Politechnika Świętokrzyska

Wydział Elektrotechniki, Automatyki i Informatyki

Maksymilian Sowula

Paweł Marek

Jakub Szczur

Daniel Cieślak

Numer grupy dziekańskiej: 1ID21A

Kryptografia

Złożone Struktury Danych

1. Wprowadzenie do kryptografii

Kryptografia jest jedną z kluczowych dziedzin współczesnej informatyki, stanowiąc fundament bezpieczeństwa danych w systemach teleinformatycznych. Jej głównym celem jest ochrona informacji przed nieuprawnionym dostępem poprzez stosowanie metod matematycznych, algorytmów szyfrujących oraz struktur logicznych, które zapewniają poufność, integralność, autentyczność i niezaprzeczalność danych.

Rozwój kryptografii jest nierozerwalnie związany z postępem technologicznym — od prostych systemów podstawieniowych używanych w starożytności, przez złożone maszyny elektromechaniczne XX wieku, aż po współczesne algorytmy oparte na zaawansowanej matematyce i teorii liczb. Obecnie kryptografia nie ogranicza się jedynie do szyfrowania wiadomości, lecz obejmuje również takie zagadnienia jak podpisy cyfrowe, uwierzytelnianie, zarządzanie tożsamością, bezpieczne protokoły komunikacyjne czy systemy kryptowalut.

W niniejszym rozdziale przedstawiono podstawy teoretyczne kryptografii, jej historię, kluczowe pojęcia oraz główne cele, jakie realizuje w kontekście bezpieczeństwa informacji.

1.1. Historia i rozwój

Historia kryptografii sięga tysięcy lat wstecz. Pierwsze metody szyfrowania miały charakter manualny i wykorzystywały proste mechanizmy zamiany znaków. W starożytnym Egipcie stosowano nietypowe hieroglify w celu ukrycia znaczenia wiadomości. W starożytnej Grecji używano skytale - drewnianego walca, wokół którego nawijano pergamin z zapisaną wiadomością; dopiero walec o odpowiedniej średnicy pozwalał odczytać tekst.

Kryptografia w starożytnym Rzymie została rozwinięta przez Juliusza Cezara, który stosował tzw. szyfr Cezara - przesunięcie liter alfabetu o stałą wartość. Była to jedna z pierwszych form szyfru podstawieniowego, której celem było utrudnienie odczytania wiadomości przez niepowołane osoby.

W kolejnych wiekach metody szyfrowania ewoluowały. W renesansie włoski uczony Leon Battista Alberti opracował polialfabetyczny szyfr podstawieniowy, w którym stosowano kilka różnych alfabetów w jednej wiadomości. W XVII wieku Blaise de Vigenère zaproponował tzw. szyfr Vigenère'a, uznawany długo za „niezłamywalny", aż do XIX wieku, gdy Charles Babbage i Friedrich Kasiski opracowali metody jego skutecznego łamania.

XX wiek przyniósł ogromny przełom w kryptografii za sprawą maszyn szyfrujących. Najbardziej znaną z nich była Enigma, używana przez armię niemiecką podczas II wojny światowej. Złamanie jej kodu przez zespół kierowany przez Alana Turinga i polskich kryptologów (Rejewskiego, Różyckiego i Zygalskiego) uznawane jest za jedno z najważniejszych osiągnięć kryptografii i punkt zwrotny w historii informatyki.

Wraz z rozwojem komputerów w drugiej połowie XX wieku kryptografia przeszła z poziomu mechanicznego na matematyczny. W latach 70. opracowano pierwsze algorytmy kryptografii symetrycznej, takie jak DES (Data Encryption Standard), a w 1976 roku Whitfield Diffie i Martin Hellman zaproponowali koncepcję kryptografii asymetrycznej - przełomowy model wykorzystujący dwa różne klucze: publiczny i prywatny. Kilka lat później powstał jeden z najpopularniejszych algorytmów asymetrycznych — RSA, którego bezpieczeństwo opiera się na trudności faktoryzacji dużych liczb pierwszych.

Współcześnie kryptografia stała się nieodłącznym elementem infrastruktury cyfrowej. Stosowana jest w protokołach komunikacyjnych takich jak SSL/TLS, w systemach podpisu elektronicznego, w szyfrowaniu danych na dyskach (np. BitLocker, VeraCrypt), a także w technologiach blockchain, które bazują na kryptograficznych funkcjach skrótu (np. SHA-256). Obecny kierunek rozwoju kryptografii zmierza ku kryptografii postkwantowej, odpornej na potencjalne zagrożenia związane z komputerami kwantowymi.

1.2. Podstawowe pojęcia

Kryptografia, mimo że opiera się na złożonych teoriach matematycznych, można ją opisać poprzez kilka kluczowych pojęć, które tworzą jej fundament.

Szyfrowanie (ang. encryption) - proces przekształcania czytelnej informacji (tzw. tekst jawny, plaintext) w formę nieczytelną dla osób nieuprawnionych (tzw. szyfrogram, ciphertext). Operacja ta odbywa się z użyciem określonego algorytmu i klucza kryptograficznego.

Deszyfrowanie (ang. decryption) - proces odwrotny do szyfrowania, pozwalający przy użyciu odpowiedniego klucza odzyskać oryginalną treść wiadomości.

Klucz kryptograficzny (ang. cryptographic key) - ciąg znaków lub liczb, który determinuje sposób działania algorytmu szyfrującego. W kryptografii symetrycznej ten sam

klucz służy zarówno do szyfrowania, jak i deszyfrowania, natomiast w asymetrycznej stosuje się dwa różne klucze: publiczny i prywatny.

Algorytm kryptograficzny (ang. cryptographic algorithm) - zestaw reguł matematycznych definiujących sposób przekształcania danych. Przykładami nowoczesnych algorytmów są AES, RSA, ECC, SHA czy ChaCha20.

Kryptoanaliza (ang. cryptanalysis) - nauka zajmująca się analizą systemów kryptograficznych w celu ich złamania, czyli odszyfrowania danych bez znajomości klucza. Kryptoanaliza pełni ważną rolę w testowaniu bezpieczeństwa algorytmów.

Funkcja skrótu (ang. hash function) - funkcja matematyczna przekształcająca dowolnie długi ciąg danych w wynik o stałej długości (tzw. skrót, hash). Funkcje skrótu są szeroko stosowane w weryfikacji integralności danych, systemach podpisów cyfrowych oraz przechowywaniu haseł.

Podpis cyfrowy (ang. digital signature) - kryptograficzny mechanizm pozwalający potwierdzić autentyczność i integralność danych oraz tożsamość nadawcy.

W praktyce systemy kryptograficzne łączą te elementy w złożone struktury bezpieczeństwa, wykorzystywane w aplikacjach sieciowych, bankowości elektronicznej, komunikatorach szyfrowanych (np. Signal, WhatsApp) czy systemach zarządzania tożsamością (PKI - Public Key Infrastructure).

1.3. Podstawowe cele

Kryptografia realizuje szereg celów, które łącznie zapewniają bezpieczeństwo informacji. Do najważniejszych należą:

1. Poufność (Confidentiality) - gwarantuje, że dostęp do informacji mają jedynie uprawnione osoby. W praktyce poufność osiąga się poprzez szyfrowanie danych, zarówno podczas transmisji (np. protokoły HTTPS, VPN), jak i przechowywania (np. szyfrowanie dysków).
2. Integralność (Integrity) - zapewnia, że dane nie zostały zmodyfikowane w sposób nieautoryzowany. Weryfikację integralności realizuje się m.in. przy pomocy funkcji skrótu (np. SHA-256) lub kodów uwierzytelniających wiadomość (MAC - Message Authentication Code).

3. Uwierzytelnianie (Authentication) - umożliwia potwierdzenie tożsamości nadawcy lub odbiorcy danych. Mechanizmy uwierzytelniania wykorzystywane są m.in. w logowaniu użytkowników, systemach bankowości internetowej oraz podczas wymiany kluczy w protokołach komunikacyjnych.
4. Niezaprzeczalność (Non-repudiation) - uniemożliwia nadawcy wyparcie się autoryzowanego działania, np. wysłania wiadomości czy złożenia podpisu elektronicznego. W praktyce realizowana jest poprzez podpisy cyfrowe oparte na kryptografii asymetrycznej.

Wszystkie te elementy stanowią spójną strukturę bezpieczeństwa - żaden z nich nie gwarantuje pełnej ochrony samodzielnie. Dopiero ich wspólne zastosowanie pozwala budować systemy odporne na ataki, spełniające wymagania współczesnych standardów bezpieczeństwa, takich jak ISO/IEC 27001 czy NIST SP 800-57.

1.4. Klasyfikacja algorytmów kryptograficznych

Algorytmy kryptograficzne można podzielić według różnych kryteriów, jednak najczęściej stosowanym jest podział ze względu na sposób wykorzystania kluczy. Wyróżnia się tu trzy główne klasy: kryptografię klasyczną, kryptografię symetryczną oraz kryptografię asymetryczną. Każda z tych grup charakteryzuje się innymi właściwościami, poziomem bezpieczeństwa, a także zastosowaniami praktycznymi.

Klasyczna kryptografia obejmuje najstarsze metody szyfrowania, w których przekształcenia dokonywano ręcznie lub przy użyciu prostych urządzeń mechanicznych. Wraz z rozwojem informatyki, pojawiły się algorytmy symetryczne - wykorzystujące ten sam klucz do szyfrowania i deszyfrowania danych - oraz algorytmy asymetryczne, oparte na dwóch różnych, lecz matematycznie powiązanych kluczach.

Podział ten jest fundamentalny dla zrozumienia współczesnych systemów bezpieczeństwa, gdyż każda z metod posiada unikalne zalety i ograniczenia, a w praktyce często stosuje się ich kombinację w tzw. systemach hybrydowych.

1.4.1. Kryptografia klasyczna

Kryptografia klasyczna stanowi historyczny fundament współczesnych metod szyfrowania. Opiera się głównie na operacjach podstawieniowych i przestawieniowych

wykonywanych na znakach alfabetu. Choć obecnie ma głównie znaczenie edukacyjne i historyczne, jej koncepcje i mechanizmy stanowiły podstawę do opracowania współczesnych algorytmów.

Szyfry podstawieniowe polegają na zastępowaniu znaków tekstu jawnego innymi znakami według określonego klucza. Najprostszym przykładem jest wspomniany szyfr Cezara, w którym każda litera przesuwana jest o stałą liczbę pozycji w alfabecie. Bardziej zaawansowanym wariantem jest szyfr Vigenère'a, wykorzystujący wielokrotne alfabety podstawieniowe, dzięki czemu trudniej go złamać metodami statystycznymi.

Szyfry przestawieniowe z kolei zmieniają kolejność znaków w wiadomości bez modyfikacji samych symboli. Przykładem może być szyfr kolumnowy, w którym litery zapisywane są w tabeli według określonego klucza, a następnie odczytywane kolumnami w ustalonej kolejności.

W XX wieku klasyczne systemy szyfrowania zostały rozwinięte w kierunku mechanicznych maszyn szyfrujących, takich jak Enigma, Lorenz czy Purple. Zastosowanie elementów elektromechanicznych umożliwiło realizację bardziej złożonych przekształceń, zwiększając odporność szyfrów na analizę.

Choć kryptografia klasyczna nie jest już używana w systemach produkcyjnych ze względu na niską odporność na ataki komputerowe, jej zasady - takie jak operacje permutacji, substytucji i cyklicznego kluczowania - stanowią podstawę projektowania nowoczesnych algorytmów blokowych, np. AES (Advanced Encryption Standard).

1.4.2. Kryptografia symetryczna

Kryptografia symetryczna, zwana również kryptografią klucza tajnego (secret-key cryptography), to grupa metod, w których ten sam klucz wykorzystywany jest zarówno do szyfrowania, jak i do deszyfrowania danych. Oznacza to, że obie strony komunikacji muszą dysponować tym samym, wcześniej uzgodnionym kluczem, co stanowi zarówno jej zaletę (szybkość działania), jak i słabość (trudność w bezpiecznym przekazaniu klucza).

Podstawowy model symetrycznego szyfrowania można zapisać jako:

$$C = E_K(P)$$

$$P = D_K(C)$$

gdzie:

- P - tekst jawny (plaintext),
- C - szyfrogram (ciphertext),
- E_K - funkcja szyfrowania,
- D_K - funkcja deszyfrowania,
- K - wspólny klucz.

Wyróżnia się dwa główne typy algorytmów symetrycznych:

- Szyfry blokowe (block ciphers) - przetwarzają dane w blokach o stałej długości, np. 64 lub 128 bitów. Przykładami są DES, 3DES, AES, Blowfish, Twofish, Serpent.
- Szyfry strumieniowe (stream ciphers) - przetwarzają dane bit po bicie lub bajt po bajcie, generując pseudolosowy strumień klucza (np. RC4, ChaCha20, Salsa20).

Do głównych zalet kryptografii symetrycznej należą:

- bardzo duża szybkość działania,
- mniejsze zapotrzebowanie na zasoby obliczeniowe,
- łatwa implementacja sprzętowa.

Jej głównym ograniczeniem jest problem dystrybucji klucza - przekazanie go drugiej stronie w sposób bezpieczny jest trudne, zwłaszcza w sieciach otwartych.

Z tego powodu w nowoczesnych systemach często stosuje się podejście hybrydowe, w którym klucz symetryczny jest szyfrowany algorytmem asymetrycznym i dopiero potem wykorzystywany do szyfrowania danych użytkowych (np. w protokole TLS/SSL).

