczbę działań, co uzasadniliśmy w rozdziale dotyczącym wydajności (w sekcji 6.1).

Kolejnym parametrem jaki możemy wybierać jest rozmiar alfabetu danych, a co za tym idzie rozmiar pierścienia $\mathbb{K}$. Zarówno teoretyczne oszacowania złożoności algorytmów, jak i wyniki testów pokazują, że najlepszą prędkość uzyskujemy dla największego rozmiaru pierścienia ($Z_{2^{64}}$).

Porównanie właściwości kryptograficznych nowych algorytmów z poprzednimi (teoretyczne oraz symulacje)

W rozdziale 5 przedstawiliśmy problem, dotyczący podstawowego algorytmu szyfrującego, który został wykryty przy pomocy symulacji komputerowej, a następnie sprawdzony matematycznie. Wprowadziliśmy też rozszerzenie algorytmów, które poprawia dwie własności dotyczące bezpieczeństwa:

- zmiana pojedynczego elementu tekstu otwartego przy ustalonym kluczu prowadzi najczęściej do ponad 90% zmian w szyfrogramie,
- struktura incydencji grafu zostaje ukryta „za" przekształceniem afinicznym i nie można przeszukiwać grafu, korzystając bezpośrednio z równań struktury incydencji.

W rozdziale 6 zaprezentowaliśmy symetryczny szyfr oparty na rodzinach grafów $D_n(\mathbb{K})$ oraz $A_n(\mathbb{K})$ budowanych nad pierścieniami modulo $\mathbb{Z}_{2^s}$ ($s \in \{8, 16, 32, 64\}$)

oraz pierścieniami boolowskimi $\mathbb{B}_{2^s}$. Wcześniejsze implementacje bazowały jedynie na rodzinie $D_n(\mathbb{F}_q)$ nad ciałem skończonym $\mathbb{F}_q$. Opisaliśmy jakie dodatkowe wymagania teoretyczne muszą spełniać nasze nowe szyfry, aby były wiarygodne, a także porównaliśmy prędkość działania naszych implementacji z implementacjami historycznymi oraz dwoma ogólnie znanymi szybkimi szyframi.

Teza rozprawy: zaproponowane przez nas algorytmy klucza prywatnego oparte o grafy algebraiczne o dużej talii mają lepsze od znanych do tej pory algorytmów tej samej klasy właściwości kryptograficzne

Uważamy, że analiza i porównanie naszych nowych algorytmów z algorytmami, które stosowano wcześniej są jednoznacznie korzystne dla naszych algorytmów. Nasz system jest zarówno bezpieczniejszy, jak i szybszy. Wybrane algorytmy (dla $\mathbb{K} = \mathbb{Z}_{2^{64}}$) są nawet znacznie szybsze, przy tych samych parametrach, od znanych ze swojej szybkości standardowych szyfrów symetrycznych RC4 i DES.

Na koniec ryzykujemy stwierdzenie, że wprowadzone przez nas szyfry symetryczne mogą być z powodzeniem stosowane w praktyce.

Dodatek A: Oryginalna konstrukcja rodziny grafów $D_n(\mathbb{F})$

W tej części zaprezentujemy oryginalną konstrukcję matematyczną rodziny grafów D_n, zgodnie z koncepcją profesora Ustimenko.

Zacznijmy od zdefiniowania macierzy a_{ij} Cartana następującej postaci:

$$\begin{bmatrix} 2 & -2 \\ -2 & 2 \end{bmatrix}.$$

Następnie wybierzmy dwa bazowe parametry formalne α_1, α_2 i w tej bazie zbudujmy kratę postaci

$$k_1\ \alpha_1 + k_2\ \alpha_2, \quad k_1, k_2 \in \mathbb{Z}.$$

Wprowadźmy teraz odwzorowania liniowe

$$r_i(\alpha_j) = \alpha_j - a_{ij}\ \alpha_i.$$

Symbol r pochodzi od angielskiego *reflection*, ponieważ przekształcenia te "odbijają" jak w lustrze elementy o tym samym indeksie $r_i(\alpha_i) = -\alpha_i$.