1.4.3. Kryptografia asymetryczna

Kryptografia asymetryczna (zwana również kryptografią klucza publicznego) stanowi jedno z najważniejszych osiągnięć w dziedzinie bezpieczeństwa informacji. W przeciwieństwie do kryptografii symetrycznej, używa ona dwóch różnych kluczy:

- klucza publicznego (public key) - udostępnianego publicznie,

- klucza prywatnego (private key) - znanego tylko właścicielowi.

Proces szyfrowania i deszyfrowania można zapisać następująco:

$$C = E_{K_{pub}}(P)$$

$$P = D_{K_{pry}}$$

gdzie:

- P - tekst jawny (plaintext),
- C - szyfrogram (ciphertext),
- $E_{K_{pub}}$ - funkcja szyfrowania z użyciem klucza publicznego,
- $D_{K_{pry}}$ - funkcja deszyfrowania z użyciem klucza prywatnego.

Zależność między kluczami jest taka, że poznanie klucza publicznego nie pozwala w praktyce na wyznaczenie klucza prywatnego, o ile odpowiednie problemy matematyczne pozostają trudne do rozwiązania (np. faktoryzacja dużych liczb pierwszych, problem logarytmu dyskretnego czy problem krzywych eliptycznych).

Najważniejsze algorytmy asymetryczne:

- RSA (Rivest-Shamir-Adleman) - opiera się na trudności faktoryzacji dużych liczb.
- DSA (Digital Signature Algorithm) - stosowany głównie w podpisach cyfrowych.
- ElGamal - wykorzystuje problem logarytmu dyskretnego.
- ECC (Elliptic Curve Cryptography) - bazuje na teorii krzywych eliptycznych, oferując wysoki poziom bezpieczeństwa przy krótszych kluczach.

Kryptografia asymetryczna znajduje zastosowanie m.in. w:

- protokołach wymiany kluczy (np. Diffie-Hellman),
- systemach podpisu cyfrowego,
- uwierzytelnianiu serwerów (np. certyfikaty SSL/TLS),
- szyfrowaniu poczty elektronicznej (PGP, GPG),
- technologiach blockchain i kryptowalutach.

Główną zaletą kryptografii asymetrycznej jest eliminacja problemu dystrybucji klucza, jednak kosztem znacznie większego zapotrzebowania na moc obliczeniową. W praktyce stosuje się ją zatem do szyfrowania krótkich danych (np. kluczy sesyjnych), natomiast właściwe dane szyfruje się metodami symetrycznymi.

1.5. Zastosowania

Kryptografia znajduje dziś szerokie zastosowanie w niemal każdej dziedzinie informatyki i telekomunikacji. Wraz z rozwojem globalnej infrastruktury sieciowej, ochrony danych osobowych oraz usług cyfrowych, stała się nieodzownym elementem zapewniającym bezpieczeństwo i zaufanie w środowisku cyfrowym.

Do najważniejszych obszarów praktycznego zastosowania kryptografii należą:

1. Komunikacja i transmisja danych

Protokoły szyfrowania, takie jak TLS (Transport Layer Security) i SSL (Secure Sockets Layer), umożliwiają bezpieczne przesyłanie danych w sieci Internet. Dzięki nim możliwe jest bezpieczne korzystanie z bankowości internetowej, poczty elektronicznej czy portali społecznościowych.

2. Uwierzytelnianie i kontrola dostępu

Systemy kryptograficzne umożliwiają potwierdzenie tożsamości użytkowników i urządzeń. Stosowane są w kartach chipowych, tokenach, kluczach sprzętowych (np. YubiKey), systemach logowania dwuskładnikowego (2FA) oraz w infrastrukturach klucza publicznego (PKI).

3. Ochrona danych

Szyfrowanie dysków (np. BitLocker, VeraCrypt) oraz plików (np. GPG, AES Crypt) zapewnia ochronę danych przed nieautoryzowanym dostępem, zwłaszcza w przypadku utraty lub kradzieży urządzenia.

4. Podpisy cyfrowe i certyfikacja

Podpisy cyfrowe oparte na kryptografii asymetrycznej zapewniają autentyczność dokumentów elektronicznych i są podstawą usług zaufania zgodnych z rozporządzeniem

eIDAS. Certyfikaty cyfrowe wydawane przez urzędy certyfikacji (CA) potwierdzają wiarygodność tożsamości stron komunikujących się w sieci.

5. Systemy finansowe i kryptowaluty

Kryptografia stanowi fundament technologii blockchain, która zapewnia integralność i niezmienność danych w zdecentralizowanych systemach finansowych (np. Bitcoin, Ethereum). Funkcje skrótu oraz podpisy cyfrowe są tam wykorzystywane do zabezpieczania transakcji i weryfikacji uczestników sieci.

6. Bezpieczne protokoły i sieci VPN

Kryptograficzne mechanizmy tunelowania, takie jak IPSec, OpenVPN czy WireGuard, umożliwiają tworzenie bezpiecznych połączeń sieciowych, chroniąc dane przesyłane w publicznych sieciach.

7. Ochrona integralności oprogramowania

Podpisy cyfrowe oraz sumy kontrolne (np. SHA-256) wykorzystywane są do weryfikacji autentyczności oprogramowania i plików instalacyjnych, zapobiegając infekcjom przez złośliwe oprogramowanie.

Współczesne zastosowania kryptografii są niezwykle szerokie - od zabezpieczania komunikacji w Internecie, przez systemy bankowe, aż po technologie mobilne i IoT. W erze cyfrowej, gdzie dane stały się jednym z najcenniejszych zasobów, kryptografia pełni kluczową rolę w utrzymaniu zaufania, prywatności i integralności informacji.

2. Klasyczne metody szyfrowania

Klasyczne metody szyfrowania stanowią historyczne podstawy współczesnej kryptografii. Choć obecnie nie są już wykorzystywane w praktycznych systemach bezpieczeństwa, odegrały kluczową rolę w rozwoju metod ochrony informacji i pozwalają zrozumieć podstawowe koncepcje, takie jak substytucja, permutacja, cykliczne przesunięcia czy zastosowanie klucza.

W klasycznej kryptografii szyfrowanie polegało na przekształcaniu znaków alfabetu według określonej reguły. Algorytmy te były projektowane tak, aby utrudnić przeciwnikowi odczytanie wiadomości bez znajomości klucza, jednak przy zastosowaniu analizy częstotliwości i metod statystycznych mogły zostać stosunkowo łatwo złamane.

W niniejszym rozdziale omówione zostaną dwa najbardziej znane klasyczne szyfry: szyfr Cezara i szyfr Vigenère'a, które ilustrują ewolucję metod szyfrowania od prostych systemów podstawieniowych do bardziej złożonych algorytmów wieloalfabetycznych.

2.1. Szyfr Cezara

Szyfr Cezara (ang. Caesar cipher) jest jednym z najstarszych znanych systemów kryptograficznych. Nazwa pochodzi od Juliusza Cezara, który według przekazów używał tej metody do szyfrowania korespondencji wojskowej. Szyfr ten jest prostym przykładem szyfru podstawieniowego (substitution cipher), w którym każda litera tekstu jawnego jest zastępowana inną literą z alfabetu według ustalonego przesunięcia.

Zasada działania

Szyfr Cezara opiera się na przesunięciu liter alfabetu o stałą liczbę pozycji. Jeśli przesunięcie wynosi k, to każda litera w tekście jawnym zostaje zamieniona na literę oddaloną o k pozycji w prawo.

Formalnie działanie szyfru można zapisać następująco:

$$C = E_k(P) = (P + k) \, mod \, 26$$

$$P = D_k(C) = (C - k) \, mod \, 26$$

gdzie:

- P - liczbowy odpowiednik litery tekstu jawnego (0-25),
- C - liczbowy odpowiednik litery szyfrogramu,
- k - klucz, czyli liczba przesunięć,
- E_K - funkcja szyfrowania,
- D_K - funkcja deszyfrowania,
- 26 - liczba liter alfabetu łacińskiego.

Dla alfabetu łacińskiego (26 liter) klucz kkk może przyjmować wartości od 1 do 25.

Przykład działania

Dla klucza $k = 3$:

1. Tekst jawny: **KRYPTON**
2. Przesunięcie o 3 pozycje: K → N, R → U, Y → B, P → S, T → W, O → R, N → Q
3. Wynik szyfrowania (szyfrogram): **NUBSWRQ**

Odwrotne przesunięcie (o 3 pozycje w lewo) pozwala na odzyskanie oryginalnej wiadomości.

Zalety i wady

Zalety:

- prostota implementacji,
- łatwość zrozumienia zasad działania,
- historyczne znaczenie w rozwoju kryptografii.

Wady:

- bardzo mała przestrzeń kluczy (tylko 25 możliwych przesunięć),
- całkowita podatność na atak brutalny (brute-force),
- łatwe złamanie metodą analizy częstotliwości (każda litera w języku naturalnym występuje z inną częstością, co można wykorzystać do odczytania wiadomości).

Znaczenie

Choć szyfr Cezara nie zapewnia żadnego realnego bezpieczeństwa we współczesnych warunkach, ma ogromne znaczenie dydaktyczne. Pozwala zrozumieć ideę klucza kryptograficznego oraz podstawową zasadę szyfrowania przez przekształcenie znaków. Jest też punktem wyjścia dla bardziej złożonych szyfrów, takich jak szyfr Vigenère'a czy szyfr Affine'a, które rozwijają jego koncepcję o większą zmienność i odporność na analizę statystyczną.

2.2. Szyfr Vigenère'a

Szyfr Vigenère'a (ang. Vigenère cipher) został opracowany w XVI wieku przez francuskiego dyplomatę Blaise'a de Vigenère'a. Jest to przykład wieloalfabetycznego szyfru podstawieniowego (polyalphabetic substitution cipher), który stanowił znaczne ulepszenie w stosunku do szyfru Cezara.

W odróżnieniu od prostego przesunięcia o stałą wartość, w szyfrze Vigenère'a każda litera tekstu jawnego jest szyfrowana przy użyciu innego przesunięcia, określonego przez litery słowa-klucza.

Zasada działania

Klucz szyfrujący w szyfrze Vigenère'a stanowi ciąg liter, np. „KLUCZ". Dla każdej litery wiadomości jawnej stosuje się przesunięcie odpowiadające literze z klucza (A = 0, B = 1, …, Z = 25).

Formalnie można zapisać:

$$C_i = (P_i + K_i) \; mod \; 26$$
$$P_i = (C_i - K_i) \; mod \; 26$$

gdzie:

- P_i - i-ta litera tekstu jawnego,
- C_i - i-ta litera szyfrogramu,
- K_i - i-ta litera klucza, powtarzana cyklicznie,

- 26 - liczba liter alfabetu łacińskiego.

Przykład działania

Tekst jawny: **KRYPTON**

Klucz: **KLUCZKL**

1. Przypisanie wartości literom (A = 0, B = 1, …, Z = 25):
   - KRYPTON → 10 17 24 15 19 14 13
   - KLUCZKL → 10 11 20 2 25 10 11
2. Szyfrowanie:
   - $C_i = (P_i + K_i) \, mod \, 26$
   - Wyniki: (20, 2, 18, 17, 18, 24, 24)
3. Wynik szyfrowania: **UCSRSYY**

Zalety i wady

Zalety:

- wieloalfabetyczność (każda litera może być szyfrowana inaczej),
- znacznie trudniejszy do złamania niż szyfr Cezara,
- przez długi czas uważany za „niezłamywalny” (aż do XIX wieku).

Wady:

- podatny na analizę statystyczną w przypadku krótkiego klucza (metoda Kasiski, analiza częstotliwości powtórzeń),
- konieczność bezpiecznego przekazania klucza (podobnie jak w kryptografii symetrycznej),
- nieodporny na współczesne techniki komputerowe.

Znaczenie

Szyfr Vigenère’a był przez ponad 300 lat uważany za niezwykle bezpieczny, ponieważ rozkłada częstotliwość liter tekstu jawnego na wiele alfabetów, przez co nie można łatwo wykryć wzorców. Dopiero w XIX wieku Friedrich Kasiski opracował metodę analizy

powtarzających się fragmentów szyfrogramu, która pozwoliła określić długość klucza i złamać szyfr.

Współcześnie szyfr ten ma znaczenie głównie edukacyjne — jest doskonałym przykładem przejścia od szyfrów monoalfabetycznych (jak Cezara) do wieloalfabetycznych, stanowiąc teoretyczne preludium do nowoczesnych szyfrów blokowych, które operują na danych w sposób podobny, lecz znacznie bardziej złożony matematycznie.

Szyfr Vigenère'a był jednym z pierwszych systemów, w którym wprowadzono ideę klucza cyklicznego oraz zależności między pozycją znaku a sposobem jego szyfrowania. Koncepcja ta znalazła później odzwierciedlenie w strukturze współczesnych algorytmów, takich jak Feistel Network w DES czy rundy przekształceń w AES.

3. Kryptografia symetryczna

Kryptografia symetryczna to metoda szyfrowania, w której do zaszyfrowania i odszyfrowania wiadomości używany jest ten sam klucz tajny. Oznacza to, że zarówno nadawca, jak i odbiorca muszą posiadać identyczny klucz, który musi być utrzymywany w tajemnicy przed osobami trzecimi.

Zaletą kryptografii symetrycznej jest jej wysoka wydajność - algorytmy tego typu są znacznie szybsze niż algorytmy asymetryczne, co sprawia, że dobrze nadają się do szyfrowania dużych ilości danych. Wadą natomiast jest problem dystrybucji kluczy, czyli bezpiecznego przekazania klucza drugiej stronie komunikacji.

3.1. Algorytm DES

Algorytm DES (Data Encryption Standard) to symetryczny algorytm szyfrowania blokowego, który dzieli dane na 64-bitowe bloki, szyfrując je za pomocą 56-bitowego klucza w 16 rundach. Został opracowany przez IBM i przyjęty jako federalny standard USA w 1977 roku, ale obecnie jest uznawany za niebezpieczny ze względu na zbyt krótką długość klucza. DES opiera się na strukturze Feistela, w której dane są przetwarzane w wielu rundach, a każda runda składa się z serii permutacji i substytucji.

Zasada działania

DES działa w oparciu o zasadę szyfrowania blokowego, co oznacza, że dane są dzielone na bloki o długości 64 bitów, które są następnie szyfrowane niezależnie od siebie. Jeśli długość danych nie jest wielokrotnością 64 bitów, stosuje się tzw. padding - czyli uzupełnienie danych do pełnego bloku.

Szyfrowanie odbywa się z wykorzystaniem klucza o długości 56 bitów, chociaż formalnie zapis klucza ma 64 bity - osiem z nich służy jednak wyłącznie do kontroli parzystości, a nie do samego szyfrowania. DES przeprowadza 16 rund przetwarzania, w których wykonywane są różne operacje matematyczne i logiczne mające na celu maksymalne „wymieszanie" danych wejściowych.

Etapy działania

- Permutacja wstępna (Initial Permutation - IP)

  Na początku każdy 64-bitowy blok danych jest poddawany wstępnej permutacji, która zmienia kolejność bitów zgodnie z określoną tablicą. Następnie blok dzielony jest na dwie części - lewą (L0) i prawą (R0) - po 32 bity każda.

- Szesnaście rund szyfrowania

  W każdej z 16 rund wykonywana jest ta sama sekwencja operacji, w której prawa połowa bloku z poprzedniej rundy jest przetwarzana za pomocą specjalnej funkcji rundy.

  - Funkcja ta łączy dane z kluczem (za pomocą operacji XOR),
  - stosuje substytucje (podstawienia za pomocą tzw. S-boksów),
  - a następnie permutacje (zmiany kolejności bitów).

  Po przetworzeniu następuje zamiana stron - prawa część staje się lewą w kolejnej rundzie, a lewa zostaje zmodyfikowana wynikiem działania funkcji.

- Generowanie kluczy rundowych

  Klucz główny o długości 56 bitów podlega serii permutacji i przesunięć, które tworzą 16 kluczy rundowych, po jednym dla każdej rundy. Każdy z nich ma długość 48 bitów i jest wykorzystywany tylko raz w odpowiedniej rundzie.

- Permutacja końcowa (Final Permutation - IP^{-1})

  Po zakończeniu 16 rund obie części bloku są ponownie łączone, a całość poddawana jest permutacji końcowej, będącej odwrotnością permutacji początkowej. Wynikowy blok 64-bitowy stanowi zaszyfrowane dane (ciphertext).

Słabe strony i rozwój algorytmu

Pomimo swojej historycznej roli, algorytm DES nie spełnia współczesnych wymagań bezpieczeństwa. Jego największą wadą jest zbyt krótki klucz - 56 bitów, co czyni go podatnym na atak brute force, czyli próbę odgadnięcia klucza poprzez sprawdzenie wszystkich możliwych kombinacji. Obecnie, przy użyciu nowoczesnego sprzętu komputerowego, taki atak można przeprowadzić w ciągu kilku dni, a nawet godzin.

W dodatku, ze względu na znaną strukturę DES-a, ataki są jeszcze łatwiejsze, jeśli napastnik posiada fragmenty tekstu jawnego (tzw. known-plaintext attack).

Aby wydłużyć żywotność DES-a, opracowano jego ulepszoną wersję - 3DES (Triple DES) - w której dane są trzykrotnie szyfrowane algorytmem DES przy użyciu trzech różnych kluczy. Choć 3DES zwiększył bezpieczeństwo, był znacznie wolniejszy, dlatego również został stopniowo zastąpiony przez AES (Advanced Encryption Standard) - nowoczesny, szybszy i znacznie bardziej odporny na ataki.

3.2. Algorytm AES

AES (Advanced Encryption Standard) to nowoczesny symetryczny algorytm szyfrowania blokowego, który stanowi obecnie światowy standard szyfrowania danych. Został opracowany w celu zastąpienia przestarzałego algorytmu DES, którego bezpieczeństwo okazało się niewystarczające wobec postępu technologicznego. W 2001 roku został zatwierdzony przez NIST (National Institute of Standards and Technology) jako oficjalny standard kryptograficzny Stanów Zjednoczonych.

AES jest powszechnie stosowany w różnych dziedzinach - od szyfrowania danych w sieciach komputerowych (np. w protokołach HTTPS czy Wi-Fi), przez zabezpieczanie plików i dysków twardych, aż po systemy bankowe i urządzenia mobilne.

Zasada działania

Algorytm AES działa w oparciu o zasady kryptografii symetrycznej, co oznacza, że ten sam klucz jest używany zarówno do szyfrowania, jak i odszyfrowywania danych. AES przetwarza informacje w blokach o długości 128 bitów, a długość klucza może wynosić 128, 192 lub 256 bitów. W zależności od długości klucza wykonywana jest różna liczba rund szyfrowania:

- 10 rund dla klucza 128-bitowego,
- 12 rund dla klucza 192-bitowego,
- 14 rund dla klucza 256-bitowego.

Każda runda obejmuje zestaw operacji matematycznych wykonywanych na danych w oparciu o tzw. strukturę Substitution-Permutation Network (SPN), która łączy podstawienia i permutacje bitów w celu uzyskania wysokiego poziomu losowości i rozproszenia informacji.

Etapy działania

1. Key Expansion (Rozszerzenie klucza)

   Z pierwotnego klucza generowany jest zestaw tzw. kluczy rundowych, które są wykorzystywane kolejno w każdej rundzie szyfrowania.

2. Initial Round (Runda początkowa)

   Proces szyfrowania rozpoczyna się od operacji AddRoundKey, w której blok danych jest łączony (za pomocą operacji XOR) z pierwszym kluczem rundowym.

3. Rundy główne (Main Rounds)

   W każdej rundzie wykonywany jest zestaw czterech podstawowych operacji:

   - SubBytes - każda wartość bajtowa w bloku jest zastępowana inną, zgodnie z ustaloną tablicą podstawień (tzw. S-box), co wprowadza nieliniowość do procesu.
   - ShiftRows - bajty w każdej z czterech wierszy bloku są przesuwane cyklicznie o różną liczbę pozycji, co zwiększa rozproszenie danych.
   - MixColumns - kolumny bloku są przekształcane matematycznie (za pomocą operacji w ciele Galois $GF(2^8)$), co powoduje dalsze wymieszanie bitów.

- AddRoundKey - wynikowa macierz danych jest ponownie łączona z kolejnym kluczem rundowym.

4. Runda końcowa (Final Round)
5. W ostatniej rundzie wykonywane są tylko trzy operacje: SubBytes, ShiftRows i AddRoundKey - etap MixColumns jest pomijany. Otrzymany w ten sposób blok 128-bitowy stanowi zaszyfrowany tekst (ciphertext).

Proces deszyfrowania w AES przebiega analogicznie, jednak operacje wykonywane są w odwrotnej kolejności i przy użyciu odpowiednich odwrotnych transformacji.

Zalety i znaczenie

Algorytm AES jest obecnie uznawany za bardzo bezpieczny i odporny na wszystkie znane typy ataków kryptograficznych, w tym ataki siłowe, różnicowe czy liniowe. Dzięki dużej długości klucza oraz złożonej strukturze matematycznej, praktyczne złamanie AES metodami brute force jest obecnie niemożliwe.

Dodatkową zaletą AES jest jego wysoka wydajność - algorytm jest zoptymalizowany zarówno dla sprzętu (procesorów, kart sieciowych), jak i oprogramowania. Dzięki temu może być stosowany w bardzo szerokim zakresie zastosowań - od szyfrowania komunikacji w internecie, przez systemy bankowe, po ochronę danych w urządzeniach mobilnych i IoT.

Dzięki połączeniu bezpieczeństwa, szybkości i uniwersalności, AES stał się globalnym standardem szyfrowania danych. Jest używany przez rządy, organizacje międzynarodowe i sektor prywatny do ochrony informacji niejawnych i poufnych. Współcześnie AES stanowi podstawowy element większości nowoczesnych systemów bezpieczeństwa cyfrowego.

4. Kryptografia asymetryczna

Kryptografia asymetryczna, zwana również kryptografią klucza publicznego, to metoda szyfrowania, w której do zabezpieczania danych wykorzystywana jest para powiązanych ze sobą kluczy - klucz publiczny i klucz prywatny. W przeciwieństwie do kryptografii symetrycznej, gdzie ten sam klucz służy zarówno do szyfrowania, jak i odszyfrowywania danych, w kryptografii asymetrycznej każdy z kluczy pełni odmienną funkcję.

Klucz publiczny może być udostępniany wszystkim użytkownikom, natomiast klucz prywatny musi pozostać ściśle tajny i znany wyłącznie właścicielowi. Taka konstrukcja pozwala nie tylko na bezpieczne szyfrowanie komunikacji bez wcześniejszej wymiany kluczy, ale również na uwierzytelnianie tożsamości nadawcy poprzez tzw. podpisy cyfrowe.

Kryptografia asymetryczna stanowi dziś podstawę bezpieczeństwa w Internecie, bankowości elektronicznej, systemach płatności, komunikacji e-mail, a także w technologii blockchain i kryptowalutach.

Zasada działania

1. Klucz publiczny

   Jest to klucz, który może być swobodnie udostępniany innym użytkownikom. Służy do szyfrowania danych lub do weryfikacji podpisu cyfrowego. Przechwycenie klucza publicznego przez osoby trzecie nie stanowi zagrożenia, ponieważ nie pozwala na odszyfrowanie wiadomości bez posiadania odpowiadającego mu klucza prywatnego.

2. Klucz prywatny

   To klucz, który musi być ściśle chroniony przez właściciela. Umożliwia odszyfrowanie danych zaszyfrowanych kluczem publicznym lub tworzenie podpisów cyfrowych, które potwierdzają autentyczność nadawcy.

3. Proces szyfrowania i odszyfrowywania
   - Nadawca, chcąc przesłać poufną wiadomość, szyfruje ją kluczem publicznym odbiorcy.
   - Tylko odbiorca, który posiada odpowiedni klucz prywatny, może ją odszyfrować.

- Dzięki temu nawet w przypadku przechwycenia wiadomości przez osoby trzecie, jej treść pozostaje niemożliwa do odczytania.

4. Podpis cyfrowy

   Kryptografia asymetryczna umożliwia również tworzenie tzw. podpisów cyfrowych, które stanowią elektroniczny odpowiednik podpisu własnoręcznego. Nadawca generuje podpis, szyfrując skrót (hash) wiadomości swoim kluczem prywatnym. Odbiorca, używając klucza publicznego nadawcy, może zweryfikować podpis, co gwarantuje:

   - autentyczność - pewność, że wiadomość pochodzi od konkretnego nadawcy,
   - integralność - pewność, że treść nie została zmodyfikowana,
   - niezaprzeczalność - nadawca nie może zaprzeczyć, że wysłał podpisaną wiadomość.

Przykłady zastosowań

- Bezpieczne protokoły internetowe (HTTPS, TLS) - umożliwiają szyfrowaną komunikację między przeglądarką a serwerem.
- Komunikacja e-mail (np. PGP, S/MIME) - pozwala na bezpieczne przesyłanie zaszyfrowanych wiadomości oraz stosowanie podpisów cyfrowych.
- Bankowość internetowa i płatności online - zabezpieczają dane klientów i transakcje finansowe.
- Blockchain i kryptowaluty (np. Bitcoin, Ethereum) - klucz prywatny służy do podpisywania transakcji i zarządzania cyfrowymi aktywami.
- Systemy logowania i uwierzytelniania - umożliwiają potwierdzenie tożsamości użytkownika bez przekazywania hasła.

Najpopularniejsze algorytmy asymetryczne

Do najbardziej znanych algorytmów kryptografii asymetrycznej należą:

- RSA (Rivest-Shamir-Adleman) - jeden z pierwszych i najczęściej stosowanych algorytmów, oparty na trudności faktoryzacji dużych liczb.
- DSA (Digital Signature Algorithm) - wykorzystywany głównie do tworzenia podpisów cyfrowych.
- ECC (Elliptic Curve Cryptography) - nowoczesny algorytm oparty na krzywych eliptycznych, zapewniający wysoki poziom bezpieczeństwa przy krótszych kluczach.

Zalety i wady

Zalety:

- Brak konieczności bezpiecznej wymiany kluczy - klucz publiczny może być jawnie udostępniany.
- Możliwość uwierzytelniania i podpisywania wiadomości.
- Wysoki poziom bezpieczeństwa przy odpowiednich długościach kluczy.

Wady:

- Niższa wydajność w porównaniu z algorytmami symetrycznymi - szyfrowanie i odszyfrowywanie zajmuje więcej czasu.
- Złożoność obliczeniowa - wymaga większej mocy obliczeniowej, co może być problematyczne w urządzeniach o ograniczonych zasobach.

4.1. Algorytm RSA

Algorytm RSA (Rivest-Shamir-Adleman) to jeden z najstarszych i najczęściej stosowanych asymetrycznych algorytmów kryptograficznych. Został opracowany w 1977 roku przez trzech naukowców z Massachusetts Institute of Technology: Rona Rivesta, Adiego Shamira i Leonarda Adlemana, od których inicjałów pochodzi jego nazwa.