Następnie zdefiniujmy grupę wygenerowaną przez te odwzorowania z operacją złożenia odwzorowań:

$$W =< r_1, r_2 > .$$

Zauważmy, że jeśli jako e oznaczymy element neutralny w grupie W (odwzorowanie identycznościowe), to

$$r_1 \circ r_1 = e = r_2 \circ r_2.$$

Do grupy W należą w takim razie odwzorowania:

$$e, r_1, r_2, r_1 \circ r_2, r_2 \circ r_1, r_1 \circ r_2 \circ r_1, r_2 \circ r_1 \circ r_2, \ldots$$

Spróbujmy teraz oddziaływać wszystkimi elementami grupy W na elementy α_1 oraz α_2. Otrzymamy wtedy

$$
\begin{array}{ll}
r_1(\alpha_1) = -\alpha_1, & r_1(\alpha_2) = 2\alpha_1 + \alpha_2, \\
r_2(\alpha_1) = \alpha_1 + 2\alpha_2, & r_2(\alpha_2) = -\alpha_2, \\
r_1 \circ r_2(\alpha_1) = r_2(-\alpha_1) = -\alpha_1 - 2\alpha_2, & r_1 \circ r_2(\alpha_2) = r_2(\alpha_2) + 2r_2(\alpha_1) = 2\alpha_1 + 3\alpha_2, \\
r_2 \circ r_1(\alpha_1) = 3\alpha_1 + 2\alpha_2, & r_2 \circ r_1(\alpha_2) = -2\alpha_1 - \alpha_2, \\
r_1 \circ r_2 \circ r_1(\alpha_1) = -3\alpha_1 - 2\alpha_2, & r_1 \circ r_2 \circ r_1(\alpha_2) = 4\alpha_1 + 3\alpha_2 \\
r_2 \circ r_1 \circ r_2(\alpha_1) = 3\alpha_1 + 3\alpha_2, & r_2 \circ r_1 \circ r_2(\alpha_1) = -2\alpha_1 - 3\alpha_2, \\
\vdots & \vdots
\end{array}
$$

Zauważmy, że wszystkie te wyniki są postaci:

$$
\begin{array}{l}
\pm(k\,\alpha_1 + (k+1)\,\alpha_2) \\
\pm(k\,\alpha_2 + (k+1)\,\alpha_1)
\end{array}
\qquad k \in \mathbb{Z}.
$$

Każdy element tej postaci z dodatnimi parametrami k nazywa się *pierwiastkiem rzeczywistym dodatnim*, a z ujemnymi *pierwiastkiem rzeczywistym ujemnym*.

Do dalszych rozważań wybieramy system wszystkich pierwiastków rzeczywistych dodatnich (oznaczamy go φ^+) i dołączamy do niego zbiór pierwiastków urojonych postaci

$$k(\alpha_1 + \alpha_2), \quad k \in \mathbb{N}.$$

Rozpatrzmy teraz algebrę liniową postaci:

$$\sum \lambda_\alpha \cdot \alpha, \quad \lambda_\alpha \in \mathbb{K}, \alpha \in \varphi^+$$

Ustalmy dwa elementy bazowe α_1 i α_2 i zdefiniujmy operację multiplykatywną $[\cdot,\cdot]$ na elementach przestrzeni:

$$[\alpha, \beta] = \begin{cases} \alpha + \beta, & \text{gdy } \alpha + \beta \in \varphi^+ \\ 0, & \text{gdy } \alpha + \beta \in \varphi^- \end{cases}$$

Naszą wprowadzoną przestrzeń liniową możemy podzielić na dwie podprzestrzenie:

$$
\begin{aligned}
L_1 &= \langle \alpha_1, \alpha_1 + \alpha_2, 2\alpha_1 + \alpha_2, 2\alpha_1 + 2\alpha_2, 3\alpha_1 + 2\alpha_2, 3\alpha_1 + 3\alpha_2, \ldots \rangle \\
&= \langle \beta \mid \beta \in \varphi^+ \setminus \{\alpha_2\} \rangle \\
L_2 &= \langle \alpha_2, \alpha_1 + \alpha_2, \alpha_1 + 2\alpha_2, 2\alpha_1 + \alpha_2, 2\alpha_1 + 3\alpha_2, 3\alpha_1 + 3\alpha_2, \ldots \rangle \\
&= \langle \beta \mid \beta \in \varphi^+ \setminus \{\alpha_1\} \rangle.
\end{aligned}
$$

Elementy L_1 nazywamy punktami, a elementy L_2 liniami.