RSA wykorzystuje parę powiązanych kluczy - publiczny i prywatny - do bezpiecznego szyfrowania i odszyfrowywania danych. Klucz publiczny może być jawnie udostępniany, natomiast klucz prywatny musi pozostać ściśle tajny, ponieważ to właśnie on umożliwia odszyfrowanie wiadomości lub weryfikację podpisu cyfrowego.

Podstawą bezpieczeństwa RSA jest trudność rozkładania dużych liczb złożonych na czynniki pierwsze. W praktyce oznacza to, że choć łatwo jest pomnożyć dwie duże liczby pierwsze, to proces odwrotny - ich faktoryzacja - jest niezwykle czasochłonny i praktycznie niemożliwy do wykonania w rozsądnym czasie przy użyciu współczesnych komputerów.

Zasada działania

Działanie RSA można podzielić na trzy główne etapy:

1. Generacja kluczy

W pierwszej fazie użytkownik generuje parę kluczy:

- Klucz publiczny - może być swobodnie udostępniany i służy do szyfrowania danych lub weryfikacji podpisu cyfrowego.
- Klucz prywatny - pozostaje tajny i służy do odszyfrowywania wiadomości lub tworzenia podpisu cyfrowego.

2. Proces generacji polega na wyborze dwóch dużych liczb pierwszych (p i q), obliczeniu ich iloczynu ($n = p \times q$), który stanowi podstawę klucza, oraz wyznaczeniu odpowiednich wykładników szyfrowania (e) i odszyfrowania (d), które spełniają określone zależności matematyczne.
3. Szyfrowanie danych

   Nadawca, chcąc przesłać zaszyfrowaną wiadomość, używa klucza publicznego odbiorcy. Dane (reprezentowane jako liczby) są podnoszone do potęgi określonej przez wykładnik publiczny e i dzielone modulo n. Wynik tego działania stanowi szyfrogram, który może zostać bezpiecznie przesłany.

4. Odszyfrowanie danych

   Odbiorca, posiadający klucz prywatny, dokonuje operacji odwrotnej - podnosi otrzymany szyfrogram do potęgi określonej przez wykładnik prywatny d i również dzieli wynik modulo n. W ten sposób odzyskuje oryginalną wiadomość.

Kluczowe cechy i bezpieczeństwo

- Asymetryczność - dwa różne klucze: publiczny i prywatny, używane w procesie szyfrowania i deszyfrowania.
- Bezpieczeństwo oparte na matematyce - odporność RSA wynika z trudności faktoryzacji bardzo dużych liczb złożonych (liczących setki lub tysiące bitów).
- Skalowalność - długość klucza można dostosować do wymaganego poziomu bezpieczeństwa (obecnie zaleca się klucze o długości co najmniej 2048 bitów).
- Mimo swojej dużej niezawodności, RSA jest stosunkowo wolnym algorytmem, dlatego często używa się go nie do szyfrowania całych wiadomości, lecz jedynie do szyfrowania kluczy sesyjnych w kryptografii hybrydowej (łączącej kryptografię asymetryczną i symetryczną).

Przykłady zastosowań

- Szyfrowanie komunikacji internetowej - wykorzystywany w protokołach bezpieczeństwa takich jak TLS/SSL (np. w połączeniach HTTPS).
- Systemy bankowości elektronicznej i płatności online - zabezpiecza transmisję danych między użytkownikiem a serwerem.
- Podpisy cyfrowe - umożliwia uwierzytelnienie nadawcy i potwierdzenie integralności danych.
- Szyfrowanie wiadomości e-mail - stosowany w systemach takich jak PGP (Pretty Good Privacy).
- Infrastruktura klucza publicznego (PKI) - stanowi podstawę certyfikatów cyfrowych i autoryzacji użytkowników.

5. Implementacja i pomiary wydajności

5.1. Środowisko testowe

Macbook Pro M3 Pro - 18 GB RAM, 12 rdzeni CPU (5 wydajnościowych + 6 efektywnych)

5.2. Najważniejsze fragmenty kodu

W ramach implementacji stworzono modularną bibliotekę w języku Python, wykorzystującą biblioteki PyCryptodome (do operacji kryptograficznych) oraz standardowe moduły Pythona (os, random, multiprocessing, psutil, matplotlib, numpy). Kod został podzielony na logiczne moduły, zapewniając separację odpowiedzialności, testowalność i czytelność. Poniżej przedstawiono kluczowe fragmenty kodu z opisem ich funkcjonalności.

5.2.1 Abstrakcyjna klasa bazowa - src/algorithms/base.py

Plik definiuje abstrakcyjną klasę bazową Cipher, która ustala wspólny interfejs dla wszystkich implementacji algorytmów szyfrujących. Wymaga zaimplementowania metod encrypt, decrypt oraz get_key_data, co umożliwia polimorficzne użycie w benchmarkach i zapewnia spójność API. Użycie ABC z abstractmethod wymusza poprawną implementację w klasach pochodnych.

```
from abc import ABC, abstractmethod

class Cipher(ABC):
    @abstractmethod
    def encrypt(self, data: bytes) -> bytes:
        pass

    @abstractmethod
    def decrypt(self, data: bytes) -> bytes:
        pass

    @abstractmethod
    def get_key_data(self):
        pass
```

5.2.2 Klasyczne algorytmy - src/algorithms/classical.py

Plik zawiera implementacje klasycznych algorytmów: szyfru Cezara i Vigenère'a. Oba dziedziczą po Cipher. Szyfr Cezara używa przesunięcia modularnego w alfabecie, obsługując wielkość liter. Szyfr Vigenère'a stosuje cykliczny klucz wieloalfabetyczny. Oba operują na tekście w kodowaniu latin-1, by zachować wszystkie bajty. Metoda pomocnicza _shift_text i _vigenere_text realizuje logikę transformacji znak po znaku.

```
from .base import Cipher

class CaesarCipher(Cipher):
    def __init__(self, shift_or_key_data):
        if isinstance(shift_or_key_data, int):
            self.shift = shift_or_key_data
        else:
            self.shift = shift_or_key_data

    def encrypt(self, data: bytes) -> bytes:
        return self._shift_text(data.decode('latin-1'),
self.shift).encode('latin-1')

    def decrypt(self, data: bytes) -> bytes:
        return self._shift_text(data.decode('latin-1'),
-self.shift).encode('latin-1')

    def get_key_data(self):
        return self.shift
```

```
    def _shift_text(self, text: str, shift: int) -> str:
        result = ""
        for char in text:
            if 'a' <= char <= 'z':
                result += chr((ord(char) - ord('a') + shift) % 26 +
ord('a'))
            elif 'A' <= char <= 'Z':
                result += chr((ord(char) - ord('A') + shift) % 26 +
ord('A'))
            else:
                result += char
        return result
class VigenereCipher(Cipher):
    def __init__(self, key_or_key_data):
        if isinstance(key_or_key_data, str):
            self.key = [ord(k.lower()) - ord('a') for k in
key_or_key_data]
        else:
            self.key = key_or_key_data

    def encrypt(self, data: bytes) -> bytes:
        return self._vigenere_text(data.decode('latin-1'),
self.key).encode('latin-1')

    def decrypt(self, data: bytes) -> bytes:
        return self._vigenere_text(data.decode('latin-1'), [-k for k in
self.key]).encode('latin-1')

    def get_key_data(self):
        return self.key

    def _vigenere_text(self, text: str, key_shifts) -> str:
        key_len = len(key_shifts)
        result = ""
        for i, char in enumerate(text):
            shift = key_shifts[i % key_len]
            if 'a' <= char <= 'z':
                result += chr((ord(char) - ord('a') + shift) % 26 +
ord('a'))
            elif 'A' <= char <= 'Z':
                result += chr((ord(char) - ord('A') + shift) % 26 +
ord('A'))
            else:
                result += char
        return result
```

5.2.3 Algorytmy symetryczne - src/algorithms/symmetric.py

Zawiera implementacje algorytmów symetrycznych: AES i DES. AES używa trybu EAX (AEAD), zapewniającego poufność, integralność i autentyczność. Zwraca nonce || tag || ciphertext. DES działa w trybie ECB z ręcznym paddingiem PKCS#7 (dla celów edukacyjnych). Oba algorytmy przyjmują klucz w formie bytes.

```
from .base import Cipher
from Crypto.Cipher import AES, DES

class AESCipher(Cipher):
    def __init__(self, key_or_key_data):
        self.key = key_or_key_data

    def encrypt(self, data: bytes) -> bytes:
        cipher = AES.new(self.key, AES.MODE_EAX)
        ciphertext, tag = cipher.encrypt_and_digest(data)
        return cipher.nonce + tag + ciphertext

    def decrypt(self, data: bytes) -> bytes:
        nonce = data[:16]
        tag = data[16:32]
        ciphertext = data[32:]
        cipher = AES.new(self.key, AES.MODE_EAX, nonce=nonce)
        return cipher.decrypt_and_verify(ciphertext, tag)

    def get_key_data(self):
        return self.key

class DESCipher(Cipher):
    def __init__(self, key_or_key_data):
        self.key = key_or_key_data

    def encrypt(self, data: bytes) -> bytes:
        cipher = DES.new(self.key, DES.MODE_ECB)
        pad_len = 8 - len(data) % 8
        data_padded = data + bytes([pad_len]) * pad_len
        return cipher.encrypt(data_padded)

    def decrypt(self, data: bytes) -> bytes:
        cipher = DES.new(self.key, DES.MODE_ECB)
        decrypted = cipher.decrypt(data)
        pad_len = decrypted[-1]
        return decrypted[:-pad_len]
```

```
    def get_key_data(self):
        return self.key
```

5.2.4 Algorytm asymetryczny - src/algorithms/asymmetric.py

Implementacja RSA z użyciem paddingu PKCS#1 OAEP. Klucz importowany z formatu PEM. Szyfrowanie i deszyfrowanie realizowane w chunkach (190B dla szyfrowania, 256B dla deszyfrowania) ze względu na ograniczenie rozmiaru bloku w RSA-2048. Użyto PKCS1_OAEP dla bezpieczeństwa przed atakami CCA.

```
from .base import Cipher
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

class RSACipher(Cipher):
    def __init__(self, key_or_key_data):
        if isinstance(key_or_key_data, str):
            key_pem = key_or_key_data.encode('utf-8')
        else:
            key_pem = key_or_key_data
        self.key = RSA.import_key(key_pem)
        self.cipher_enc = PKCS1_OAEP.new(self.key.publickey())
        self.cipher_dec = PKCS1_OAEP.new(self.key)

    def encrypt(self, data: bytes) -> bytes:
        chunk_size = 190
        encrypted_chunks =
[self.cipher_enc.encrypt(data[i:i+chunk_size]) for i in range(0,
len(data), chunk_size)]
        return b"".join(encrypted_chunks)

    def decrypt(self, data: bytes) -> bytes:
        chunk_size = 256
        decrypted_chunks =
[self.cipher_dec.decrypt(data[i:i+chunk_size]) for i in range(0,
len(data), chunk_size)]
        return b"".join(decrypted_chunks)

    def get_key_data(self):
        return self.key.export_key(format='PEM').decode('utf-8')
```

5.2.4 Pomiar wydajności - src/measurement/performance.py

Moduł odpowiedzialny za precyzyjne pomiary czasu, użycia CPU i pamięci. Używa osobnego procesu (multiprocessing) dla izolacji. Monitoruje szczytowe zużycie pamięci (USS) i procent CPU. Dane wejściowe zapisywane tymczasowo do pliku, by uniknąć wpływu buforowania w pamięci.

```
import time
import psutil
import tempfile
import os
from multiprocessing import Process, Queue

class PerformanceMeasurer:
    def measure(self, cipher_class, method_name, data, key_data):
        with tempfile.NamedTemporaryFile(delete=False) as f:
            f.write(data)
            data_file = f.name
        queue = Queue()
        p = Process(target=self._target_func, args=(queue, cipher_class,
method_name, data_file, key_data))
        p.start()
        peak_mem = self._monitor_memory(p)
        p.join()
        os.unlink(data_file)
        results = queue.get()
        results["mem"] = peak_mem / (1024 ** 2)
        results["cpu"] = results["cpu"] / psutil.cpu_count()
        return results

    def _target_func(self, queue, cipher_class, method_name, data_file,
key_data):
        with open(data_file, 'rb') as f:
            data = f.read()
        cipher = cipher_class(key_data)
        process = psutil.Process()
        process.cpu_percent(interval=None)
        time.sleep(0.1)
        start = time.perf_counter()
        getattr(cipher, method_name)(data)
        elapsed = time.perf_counter() - start
        cpu_usage = process.cpu_percent(interval=None)
        queue.put({"time": elapsed, "cpu": cpu_usage})

    def _monitor_memory(self, p):
        try:
```

```
            ps_p = psutil.Process(p.pid)
            peak_mem = 0
            while p.is_alive():
                try:
                    mem_info = ps_p.memory_info()
                    current_mem = mem_info.uss if hasattr(mem_info,
'uss') else mem_info.rss
                    if current_mem > peak_mem:
                        peak_mem = current_mem
                except psutil.NoSuchProcess:
                    break
                time.sleep(0.01)
        except psutil.NoSuchProcess:
            peak_mem = 0
        return peak_mem
```

5.2.5 Generowanie danych i kluczy - src/data/generator.py

Generator losowych danych tekstowych (litery i cyfry). Używany do tworzenia danych testowych o zadanych rozmiarach (w znakach). Wynik to str, konwertowany później na bytes w latin-1.

```
import random
import string

class DataGenerator:
    def generate_random_text(self, size: int) -> str:
        return ''.join(random.choice(string.ascii_letters +
string.digits) for _ in range(size))
```