Relację incydencji między tymi elementami definiujemy następująco:

$$\bar{\mathbf{x}}\, I\, \bar{\mathbf{y}} \iff (\bar{\mathbf{x}} - \bar{\mathbf{y}})\Big|_{\varphi^+ - \alpha_1 - \alpha_2} = [\bar{\mathbf{x}}, \bar{\mathbf{y}}], \qquad \bar{\mathbf{x}} \in L_1, \bar{\mathbf{y}} \in L_2.$$

Ostatnia równość jest przyczyną takich, a nie innych równań definiujących strukturę incydencji we wprowadzonej rodzinie grafów $D_n(\mathbb{K})$, a indeksy przy współczynnikach x oraz y biorą się ze współczynników przez jakie mnożymy α_1 oraz α_2, w kolejnych elementach podprzestrzeni L_1 i L_2. Na przykład x_{22} i y_{22} odpowiadają elementowi $(2\alpha_1+2\alpha_2)$. Symbolu „prim" używamy do rozróżnienia sytuacji, w której powstają współczynniki $(i\alpha_1 + i\alpha_2)$, ponieważ może to być wynikiem operacji $[i\alpha_1 + (i-1)\alpha_2), \alpha_2]$ lub $[(i-1)\alpha_1 + i\alpha_2, \alpha_1]$.