5.2.6 Zarządzanie generowaniem i zapisem kluczy - src/keys/manager.py

Zarządza generowaniem i zapisem kluczy dla wszystkich algorytmów. Tworzy losowe klucze: przesunięcie Cezara, klucz Vigenère'a, klucze AES/DES (w hex), klucz RSA-2048 (PEM). Zapisuje je do katalogu results/keys/ w czytelnej formie.

```python
import random
import string
import os
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes

class KeyManager:
    def __init__(self):
        self.keys = {}

    def generate_keys(self):
        self.keys = {
            "Caesar": random.randint(1, 25),
            "Vigenere": ''.join(random.choice(string.ascii_uppercase)
for _ in range(10)),
            "AES-128": get_random_bytes(16),
            "AES-192": get_random_bytes(24),
            "AES-256": get_random_bytes(32),
            "DES": get_random_bytes(8),
            "RSA-2048": RSA.generate(2048).export_key()
        }

    def save_keys(self):
        os.makedirs("results/keys", exist_ok=True)
        with open("results/keys/classical_keys.txt", "w") as f:
            f.write(f"Caesar Shift: {self.keys['Caesar']}\n")
            f.write(f"Vigenere Key: {self.keys['Vigenere']}\n")
        for name in ["AES-128", "AES-192", "AES-256", "DES"]:
            with open(f"results/keys/{name.lower()}_key.txt", "w") as f:
                f.write(f"{name} Key (hex): {self.keys[name].hex()}\n")
        with open("results/keys/rsa_2048_key.pem", "wb") as f:
            f.write(self.keys["RSA-2048"])

    def get_key(self, name):
        return self.keys[name]
```

5.2.6 Runner benchmarków - src/benchmark/runner.py

Główny silnik benchmarków. Koordynuje testy dla zadanych algorytmów i rozmiarów danych. Dla każdego rozmiaru generuje dane, wykonuje iteracje szyfrowania/deszyfrowania, zbiera metryki, oblicza średnie i zapisuje wyniki do pliku tekstowego. Na końcu wywołuje rysowanie wykresów.

```
import os
import numpy as np
from measurement.performance import PerformanceMeasurer
from plotting.plotter import Plotter
from data.generator import DataGenerator

class BenchmarkRunner:
    def __init__(self, measurer, plotter, generator):
        self.measurer = measurer
        self.plotter = plotter
        self.generator = generator

    def run_suite(self, suite_name, algorithms, sizes, iterations,
log_file, keys):
        suite_results = self._initialize_results(algorithms)
        self._write_suite_header(suite_name, log_file)
        for size in sizes:
            self._run_size_tests(size, algorithms, iterations,
suite_results, log_file)
        self.plotter.plot_results(suite_name, suite_results, sizes)

    def _initialize_results(self, algorithms):
        return {alg: {"encrypt_time": [], "decrypt_time": [],
"encrypt_cpu": [], "decrypt_cpu": [], "encrypt_mem": [], "decrypt_mem":
[]} for alg in algorithms}

    def _write_suite_header(self, suite_name, log_file):
        header = f"\n{'='*20}\n {suite_name} BENCHMARK \n{'='*20}\n"
        print(header)
        log_file.write(header)

    def _run_size_tests(self, size, algorithms, iterations,
suite_results, log_file):
        text = self.generator.generate_random_text(size)
        data = text.encode('latin-1')
        self._save_sample(text, size)
        self._write_size_header(size, iterations, log_file)
        for name, cipher in algorithms.items():
            encrypt_metrics, decrypt_metrics =
self._run_algorithm_tests(cipher, data, iterations)
            avg_encrypt = {k: np.mean(v) for k, v in
encrypt_metrics.items()}
            avg_decrypt = {k: np.mean(v) for k, v in
decrypt_metrics.items()}
            self._update_results(suite_results, name, avg_encrypt,
avg_decrypt)
```

```
            self._write_result_line(name, avg_encrypt, avg_decrypt,
log_file)

    def _save_sample(self, text, size):
        with open(f"results/samples/sample_{size}_chars.txt", "w") as
sf:
            sf.write(text)

    def _write_size_header(self, size, iterations, log_file):
        header = f"\n--- Testing {size / 1024:.0f} KB ({size}
characters) over {iterations} iterations ---\n"
        print(header)
        log_file.write(header)

    def _run_algorithm_tests(self, cipher, data, iterations):
        encrypt_iter = {"time": [], "cpu": [], "mem": []}
        decrypt_iter = {"time": [], "cpu": [], "mem": []}
        for _ in range(iterations):
            encrypted = cipher.encrypt(data)
            encrypt_metrics = self.measurer.measure(cipher.__class__,
"encrypt", data, cipher.get_key_data())
            decrypt_metrics = self.measurer.measure(cipher.__class__,
"decrypt", encrypted, cipher.get_key_data())
            for k in encrypt_iter:
                encrypt_iter[k].append(encrypt_metrics[k])
            for k in decrypt_iter:
                decrypt_iter[k].append(decrypt_metrics[k])
        return encrypt_iter, decrypt_iter

    def _update_results(self, suite_results, name, avg_encrypt,
avg_decrypt):
        for m_key in suite_results[name]:
            if 'encrypt' in m_key:

suite_results[name][m_key].append(avg_encrypt[m_key.replace('encrypt_',
'')])
            elif 'decrypt' in m_key:

suite_results[name][m_key].append(avg_decrypt[m_key.replace('decrypt_',
'')])

    def _write_result_line(self, name, avg_encrypt, avg_decrypt,
log_file):
        result_line = f"{name:<10} | encrypt: {avg_encrypt['time']:.5f}s
CPU:{avg_encrypt['cpu']:.2f}% MEM:{avg_encrypt['mem']:.4f}MB | decrypt:
{avg_decrypt['time']:.5f}s CPU:{avg_decrypt['cpu']:.2f}%
```

```
MEM:{avg_decrypt['mem']:.4f}MB\n"
        print(result_line, end="")
        log_file.write(result_line)
```

5.2.7 Wykresy - src/plotting/plotter.py

Rysuje wykresy wyników za pomocą matplotlib. Tworzy siatkę 2×3: czas, CPU, pamięć dla szyfrowania i deszyfrowania. Każdy algorytm ma inny znacznik. Wykresy zapisywane jako PNG w katalogu results/.

```
import matplotlib.pyplot as plt

class Plotter:
    def plot_results(self, suite_name, results, sizes):
        fig, axs = plt.subplots(2, 3, figsize=(24, 12))
        fig.suptitle(f"{suite_name} Algorithm Benchmark", fontsize=16)
        operations = ['encrypt', 'decrypt']
        metrics = [("time", "Time", "Time (s)"), ("cpu", "CPU Usage",
"CPU (%)"), ("mem", "RAM Usage", "RAM (MB)")]
        markers = ['o', 's', '^', 'D', 'v', '*', 'p', 'h']
        for i, op in enumerate(operations):
            for j, (metric, title, ylabel) in enumerate(metrics):
                ax = axs[i, j]
                for idx, (alg, data) in enumerate(results.items()):
                    marker = markers[idx % len(markers)]
                    ax.plot(sizes, data[f"{op}_{metric}"],
marker=marker, label=alg)
                ax.set_title(f"{op.capitalize()} {title}")
                ax.set_xlabel("Data Size [chars]")
                ax.set_ylabel(ylabel)
                ax.legend()
                ax.grid(True, which="both", ls="--")
        plt.tight_layout(rect=[0, 0, 1, 0.96])
        plt.savefig(f"results/benchmark_{suite_name.lower().replace(' ',
'_')}.png")
        plt.show()
```

5.2.7 Plik główny - src/main.py

Punkt wejścia aplikacji. Inicjuje wszystkie komponenty, generuje klucze, tworzy instancje algorytmów, uruchamia benchmarki dla trzech grup (klasyczne, symetryczne, asymetryczne) i zapisuje pełne wyniki (w tym klucze) do pliku tekstowego.

```
import os
import numpy as np
from algorithms.classical import CaesarCipher, VigenereCipher
from algorithms.symmetric import AESCipher, DESCipher
from algorithms.asymmetric import RSACipher
from measurement.performance import PerformanceMeasurer
from plotting.plotter import Plotter
from data.generator import DataGenerator
from keys.manager import KeyManager
from benchmark.runner import BenchmarkRunner

def main():
    os.makedirs("results/keys", exist_ok=True)
    os.makedirs("results/samples", exist_ok=True)

    iterations = 3

    key_manager = KeyManager()
    key_manager.generate_keys()
    key_manager.save_keys()

    measurer = PerformanceMeasurer()
    plotter = Plotter()
    generator = DataGenerator()
    runner = BenchmarkRunner(measurer, plotter, generator)

    classical_symmetric_sizes = [10000, 100000, 1000000, 10000000,
50000000, 100000000, 200000000]
    asymmetric_sizes = classical_symmetric_sizes

    algorithm_suites = {
        "Classical": {
            "Caesar": CaesarCipher(key_manager.get_key("Caesar")),
            "Vigenere": VigenereCipher(key_manager.get_key("Vigenere")),
        },
        "Symmetric": {
            "AES-128": AESCipher(key_manager.get_key("AES-128")),
```

```
            "AES-192": AESCipher(key_manager.get_key("AES-192")),
            "AES-256": AESCipher(key_manager.get_key("AES-256")),
            "DES": DESCipher(key_manager.get_key("DES")),
        },
        "Asymmetric": {
            "RSA-2048": RSACipher(key_manager.get_key("RSA-2048")),
        }
    }

    with open("results/benchmark_results.txt", "w") as f:
        f.write("Cryptographic Keys Used (for documentation):\n")
        f.write(f"Caesar Shift: {key_manager.get_key('Caesar')}\n")
        f.write(f"Vigenere Key: {key_manager.get_key('Vigenere')}\n")
        f.write(f"AES-128 Key (hex):
{key_manager.get_key('AES-128').hex()}\n")
        f.write(f"AES-192 Key (hex):
{key_manager.get_key('AES-192').hex()}\n")
        f.write(f"AES-256 Key (hex):
{key_manager.get_key('AES-256').hex()}\n")
        f.write(f"DES Key (hex): {key_manager.get_key('DES').hex()}\n")
        f.write("RSA-2048 Key (PEM):\n")
        f.write(key_manager.get_key("RSA-2048").decode('utf-8'))
        f.write("\n\n")
        runner.run_suite("Classical", algorithm_suites["Classical"],
classical_symmetric_sizes, iterations, f, key_manager.keys)
        runner.run_suite("Symmetric", algorithm_suites["Symmetric"],
classical_symmetric_sizes, iterations, f, key_manager.keys)
        runner.run_suite("Asymmetric", algorithm_suites["Asymmetric"],
asymmetric_sizes, iterations, f, key_manager.keys)

if __name__ == "__main__":
    main()
```

5.3. Wyniki pomiarów

Na podstawie dołączonych obrazów przedstawiających wyniki benchmarków, przedstawiamy analizę wydajności poszczególnych algorytmów kryptograficznych. Pomiary obejmują czas szyfrowania i deszyfrowania, zużycie procesora (CPU) oraz pamięci RAM dla różnych rozmiarów danych wejściowych.

5.3.1. Algorytmy asymetryczne (RSA-2048)

Asymmetric BENCHMARK

```
--- Testing 10 KB (10000 characters) over 3 iterations ---
RSA-2048 | encrypt: 0.02651s CPU:1.85% MEM:53.8906MB | decrypt: 0.11177s
CPU:4.68% MEM:53.8906MB
--- Testing 98 KB (100000 characters) over 3 iterations ---
RSA-2048 | encrypt: 0.18113s CPU:5.75% MEM:54.0677MB | decrypt: 1.03718s
CPU:8.25% MEM:54.0000MB
--- Testing 977 KB (1000000 characters) over 3 iterations ---
RSA-2048 | encrypt: 1.76566s CPU:8.56% MEM:56.5156MB | decrypt:
10.34412s CPU:8.97% MEM:56.7083MB
--- Testing 9766 KB (10000000 characters) over 3 iterations ---
RSA-2048 | encrypt: 17.96355s CPU:9.00% MEM:78.1823MB | decrypt:
104.68469s CPU:9.02% MEM:68.1458MB
--- Testing 48828 KB (50000000 characters) over 3 iterations ---
RSA-2048 | encrypt: 87.86856s CPU:9.05% MEM:173.1198MB | decrypt:
522.84710s CPU:9.03% MEM:133.7552MB
--- Testing 97656 KB (100000000 characters) over 3 iterations ---
RSA-2048 | encrypt: 175.92082s CPU:9.07% MEM:407.1667MB | decrypt:
1043.83708s CPU:9.05% MEM:261.5833MB
--- Testing 195312 KB (200000000 characters) over 3 iterations ---
RSA-2048 | encrypt: 350.21208s CPU:9.05% MEM:752.8958MB | decrypt:
2036.04543s CPU:9.08% MEM:530.8438MB
```

Analiza RSA-2048:

Czas szyfrowania i deszyfrowania:

RSA-2048 jest algorytmem asymetrycznym, co naturalnie oznacza znacznie dłuższe czasy szyfrowania i deszyfrowania w porównaniu do algorytmów symetrycznych i klasycznych.

Czas szyfrowania rośnie liniowo wraz ze wzrostem rozmiaru danych. Dla 10 KB wynosi 0.02651s, a dla 200 MB już 350.21208s.

Deszyfrowanie jest wielokrotnie wolniejsze niż szyfrowanie. Dla 10 KB zajmuje 0.11177s, a dla 200 MB aż 2036.04543s (ponad 34 minuty). To typowe dla RSA, ponieważ deszyfrowanie wymaga wykonania operacji na większych liczbach.

Zużycie procesora (CPU):

W przypadku mniejszych danych (10 KB) zużycie CPU jest relatywnie niskie (1.85% dla szyfrowania, 4.68% dla deszyfrowania), jednak szybko wzrasta i stabilizuje się na poziomie około 9% dla większych rozmiarów danych, co sugeruje, że algorytm jest dość

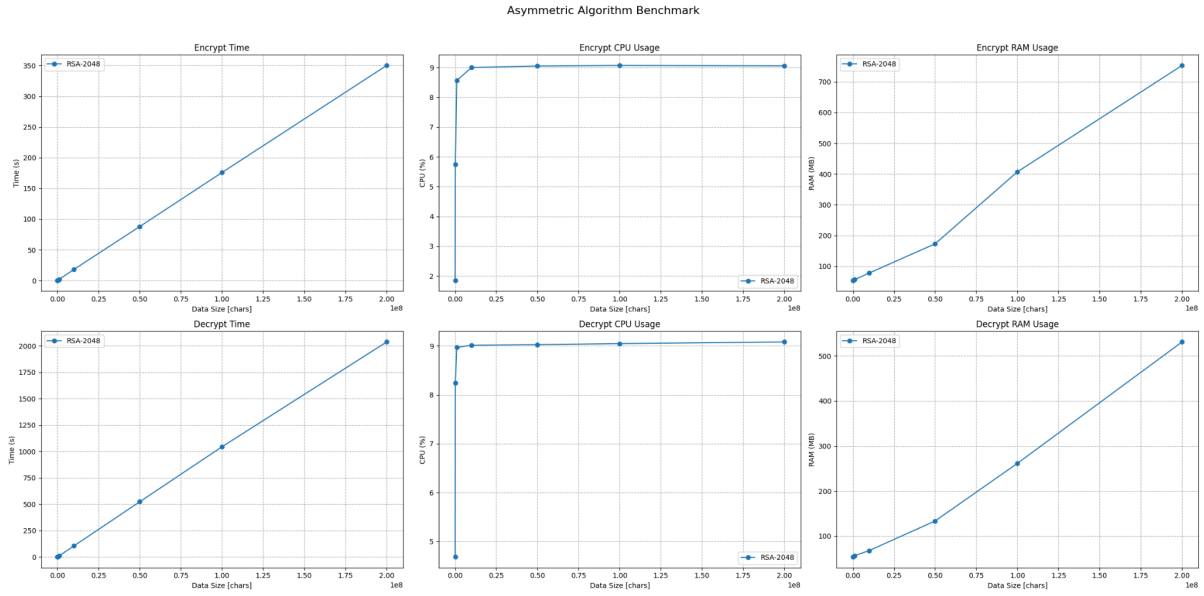
intensywny obliczeniowo i wykorzystuje dostępną moc CPU w sposób ciągły, ale nie maksymalny (w odniesieniu do całkowitej liczby rdzeni, wynik jest uśredniony).

Zużycie pamięci (RAM):

Początkowe zużycie pamięci jest na poziomie około 54 MB.

Wraz ze wzrostem rozmiaru danych, zużycie RAM rośnie liniowo, osiągając blisko 753 MB dla szyfrowania i 530 MB dla deszyfrowania dla 200 MB danych. Jest to spowodowane koniecznością przechowywania dużych bloków danych do przetwarzania, szczególnie z uwagi na podział na "chunki" w implementacji PyCryptodome dla RSA.

Wykresy dla Algorytmów Asymetrycznych:



Wykresy te jasno ilustrują liniową zależność czasu szyfrowania/deszyfrowania oraz zużycia RAM od rozmiaru danych dla RSA. Zużycie CPU stabilizuje się na wysokim poziomie, co wskazuje na intensywne wykorzystanie zasobów obliczeniowych.

5.3.2. Algorytmy klasyczne (Caesar, Vigenere)

Classical BENCHMARK

```
--- Testing 10 KB (10000 characters) over 3 iterations ---
Caesar | encrypt: 0.00274s CPU:0.24% MEM:53.2865MB | decrypt: 0.00254s
CPU:0.22% MEM:53.1771MB
Vigenere | encrypt: 0.00354s CPU:0.30% MEM:53.2083MB | decrypt: 0.00324s
CPU:0.28% MEM:53.1777MB
--- Testing 98 KB (100000 characters) over 3 iterations ---
```

```
Caesar | encrypt: 0.01938s CPU:1.42% MEM:53.1927MB | decrypt: 0.02157s
CPU:1.55% MEM:53.1667MB
Vigenere | encrypt: 0.02980s CPU:2.02% MEM:53.1979MB | decrypt: 0.04653s
CPU:1.79% MEM:53.1510MB
--- Testing 977 KB (1000000 characters) over 3 iterations ---
Caesar | encrypt: 0.21722s CPU:5.68% MEM:56.3333MB | decrypt: 0.20814s
CPU:5.92% MEM:56.4688MB
Vigenere | encrypt: 0.22333s CPU:6.20% MEM:56.3906MB | decrypt: 0.21578s
CPU:6.13% MEM:56.3073MB
--- Testing 9766 KB (10000000 characters) over 3 iterations ---
Caesar | encrypt: 1.87227s CPU:8.42% MEM:85.3594MB | decrypt: 1.91512s
CPU:8.51% MEM:84.7135MB
Vigenere | encrypt: 2.21715s CPU:8.62% MEM:84.9375MB | decrypt: 2.17258s
CPU:8.65% MEM:85.1771MB
--- Testing 48828 KB (50000000 characters) over 3 iterations ---
Caesar | encrypt: 9.28691s CPU:8.85% MEM:179.8854MB | decrypt: 9.24845s
CPU:8.94% MEM:199.6458MB
Vigenere | encrypt: 18.39576s CPU:8.99% MEM:196.0573MB | decrypt:
10.96554s CPU:8.98% MEM:196.6719MB
--- Testing 97656 KB (100000000 characters) over 3 iterations ---
Caesar | encrypt: 18.20934s CPU:8.93% MEM:294.9479MB | decrypt:
19.05352s CPU:8.90% MEM:266.3021MB
Vigenere | encrypt: 21.39576s CPU:8.99% MEM:322.4271MB | decrypt:
21.95003s CPU:8.98% MEM:309.8698MB
--- Testing 195312 KB (200000000 characters) over 3 iterations ---
Caesar | encrypt: 35.59866s CPU:9.03% MEM:517.3438MB | decrypt:
37.83614s CPU:8.93% MEM:478.8750MB
Vigenere | encrypt: 42.50445s CPU:9.05% MEM:618.8229MB | decrypt:
42.86230s CPU:9.05% MEM:540.4375MB
```

Analiza algorytmów klasycznych (Caesar, Vigenere):

Czas szyfrowania i deszyfrowania:

Oba algorytmy są bardzo szybkie dla małych danych, mierzone w milisekundach (np. dla 10 KB).

Czasy szyfrowania i deszyfrowania dla obu algorytmów są do siebie zbliżone i rosną liniowo wraz ze wzrostem rozmiaru danych.

Szyfr Vigenère'a jest nieznacznie wolniejszy niż szyfr Cezara, co jest spodziewane ze względu na bardziej złożoną logikę (operacje na kluczu cyklicznym). Różnice te stają się bardziej zauważalne przy większych rozmiarach danych.

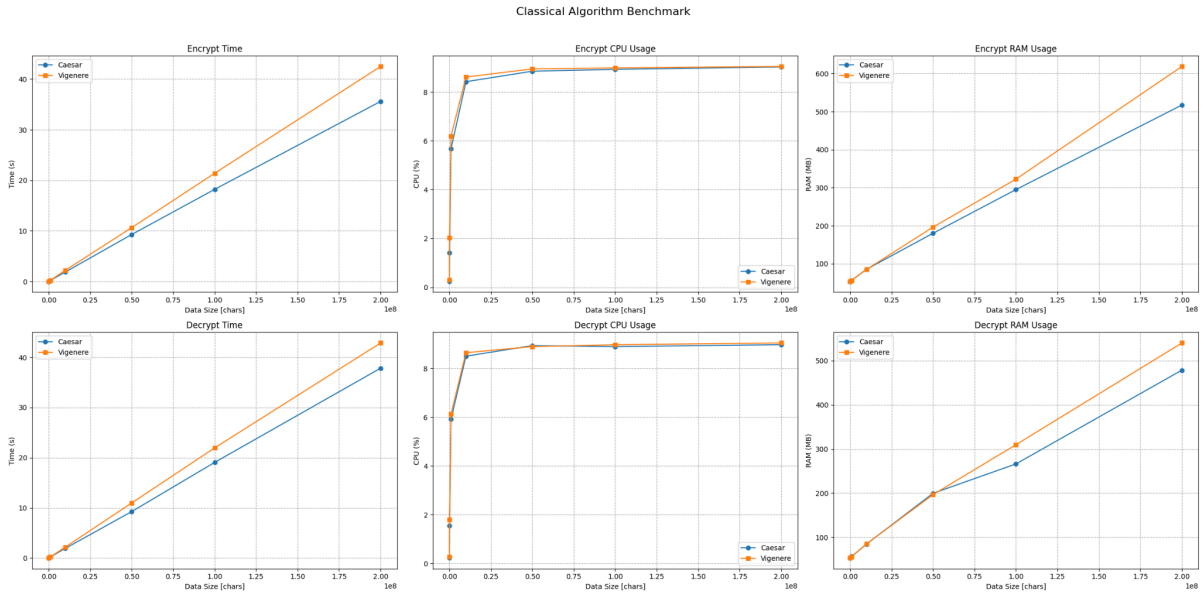
Zużycie procesora (CPU):

Podobnie jak w przypadku RSA, zużycie CPU zaczyna się od niskich wartości dla małych danych i stopniowo wzrasta, stabilizując się na poziomie około 8-9% dla większych rozmiarów. Jest to zachowanie typowe dla algorytmów, które skalują się liniowo z danymi i przetwarzają je sekwencyjnie.

Zużycie pamięci (RAM):

Zużycie pamięci jest również liniowo zależne od rozmiaru danych wejściowych. Dla 200 MB danych, Caesar zużywa około 517 MB (szyfrowanie) i 478 MB (deszyfrowanie), a Vigenere około 618 MB (szyfrowanie) i 540 MB (deszyfrowanie). Nieco wyższe zużycie pamięci przez Vigenère'a może wynikać z dodatkowych operacji na kluczu w każdej iteracji.

Wykresy dla Algorytmów Klasycznych:



Wykresy potwierdzają liniową skalowalność obu algorytmów klasycznych pod względem czasu i zużycia pamięci. Widać również niewielką różnicę wydajności na korzyść szyfru Cezara.

5.3.3. Algorytmy symetryczne (AES-128, AES-192, AES-256, DES)

Symmetric BENCHMARK

```
--- Testing 10 KB (10000 characters) over 3 iterations ---
AES-128 | encrypt: 0.00011s CPU:0.64% MEM:53.7188MB | decrypt: 0.00072s
CPU:0.59% MEM:53.5781MB
AES-192 | encrypt: 0.00784s CPU:0.64% MEM:53.5625MB | decrypt: 0.00076s
```

```
CPU:0.71% MEM:53.7449MB
AES-256 | encrypt: 0.00786s CPU:0.64% MEM:53.7917MB | decrypt: 0.00076s
CPU:0.60% MEM:53.6354MB
DES | encrypt: 0.00229s CPU:0.29% MEM:53.2604MB | decrypt: 0.00222s
CPU:0.29% MEM:53.4427MB
--- Testing 98 KB (100000 characters) over 3 iterations ---
AES-128 | encrypt: 0.00986s CPU:0.77% MEM:53.7240MB | decrypt: 0.00907s
CPU:0.73% MEM:53.7552MB
AES-192 | encrypt: 0.00841s CPU:0.68% MEM:53.5833MB | decrypt: 0.00899s
CPU:0.72% MEM:53.7135MB
AES-256 | encrypt: 0.00940s CPU:0.75% MEM:53.7396MB | decrypt: 0.00919s
CPU:0.74% MEM:53.5417MB
DES | encrypt: 0.00348s CPU:0.29% MEM:53.2969MB | decrypt: 0.00343s
CPU:0.29% MEM:53.3385MB
--- Testing 977 KB (1000000 characters) over 3 iterations ---
AES-128 | encrypt: 0.01558s CPU:1.17% MEM:57.5469MB | decrypt: 0.01534s
CPU:1.16% MEM:57.5000MB
AES-192 | encrypt: 0.01782s CPU:1.32% MEM:57.4531MB | decrypt: 0.01809s
CPU:1.34% MEM:57.4115MB
AES-256 | encrypt: 0.01894s CPU:1.39% MEM:57.4115MB | decrypt: 0.01924s
CPU:1.41% MEM:57.4896MB
DES | encrypt: 0.01582s CPU:1.19% MEM:57.2188MB | decrypt: 0.01579s
CPU:1.19% MEM:56.1823MB
--- Testing 9766 KB (10000000 characters) over 3 iterations ---
AES-128 | encrypt: 0.06931s CPU:3.62% MEM:91.8438MB | decrypt: 0.07114s
CPU:3.65% MEM:91.7292MB
AES-192 | encrypt: 0.07810s CPU:3.88% MEM:91.7188MB | decrypt: 0.07601s
CPU:3.88% MEM:91.7656MB
AES-256 | encrypt: 0.08720s CPU:4.12% MEM:91.8229MB | decrypt: 0.08140s
CPU:3.96% MEM:91.8385MB
DES | encrypt: 0.10000s CPU:4.44% MEM:91.5156MB | decrypt: 0.09636s
CPU:4.35% MEM:91.9792MB
--- Testing 48828 KB (50000000 characters) over 3 iterations ---
AES-128 | encrypt: 0.30100s CPU:6.62% MEM:244.3894MB | decrypt: 0.29519s
CPU:6.88% MEM:244.3542MB
AES-192 | encrypt: 0.33599s CPU:6.89% MEM:244.3750MB | decrypt: 0.33491s
CPU:6.88% MEM:244.3906MB
AES-256 | encrypt: 0.37312s CPU:7.09% MEM:244.3438MB | decrypt: 0.37083s
CPU:7.11% MEM:244.4115MB
DES | encrypt: 0.44109s CPU:7.54% MEM:244.0363MB | decrypt: 0.43869s
CPU:7.33% MEM:196.3882MB
--- Testing 97656 KB (100000000 characters) over 3 iterations ---
AES-128 | encrypt: 0.58429s CPU:7.50% MEM:434.2865MB | decrypt: 0.58903s
CPU:7.61% MEM:435.1771MB
AES-192 | encrypt: 0.66392s CPU:7.72% MEM:435.1094MB | decrypt: 0.65783s
CPU:7.79% MEM:435.1198MB
```

```
AES-256 | encrypt: 0.72097s CPU:7.99% MEM:435.1510MB | decrypt: 0.72875s
CPU:7.95% MEM:435.1510MB
DES | encrypt: 0.86776s CPU:8.11% MEM:434.7865MB | decrypt: 0.86345s
CPU:8.11% MEM:339.4062MB
--- Testing 195312 KB (200000000 characters) over 3 iterations ---
AES-128 | encrypt: 1.13412s CPU:8.16% MEM:815.8854MB | decrypt: 1.14449s
CPU:8.22% MEM:816.6198MB
AES-192 | encrypt: 1.27014s CPU:8.39% MEM:816.5886MB | decrypt: 1.26434s
CPU:8.40% MEM:816.6406MB
AES-256 | encrypt: 1.43012s CPU:8.46% MEM:816.5677MB | decrypt: 1.42999s
CPU:8.47% MEM:816.6927MB
DES | encrypt: 1.73317s CPU:8.52% MEM:816.3125MB | decrypt: 1.70558s
CPU:8.56% MEM:625.5729MB
```