Dodatek B: Fragmenty kodu implementacji w języku C#

```
public enum StepNr
{
    Even,
    Odd
}

public enum CryptoModes
{
    None,
    Encrypt,
    Decrypt
}

     public interface ICryptoAlgorithm4Byte
{
    void Encrypt(ref uint[] text, uint[] password);

    void Decrypt(ref uint[] text, uint[] password);
}

public interface ICryptoStep4Byte
{
    void Encrypt(ref uint[] text, uint passElem);

    void Decrypt(ref uint[] text, uint passElem);
}

public interface IPasswordChanger
{
    byte[] ChangeBytes(byte[] password);
    ushort[] ChangeShorts(ushort[] password);
    uint[] ChangeInts(uint[] password);
    ulong[] ChangeLongs(ulong[] password);
}
```

```
 public class EvenOddGraphAlgorithm4B :
    ICryptoAlgorithm4Byte
{
    private Type algorithmType;
    private ConstructorInfo constrInfo;

    public EvenOddGraphAlgorithm4B(Type algorithmType)
    {
        this.algorithmType = algorithmType;
        Type[] argTypes = new Type[2];
        argTypes[0] = typeof(StepNr);
        argTypes[1] = typeof (int);
        constrInfo = algorithmType.GetConstructor(argTypes)
            ;
        if (!typeof(ICryptoStep4Byte).IsAssignableFrom(
            algorithmType))
            throw new ArgumentException("not␣
                ICryptoStep4Byte␣instance␣arg␣for␣
                EvenOddGraphAlg");
    }

    public void Encrypt(ref uint[] text, uint[] password)
    {
        Object[] constrParams = new object[] { StepNr.Even,
            text.Length };
        ICryptoStep4Byte oneStepAlg =
            (ICryptoStep4Byte) constrInfo.Invoke(
                constrParams);
        for (int i = 0; i < password.Length; i++)
        {
            oneStepAlg.Encrypt(ref text, password[i]);
        }
    }

    public void Decrypt(ref uint[] text, uint[] password)
    {
        Object[] constrParams;
        ICryptoStep4Byte oneStepAlg = null;
        Boolean passLengthIsEven = password.Length % 2 ==
            1;
        if (passLengthIsEven)
            constrParams = new object[] { StepNr.Even, text
                .Length };
        else
            constrParams = new object[] { StepNr.Odd, text.
                Length };
        oneStepAlg = (ICryptoStep4Byte) constrInfo.Invoke(
            constrParams);
        for (int i = password.Length - 1; i >= 0; i--)
```

```
        {
            oneStepAlg.Decrypt(ref text, password[i]);
        }
    }
}

    public class Dnk1StepAlgorithm4Byte : ICryptoStep4Byte
{

    //Do encryption on 4+ elements tables!!!
    private StepNr stepNr; //after each step change to
       other
    private uint[] codedInts;


    public Dnk1StepAlgorithm4Byte(StepNr stepParity, int
       tableLength)
    {
        stepNr = stepParity;
        codedInts = new uint[tableLength];
    }

    public void Encrypt(ref uint[] text, uint passElem)
    {
        if (stepNr == StepNr.Even)
        {
            encryptEvenStep(ref text, passElem);
        }
        else
            encryptOddStep(ref text, passElem);
        changeStepParity();
    }

    private void changeStepParity()
    { //change step parity
        if (stepNr == StepNr.Even)
            stepNr = StepNr.Odd;
        else stepNr = StepNr.Even;
    }

    public void Decrypt(ref uint[] text, uint passElem)
    {
        if (stepNr == StepNr.Even)
        {
            decryptEvenStep(ref text, passElem);
        }
        else
            decryptOddStep(ref text, passElem);
        changeStepParity();
```

```
}

private void encryptEvenStep(ref uint[] text, uint
    passElem)
{   // xs ==> ys
    uint x0 = text[0];
    uint y0 = (uint)((x0 + passElem) & 0xFFFFFFFF);
    int uintsLength = codedInts.Length;
    int j;
    codedInts[0] = y0;
    codedInts[1] = (uint)((text[1] + x0 * y0) & 0
        xFFFFFFFF);
    codedInts[2] = (uint)((text[2] + x0 * codedInts[1])
         & 0xFFFFFFFF);
    codedInts[3] = (uint)((text[3] + text[1] * y0) & 0
        xFFFFFFFF);
    for (j = 4; j < uintsLength - 3; j += 4)
    {
        codedInts[j] = (uint)((text[j] + y0 * text[j -
            2]) & 0xFFFFFFFF);
        codedInts[j + 1] = (uint)((text[j + 1] + x0 *
            codedInts[j - 1]) & 0xFFFFFFFF);
        codedInts[j + 2] = (uint)((text[j + 2] + x0 *
            codedInts[j]) & 0xFFFFFFFF);
        codedInts[j + 3] = (uint)((text[j + 3] + y0 *
            text[j + 1]) & 0xFFFFFFFF);
    }
    //last few elements
    if (j < uintsLength) codedInts[j] = (uint)((text[j]
         + y0 * text[j - 2]) & 0xFFFFFFFF);
    if (j + 1 < uintsLength) codedInts[j + 1] = (uint)
        ((text[j + 1] + x0 * codedInts[j - 1]) & 0
        xFFFFFFFF);
    if (j + 2 < uintsLength) codedInts[j + 2] = (uint)
        ((text[j + 2] + x0 * codedInts[j]) & 0xFFFFFFFF)
        ;
    if (j + 3 < uintsLength) codedInts[j + 3] = (uint)