Analiza algorytmów symetrycznych (AES, DES):

Czas szyfrowania i deszyfrowania:

Wszystkie algorytmy symetryczne są niezwykle szybkie, zwłaszcza w porównaniu do RSA. Czasy są mierzone w milisekundach nawet dla danych rzędu megabajtów.

Dla AES, czas szyfrowania i deszyfrowania jest bardzo zbliżony. Wraz ze wzrostem długości klucza (AES-128 do AES-256), czas przetwarzania nieznacznie wzrasta, co jest zgodne z oczekiwaniami, ponieważ dłuższe klucze wymagają więcej operacji.

DES, pomimo bycia starszym algorytmem, okazuje się zaskakująco konkurencyjny w kontekście czasu przetwarzania dla mniejszych rozmiarów danych. Jednak dla większych rozmiarów (np. 200 MB), staje się wolniejszy niż AES.

Zużycie procesora (CPU):

Algorytmy symetryczne cechują się relatywnie niskim zużyciem CPU dla mniejszych danych. Podobnie jak w poprzednich przypadkach, zużycie to rośnie wraz ze wzrostem danych, stabilizując się na poziomie około 7-8% dla AES i około 8-9% dla DES dla największych rozmiarów danych. Jest to efektywne wykorzystanie zasobów, które skaluje się liniowo.

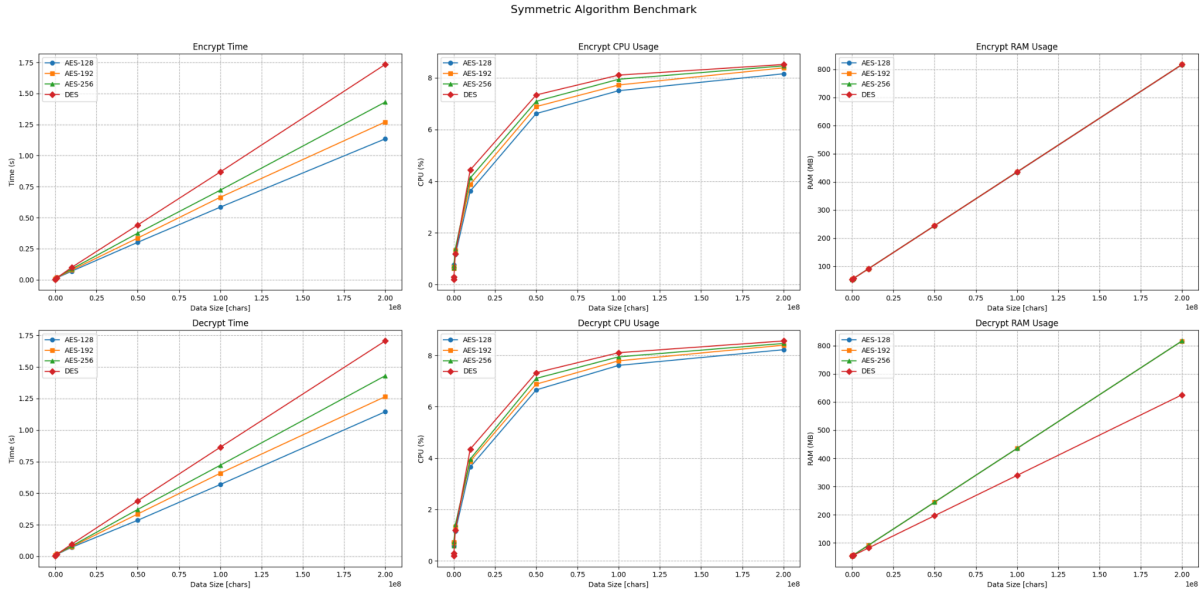
Zużycie pamięci (RAM):

Początkowe zużycie pamięci jest niskie (ok. 53 MB).

Zużycie RAM dla AES rośnie liniowo z rozmiarem danych, osiągając około 816 MB dla 200 MB danych.

Ciekawym spostrzeżeniem jest, że DES dla 200 MB danych wykazuje nieco niższe zużycie RAM podczas deszyfrowania (625 MB) w porównaniu do szyfrowania (816 MB) i algorytmów AES. Może to wynikać ze specyfiki implementacji paddingu i zarządzania pamięcią przez bibliotekę PyCryptodome dla DES w trybie ECB.

Wykresy dla Algorytmów Symetrycznych:



Wykresy dla algorytmów symetrycznych potwierdzają ich wysoką wydajność i liniową skalowalność. Widać, że AES, mimo różnych długości kluczy, działa bardzo podobnie pod względem szybkości, z niewielkimi różnicami na korzyść krótszych kluczy. DES jest również szybki, ale w miarę wzrostu danych staje się wolniejszy niż AES.

6. Wnioski

Przedstawione w niniejszym sprawozdaniu analizy algorytmów kryptograficznych oraz wyniki przeprowadzonych benchmarków dostarczają kompleksowego obrazu ich charakterystyki wydajnościowej i bezpieczeństwa. Głęboka interpretacja zebranych danych, w połączeniu z kontekstem teoretycznym, pozwala na wyciągnięcie szeregu kluczowych wniosków dotyczących praktycznych zastosowań, kompromisów projektowych oraz przyszłych wyzwań w dziedzinie kryptografii. Celem tego rozdziału jest nie tylko podsumowanie obserwacji, ale także ich pogłębiona analiza, wskazanie implikacji dla inżynierii bezpieczeństwa oraz zarysowanie perspektyw rozwoju.

6.1. Krytyczna Analiza Wydajności Algorytmów Kryptograficznych

Pomiary czasu, zużycia procesora (CPU) i pamięci (RAM) jednoznacznie uwidoczniły fundamentalne różnice w wydajności pomiędzy klasami algorytmów: klasycznymi, symetrycznymi i asymetrycznymi.

6.1.1. Dominacja Algorytmów Symetrycznych w Przetwarzaniu Danych

Wyniki benchmarków algorytmów symetrycznych, takich jak AES (128, 192, 256) oraz DES, wyraźnie wskazują na ich niezrównaną efektywność w szyfrowaniu i deszyfrowaniu dużych wolumenów danych.

Dla danych o rozmiarze 200 MB:

AES-128: Szyfrowanie około 1.13 s, deszyfrowanie około 1.14 s.

AES-256: Szyfrowanie około 1.43 s, deszyfrowanie około 1.43 s.

DES: Szyfrowanie około 1.73 s, deszyfrowanie około 1.70 s.

Te wyniki są rzędy wielkości lepsze niż dla algorytmu RSA-2048, który dla tej samej ilości danych potrzebował odpowiednio 350 s na szyfrowanie i ponad 2000 s na deszyfrowanie. Liniowa skalowalność czasów przetwarzania i zużycia pamięci dla AES oraz DES, przy stosunkowo niskim i stabilnym zużyciu CPU (około 7-9% dla dużych danych), potwierdza, że algorytmy te są doskonale zoptymalizowane do masowego przetwarzania. Implementacja w trybie EAX (AES) zapewnia dodatkowo uwierzytelnianie i integralność danych, co minimalizuje narzut związany z oddzielnym hashowaniem.

Nieznaczne różnice w wydajności między różnymi długościami klucza AES (128, 192, 256 bitów) są zgodne z teorią - dłuższe klucze wymagają większej liczby rund i/lub bardziej złożonych operacji w poszczególnych rundach, co minimalnie wydłuża czas przetwarzania. Z perspektywy praktycznej, różnice te są na tyle marginalne, że wybór AES-256 nad AES-128 jest uzasadniony przede wszystkim względami bezpieczeństwa (zwiększona odporność na ataki siłowe), a nie obawami o wydajność.

DES, mimo że historycznie istotny, okazał się wolniejszy od AES przy przetwarzaniu dużych bloków danych, co potwierdza jego stopniowe wycofywanie z zastosowań krytycznych. Należy podkreślić, że użycie trybu ECB w naszej implementacji DES (dla celów edukacyjnych i demonstracji różnic w wydajności) jest w praktyce odradzane ze względu na jego podatność na ataki wynikające z powtarzających się wzorców. W realnych systemach DES byłby używany w trybach takich jak CBC, co dodatkowo zwiększyłoby jego złożoność obliczeniową.

6.1.2. Koszt Obliczeniowy Kryptografii Asymetrycznej

Algorytm RSA-2048, jako reprezentant kryptografii asymetrycznej, wykazał drastycznie niższą wydajność w porównaniu do algorytmów symetrycznych. Długie czasy szyfrowania i deszyfrowania (szczególnie deszyfrowania) są bezpośrednim rezultatem złożoności obliczeniowej operacji na dużych liczbach pierwszych, stanowiących fundament jego bezpieczeństwa.

Dla 200 MB danych:

RSA-2048 Szyfrowanie: około 350 sekund (prawie 6 minut).

RSA-2048 Deszyfrowanie: około 2036 sekund (ponad 34 minuty).

Ta przepaść wydajnościowa jest kluczowa dla zrozumienia, dlaczego RSA i inne algorytmy asymetryczne nie są używane do szyfrowania całej zawartości komunikatów czy plików. Ich główna rola leży w mechanizmach, które wymagają unikalnych właściwości kluczy publicznych i prywatnych:

Bezpieczna wymiana kluczy symetrycznych: RSA jest wykorzystywany do szyfrowania krótkich, losowo generowanych kluczy sesyjnych, które następnie są używane przez algorytmy symetryczne do szyfrowania właściwych danych.

Podpisy cyfrowe: RSA służy do podpisywania skrótów wiadomości (hashy), co gwarantuje autentyczność i integralność, a co za tym idzie, niezaprzeczalność.

Uwierzytelnianie: Weryfikacja tożsamości w protokołach takich jak TLS/SSL.

Zużycie CPU dla RSA szybko osiąga wysoki poziom (około 9%), co świadczy o intensywnym charakterze obliczeń. Znaczące zużycie pamięci (do 750 MB dla szyfrowania) jest konsekwencją przetwarzania danych w "chunkach", co wynika z ograniczeń rozmiaru bloku w RSA (dla klucza 2048 bitów, maksymalny rozmiar danych do zaszyfrowania w jednym bloku to 190 bajtów dla paddingu OAEP). Operacje na tak dużej liczbie małych bloków generują narzut związany z wielokrotnym wywoływaniem funkcji kryptograficznych i zarządzaniem pamięcią dla każdego fragmentu.

6.1.3. Algorytmy Klasyczne - Szybkość Bez Bezpieczeństwa

Szyfry Cezara i Vigenère’a, choć historycznie ważne, w kontekście współczesnej wydajności i bezpieczeństwa odgrywają rolę wyłącznie dydaktyczną. Ich czasy szyfrowania i deszyfrowania są porównywalne z algorytmami symetrycznymi dla małych danych, ale ich fundamentalna słabość leży w niskim poziomie bezpieczeństwa.

Dla 200 MB danych:

Caesar: Szyfrowanie około 35.6 s, deszyfrowanie około 37.8 s.

Vigenere: Szyfrowanie około 42.5 s, deszyfrowanie około 42.8 s.