        ((text[j + 3] + y0 * text[j + 1]) & 0xFFFFFFFF);

    swapCodedWithPlain(ref text);
}

private void swapCodedWithPlain(ref uint[] plain)
{
    uint[] temp = codedInts;
    codedInts = plain;
    plain = temp;
}
```

```
private void encryptOddStep(ref uint[] text, uint
    passElem)
{
    //..... analogicznie jak encryptEvenStep.....
}

private void decryptEvenStep(ref uint[] text, uint
    passElem)
{
     // ys ==> xs
     uint y0 = text[0];
     uint x0 = (uint)((y0 - passElem) & 0xFFFFFFFF);
     int uintsLength = codedInts.Length;
     int j;
     codedInts[0] = x0;
     codedInts[1] = (uint)((text[1] - x0 * y0) & 0
        xFFFFFFFF);
     codedInts[2] = (uint)((text[2] - x0 * text[1]) & 0
        xFFFFFFFF);
     codedInts[3] = (uint)((text[3] - codedInts[1] * y0)
         & 0xFFFFFFFF);
     for (j = 4; j < uintsLength - 3; j += 4)
     {
         codedInts[j] = (uint)((text[j] - y0 * codedInts
            [j - 2]) & 0xFFFFFFFF);
         codedInts[j + 1] = (uint)((text[j + 1] - x0 *
            text[j - 1]) & 0xFFFFFFFF);
         codedInts[j + 2] = (uint)((text[j + 2] - x0 *
            text[j]) & 0xFFFFFFFF);
         codedInts[j + 3] = (uint)((text[j + 3] - y0 *
            codedInts[j + 1]) & 0xFFFFFFFF);
     }//for j
     //few last elements
     if (j < uintsLength) codedInts[j] = (uint)((text[j]
         - y0 * codedInts[j - 2]) & 0xFFFFFFFF);
     if (j + 1 < uintsLength) codedInts[j + 1] = (uint)
        ((text[j + 1] - x0 * text[j - 1]) & 0xFFFFFFFF);
     if (j + 2 < uintsLength) codedInts[j + 2] = (uint)
        ((text[j + 2] - x0 * text[j]) & 0xFFFFFFFF);
     if (j + 3 < uintsLength) codedInts[j + 3] = (uint)
        ((text[j + 3] - y0 * codedInts[j + 1]) & 0
        xFFFFFFFF);

     swapCodedWithPlain(ref text);
}

private void decryptOddStep(ref uint[] text, uint
    passElem)
{
```

```
        //..... analogicznie jak decryptEvenStep.....
    }

}

class LastBitFixedPasswordChanger: IPasswordChanger
{
              public uint[] ChangeInts(uint[] password)
    {
        // pass[i]+pass[i+1] have to be odd  ->  | (leng%2)
        byte offset = 0;
        uint leng = 0;
        uint temp = 0;
        int newLength = (int)(password.Length + password.
           Length / 31);
        if (password.Length % 31 != 0)
            newLength++;
        uint[] result = new uint[newLength];
        for (int i = 0; i < password.Length; i++)
        {
            temp = password[i];
            result[leng] = (uint)(result[leng] << (32 -
               offset)
                                  | ((temp >> (offset + 1)
                                     ) << 1)
                                  | (leng % 2));
            leng++;
            offset = (byte)((offset + 1) % 32);
            result[leng] = (uint)((temp % (2 << offset)));
        }
        result[leng] = (uint)(((result[leng] << (31 -
           offset)) << 1) | (leng % 2));
        return result;
    }
    public  byte[] ChangeBytes(byte[] password)
    {
         //....analogicznie
    }
    public  ushort[] ChangeShorts(ushort[] password)
    {
         //....analogicznie
    }
    public ulong[] ChangeLongs(ulong[] password)
    {
         //....analogicznie
    }
}

     public class DecimalNumbersConverter
```

```
{
    public static uint[] Convert2uints(byte[] openTextBytes
       )
    {
        uint[] uintArray = new uint[uintLengthFrom(
           openTextBytes.Length)];
        int j = 0;
        int i = 0;
        int bytesDiv4 = openTextBytes.Length / 4;
        int bytesMod4 = openTextBytes.Length % 4;
        for (; i < bytesDiv4; i += 4, j++)
        {
            uintArray[j] = BitConverter.ToUInt32(
               openTextBytes, i);
        }
        //last element with 1-3 bytes in other array
        if (bytesMod4 > 0)
        {
            byte[] tempBytes = new byte[4];
            int k = 0;
            for (; k < bytesMod4; k++)
                tempBytes[k] = openTextBytes[i + k];

            for (; k < 4; k++)
                tempBytes[k] = 0;
            uintArray[j] = BitConverter.ToUInt32(tempBytes,
                0);
        }
        return uintArray;
    }

    public static ushort[] Convert2ushorts(byte[]
       openTextBytes)
    {
        //..... analogicznie .....
    }
          public static ulong[] Convert2Ulongs(byte[]
             openTextBytes)
    {
        //..... analogicznie .....
    }
    internal static int uLongLengthFrom(int byteLength)
    {
        return (byteLength + 7) / 8;
    }
    internal static int uintLengthFrom(int byteLength)
    {
        return (byteLength + 3) / 4;
    }
```

```
    internal static int ushortLengthFrom(int byteLength)
    {
        return (byteLength + 1) / 2;
    }
}
```

Spis rysunków

Spis tabel

Bibliografia

[1] G. Agnarsson, R. Greenlaw, *Grapth Theory: Modeling, Applications, and Algorithms*, Pearson Education, 2007.

[2] G. Bard, *Algebraic Cryptanalysis*, Springer, 2009.

[3] N. Biggs, *Algebraic Graph Theory*, (2nd ed), Cambridge, University Press, 1993.

[4] N. Biggs, *Graphs with large girth*, Ars Combinatoria, 25C (1988), 73–80.

[5] B. Bollobás, *Extremal Graph Theory*, Academic Press, London, 1978.

[6] B. Bollobás. *Random Graphs* , Cambridge University Press, 2001.

[7] J. A. Bondy, M. Simonovits. *Cycles of even length in graphs*, J. Combin.Theory, Ser. B, 16 (1974) 87-105.

[8] N. Bourbaki, *Elements of mathematics - Algebra, chapters 1-3*. Springer, 1989.

[9] J. A. Buchmann. *Wprowadzenie do kryptografii*. Wydawnictwo Naukowe PWN SA, 2006.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, The MIT Press, 3rd ed., 2009.

[11] W. Diffie, M. Hellman, *New Directions in Cryptography*, IEEE Transactions on Information Theory, IT-22: 644–654.

[12] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik, 1 (1959), 269–271.

[13] J. Ding, J. Gower, D. Schmidt, *Multivariate public key cryptosystems*, Springer, 2006.

[14] T. ElGamal, *A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, IEEE Transactions on Information Theory, v. IT-31, n. 4, 1985, 469–472.

[15] P. Erdős, A. Rényi, and V. T. Sós. *On a problem of graph theory.* Studia Sci. Math. Hungar., 1(1966), 215–235.

[16] P. Erdős, M. Simonovits, *Compactness results in extremal graph theory*, Combinatorica, **2**(3) (1982), 275–288.

[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design patterns, software engineering, object-oriented programming*, Addison-Wesley, 1994.

[18] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[19] C. Godsil, G. Royle, *Algebraic Graph Theory*, Graduate Texts in Mathematics 207, Springer-Verlag, New York, 2001.

[20] A. Kipnis, A. Shamir. *Cryptanalysis of the public key cryptosystem by relinearization.* M. Wiener, redaktor, Advances in Cryptology — CRYPTO'99, vol. 1666 serii *Lecture Notes in Computer Science*, 19–30.

[21] M. Klisowski, V. Ustimenko. *On the implementation of public keys algorithms based on algebraic graphs over finite commutative rings.* IMCSIT, 2010, 303–308.

[22] N. Koblitz, *Algebraic aspects of Cryptography*, v. 3, Springer, 1998.

[23] N. Koblitz, *A Course in Number Theory and Cryptography*, Springer-Verlag, 2nd ed., 1994.

[24] C. Kościelny, M. Kurkowski, M. Srebrny, *Kryptografia, teoretyczne podstawy i praktyczne zastosowania*, Wydawnictwo PJWSTK, 2009.

[25] S. J. Kotorowicz, V. Ustimenko, *On the implementation of cryptoalgorithms based on algebraic graphs over some commutative rings*, Condens. Matter Phys. 11 (2008), no. 2(54), 347–360.

[26] S. J. Kotorowicz, V. Ustimenko, U. Romańczuk, *On the implementation of stream ciphers based on a new family of algebraic graphs*, IEEE Computer Society Press, Proceedings of the Conference CANA, FedSCIS, 485–490.

[27] S. Lang, *Algebra*, third ed., Addison-Wesley Publishing Co., Reading, Mass., 1993.

[28] F.Lazebnik, V.Ustimenko, *Some Algebraic Constructions of Dense Graphs of Large Girth and of Large Size*, DIMACS series in Discrete Mathematics and Theoretical Computer Science, V. 10(1993), 75–93.

[29] F.Lazebnik, V.Ustimenko, *Explicit construction of graphs with an arbitrary large girth and of large size*, Discrete Appl. Math., 60, 1995, 275–284.

[30] F. Lazebnik, V. Ustimenko, A. J. Woldar, *New Series of Dense Graphs of High Girth*, Bull (New Series) of AMS, v.32, N1, (1995), 73–79.

[31] R. Lidl, H. Niederreiter. *Finite Fields*. Numer 20 serii Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1997.

[32] W. E. Madryga, *A High Performance Encryption Algorithm*, Computer Security: A Global Challenge, elsevier Science Publishers, 1984, 557–570.

[33] T. Matsumoto, H. Imai, *Public quadratic polynomial-tuples for efficient signature verification and message-encryption*. Eurocrypt '88, Springer-Verlag (1988), 419–453.

[34] B. Mortimer, *Permutation groups containing affine group of the same degree*, J. London Math. Soc., 1971, 15, N3, 445–455.

[35] R. Ore, *Graph Theory*, Wiley, London, 1971.

[36] J. Patarin, *Cryptanalysis of the Matsumoto and Imai public key scheme of Eurocrypt '88*, Advances in Cryptology-Crypto '95, Springer-Verlag, 248–261.

[37] J. Patarin, *Asymmetric cryptography with a hidden monomial*, Advances in Cryptology- Crypto '96, Springer-Verlag, 45–60.

[38] J. Patarin, N. T. Courtois, L. Goubin, *FLASH, a fast multivariate signature algorithm*, CT-RSA01, '2001, LNCS Vol. 2020, 298–307.

[39] M. J. Robshaw, *Stream Ciphers Technical Report TR-701*, v. 2.0, RSA Laboratories, 1995.

[40] U. Romańczuk, V. Ustimenko, *On the key exchange with new cubical maps based on graphs*, Annales UMCS Informatica, 2011-4, No 11, 11–19.

[41] U. Romańczuk, V. Ustimenko, *On Dynamical Systems of Large Girth and their Applications to Multivariate Cryptography* , rozdział w książce "Artificial intelligence Evolutionary, Computing and Metaheuristics, In the footsteps of Allan Turing", seria: Studies in Computational Intelligence, vol. 427, Springer, 2013, 231–256.

[42] B. Schneier, *Applied Cryptography*, John Wiley & Sons, 1994.

[43] C. E. Shannon. *Communication theory of secrecy systems*. Bell Systems Technical Journal, 28, 1949, 656–715.

[44] A. Touzene, V. Ustimenko, *CRYPTALL – a System to Encrypt All types of Data*, Notices of Kiev Mohyla Academy, 2004: 57.

[45] V. Ustimenko, *Coordinatisation of Trees and their Quotients*, Voronoj's Impact on Modern Science, Kiev, Institute of Mathematics, 1998, vol. 2, 125–152.

[46] V. Ustimenko, *CRYPTIM: Graphs as Tools for Symmetric Encryption*, Lecture Notes in Computer Science, Springer, 2001, v. 2227, 278–287.

[47] V. Ustimenko, *Linguistic Dynamical Systems, Graphs of Large Girth and Cryptography*, Journal of Mathematical Sciences, Springer, vol.140, N3 (2007), 412–434.

[48] V. Ustimenko, *On extremal graph theory and symbolic computations*, Reports of the National Academy of Science of Ukraine, 2013, No 2, 42–48.

[49] B.L. van der Waerden, *Algebra. I*, Springer-Verlag, New York, 1991.

[50] R. Wenger, *Extremal graphs with no* C^4, C^6,or C^{10}*'s*, J. of Combinatorial Theory, Series **B 52**, 1991, 113–116.

[51] A. Wróblewska, *On some properties of graph based public keys*, Albanian Journal of Mathematics, Vol. 2, No 3, 2008, 229–234.