Szyfr Vigenère’a jest nieco wolniejszy od Cezara, co wynika z bardziej złożonej operacji na kluczu cyklicznym. Zużycie CPU i RAM wykazuje podobne trendy wzrostowe jak w przypadku AES/DES, osiągając blisko 9% CPU i kilkaset MB RAM dla dużych danych. To jednak szybkość osiągnięta kosztem bezpieczeństwa - oba szyfry są trywialnie łamane współczesnymi metodami kryptoanalizy (analiza częstotliwości, test Kasiski dla Vigenère’a). Wnioskiem jest, że szybkość nie jest jedynym kryterium wyboru algorytmu; bezpieczeństwo musi być zawsze nadrzędne.

6.2. Implikacje Praktyczne i Projektowanie Systemów Kryptograficznych

Wyniki benchmarków mają bezpośrednie przełożenie na sposób projektowania i implementacji bezpiecznych systemów informatycznych.

6.2.1. Architektura Hybrydowa jako Standard Przemysłowy

Dominacja algorytmów symetrycznych w szybkości przetwarzania i algorytmów asymetrycznych w mechanizmach wymiany kluczy i uwierzytelniania prowadzi do

naturalnego wniosku: nowoczesne systemy bezpieczeństwa muszą opierać się na architekturze hybrydowej. Ten model łączy zalety obu typów kryptografii:

Bezpieczna wymiana klucza sesyjnego: Algorytm asymetryczny (np. RSA lub częściej ECC ze względu na wydajność i bezpieczeństwo przy krótszych kluczach) jest używany do szyfrowania jednorazowego, losowo wygenerowanego klucza symetrycznego (tzw. klucza sesyjnego).

Szyfrowanie danych: Otrzymany klucz sesyjny jest następnie wykorzystywany przez szybki algorytm symetryczny (np. AES) do szyfrowania właściwej, dużej ilości danych.

Integralność i autentyczność: Dodatkowo, funkcje skrótu kryptograficznego (np. SHA-256) oraz algorytmy podpisów cyfrowych (np. RSA-DSA, ECDSA) są używane do zapewnienia integralności i autentyczności wiadomości.

Przykładem takiej architektury są protokoły TLS/SSL, które zabezpieczają komunikację w Internecie. Wyniki naszych pomiarów wyraźnie uzasadniają to podejście, pokazując, że próba szyfrowania całej sesji za pomocą wyłącznie RSA byłaby nierealna pod względem wydajności.

6.2.2. Wybór Algorytmu a Kontekst Zastosowania

Wybór konkretnego algorytmu musi być podyktowany specyficznymi wymaganiami aplikacji:

Systemy wymagające wysokiej przepustowości danych (np. VPN, szyfrowanie dysków, strumieniowanie mediów): AES jest de facto standardem. Jego doskonała wydajność i odporność na znane ataki czynią go idealnym wyborem. Długość klucza (128, 192, 256) powinna być dostosowana do wymaganego poziomu bezpieczeństwa (np. AES-256 dla danych niejawnych).

Systemy zarządzania kluczami, certyfikaty, podpisy cyfrowe (np. PKI, bankowość elektroniczna, kryptowaluty): Algorytmy asymetryczne są niezbędne. Ich wyższy koszt obliczeniowy jest akceptowalny, ponieważ operacje te są wykonywane rzadziej (np. raz na sesję) lub na małych danych (np. skróty wiadomości).

Starsze systemy lub specyficzne środowiska sprzętowe: DES (lub 3DES) może być wciąż spotykany, choć jego bezpieczeństwo jest wątpliwe. W naszej implementacji DES okazał się porównywalny z AES pod względem szybkości dla mniejszych danych, ale jego 56-bitowy klucz jest rażąco za krótki.

6.2.3. Optymalizacja Implementacji i Środowiska Wykonawczego

Pomiary zużycia CPU, które stabilizowały się na poziomie około 9% dla większości algorytmów (na MacBooku Pro M3 Pro z 12 rdzeniami), sugerują, że implementacja w Pythonie, nawet z wykorzystaniem bibliotek C (PyCryptodome), napotyka na ograniczenia związane z Python Global Interpreter Lock (GIL). GIL uniemożliwia pełne wykorzystanie wielu rdzeni procesora dla kodu Pythonowego w jednym procesie, co może tłumaczyć brak wzrostu zużycia CPU powyżej pewnego poziomu, nawet dla algorytmów intensywnych obliczeniowo. W przypadku krytycznych zastosowań, gdzie wydajność jest absolutnie kluczowa, rozważyć należy:

Implementacje w językach kompilowanych (C/C++): Pozwalają na pełne wykorzystanie sprzętowych instrukcji kryptograficznych (np. AES-NI w procesorach Intel/AMD) i operacje wielowątkowe.

Wykorzystanie GPU/FPGA: Dla ekstremalnie wysokiej przepustowości, akceleratory sprzętowe mogą znacząco zwiększyć wydajność operacji kryptograficznych.

Programowanie wieloprocesowe: Nasze wykorzystanie multiprocessing dla pomiarów jest właściwym podejściem do obejścia GIL, ale sam algorytm musi być zaprojektowany tak, aby móc działać równolegle na wielu rdzeniach lub procesach.

6.3. Wpływ Długości Klucza na Bezpieczeństwo i Wydajność

Długość klucza jest podstawowym parametrem, który ma bezpośredni wpływ zarówno na bezpieczeństwo, jak i na wydajność algorytmu kryptograficznego.

6.3.1. Długość Klucza a Bezpieczeństwo

Algorytmy symetryczne (AES): Wzrost długości klucza (np. z 128 do 256 bitów) wykładniczo zwiększa przestrzeń kluczy, co czyni ataki siłowe (brute force) praktycznie niemożliwymi. Przejście z 128 na 256 bitów zwiększa bezpieczeństwo o 2^128, co jest ogromnym skokiem. Obecnie AES-128 jest uważany za bezpieczny, ale AES-256 jest rekomendowany dla najbardziej wrażliwych danych i jako zabezpieczenie przed przyszłymi, bardziej zaawansowanymi atakami.

Algorytmy asymetryczne (RSA): Długość klucza (np. 2048 bitów) odnosi się do rozmiaru modułu n. Bezpieczeństwo RSA opiera się na trudności faktoryzacji dużych liczb. Im dłuższy klucz, tym trudniej przeprowadzić atak faktoryzacji. Dziś klucze RSA o długości 2048 bitów są standardem, a 3072 bity stają się coraz częściej zalecane. Długości 1024 bity są już uznawane za niebezpieczne.

Algorytmy klasyczne: Długość "klucza" (przesunięcie Cezara, słowo kluczowe Vigenère'a) jest znikoma w porównaniu do wymagań współczesnej kryptografii. Przestrzeń klucza jest na tyle mała, że atak siłowy jest trywialny.

6.3.2. Długość Klucza a Wydajność

Algorytmy symetryczne: Jak zaobserwowano w przypadku AES, wzrost długości klucza powoduje minimalny spadek wydajności. Jest to akceptowalny kompromis, biorąc pod uwagę znaczny wzrost bezpieczeństwa. Ten niewielki spadek wynika z większej liczby rund lub bardziej złożonych operacji kluczowych.

Algorytmy asymetryczne: Wzrost długości klucza w algorytmach asymetrycznych (np. z 1024 do 2048 bitów) skutkuje znaczącym spadkiem wydajności. Operacje na większych liczbach są z natury bardziej kosztowne obliczeniowo. Dlatego wybór odpowiedniej długości klucza RSA jest kompromisem między bezpieczeństwem a akceptowalnym kosztem obliczeniowym.

6.4. Znaczenie Kontekstu Implementacji: PyCryptodome i Python

Wykorzystanie biblioteki PyCryptodome w języku Python miało istotny wpływ na uzyskane wyniki:

Wydajność: PyCryptodome to dobrze zoptymalizowana biblioteka, która dla operacji kryptograficznych korzysta z kodu zaimplementowanego w C/C++. Dzięki temu algorytmy symetryczne i asymetryczne (szczególnie AES) osiągnęły znacznie lepsze czasy niż czyste implementacje w Pythonie. Mimo to, narzut związany z interfejsem Python-C oraz wspomniane ograniczenia GIL mogą wpływać na ogólne zużycie CPU i skalowalność w aplikacjach wielowątkowych.

Pamięć: Obserwowane zużycie pamięci, które rosło liniowo z rozmiarem danych, jest typowe dla przetwarzania dużych bloków danych w Pythonie. Każdy blok danych wejściowych, pośrednich i wyjściowych musi być przechowywany w pamięci, co przy danych rzędu setek megabajtów naturalnie prowadzi do zużycia kilkuset megabajtów RAM. Implementacja RSA z chunkowaniem również zwiększa zapotrzebowanie na pamięć, ponieważ wyniki pośrednie z każdego "chunka" muszą być przechowywane przed ich połączeniem.

Klasyczne algorytmy: Implementacja szyfrów Cezara i Vigenère'a w czystym Pythonie (operacje na łańcuchach znaków) jest mniej efektywna niż implementacje w C, co

widać po ich czasach dla dużych danych. Ich liniowa skalowalność jest jednak zachowana, co wynika z prostoty ich operacji.

6.5. Perspektywy Rozwoju i Wyzwania Przyszłości

Wyniki obecnych benchmarków, choć miarodajne, stanowią jedynie migawkę w dynamicznie zmieniającym się świecie kryptografii. Kluczowe wyzwania i kierunki rozwoju obejmują:

6.5.1. Kryptografia Postkwantowa (PQC)

Największym zagrożeniem dla obecnych algorytmów asymetrycznych (RSA, ECC) jest rozwój komputerów kwantowych. Algorytmy takie jak Shor's algorithm są teoretycznie zdolne do złamania RSA poprzez efektywną faktoryzację dużych liczb i dyskretny logarytm w czasie wielomianowym. Algorytmy symetryczne (np. AES) są mniej zagrożone, ponieważ atak kwantowy na nie (algorytm Grovera) skraca efektywną długość klucza o połowę, co oznacza, że AES-256 byłby równoważny bezpieczeństwu AES-128 na komputerze klasycznym. W odpowiedzi na to zagrożenie trwają intensywne prace nad kryptografią postkwantową (PQC), której algorytmy są odporne na ataki kwantowe. Wyzwania to:

Wydajność: Nowe algorytmy PQC są często wolniejsze i wymagają większych kluczy niż ich klasyczne odpowiedniki, co może wprowadzić nowe kompromisy wydajnościowe. Nasze obecne benchmarki mogą posłużyć jako punkt odniesienia do porównania wydajności PQC w przyszłości.

Standardyzacja: Proces standaryzacji algorytmów PQC przez NIST jest w toku, a przyjęcie nowych standardów będzie kluczowe dla ich globalnego wdrożenia.

6.5.2. Kryptografia Homomorficzna i Zero-Knowledge Proofs

Nowe gałęzie kryptografii, takie jak kryptografia homomorficzna (FHE - Fully Homomorphic Encryption) i Zero-Knowledge Proofs (ZKP), otwierają możliwości przetwarzania danych w chmurze bez konieczności ich deszyfrowania lub udowadniania faktu bez ujawniania samej informacji. Obecnie są one bardzo kosztowne obliczeniowo, ale ich rozwój ma potencjał zrewolucjonizowania prywatności danych. Badania wydajnościowe tych algorytmów będą kluczowe dla ich praktycznego zastosowania.

6.5.3. Bezpieczeństwo Sprzętowe i Enklawy Zaufania

Współczesna kryptografia coraz częściej przenosi się do sprzętu. Instrukcje AES-NI, moduły TPM (Trusted Platform Module) czy enklawy zaufania (np. Intel SGX, ARM TrustZone) oferują zwiększone bezpieczeństwo poprzez izolowanie operacji kryptograficznych od reszty systemu operacyjnego. Nasze benchmarki, choć uruchomione na

M3 Pro, mogłyby zostać poszerzone o analizę wpływu takich sprzętowych mechanizmów na wydajność.

6.5.4. Audyty Bezpieczeństwa i Kryptoanaliza

Kryptografia to ciągła gra między projektantami algorytmów a kryptoanalitykami. Stale rozwijane są nowe metody ataków, co zmusza do rewizji i ewentualnego zastępowania dotychczasowych standardów. Nasze sprawozdanie podkreśla, dlaczego algorytmy takie jak DES czy szyfry klasyczne stały się niebezpieczne - ich analiza kryptograficzna wykazała podatności, które uniemożliwiają ich dalsze bezpieczne stosowanie. Regularne audyty i testy penetracyjne systemów wykorzystujących kryptografię są więc nieodzowne.

6.6. Podsumowanie Wniosków

Przeprowadzone badanie w sposób empiryczny potwierdziło fundamentalne zasady kryptografii, ukazując złożoną zależność między bezpieczeństwem, wydajnością i specyfiką zastosowania algorytmów.

Kryptografia symetryczna (AES) pozostaje złotym standardem dla szybkiego i bezpiecznego szyfrowania dużych ilości danych, będąc filarem większości współczesnych systemów komunikacji i przechowywania danych. Jej liniowa skalowalność i wysoka wydajność, nawet przy rosnących wymaganiach bezpieczeństwa (dłuższe klucze), czynią ją niezastąpioną.

Kryptografia asymetryczna (RSA), choć znacznie wolniejsza, jest kluczowa dla bezpiecznej wymiany kluczy, uwierzytelniania i niezaprzeczalności w otwartych sieciach. Jej koszt obliczeniowy wymaga strategicznego wykorzystania w architekturach hybrydowych.

Algorytmy klasyczne są dowodem na ewolucję kryptografii i podkreślają, że bezpieczeństwo algorytmu jest znacznie ważniejsze niż jego teoretyczna prostota czy historyczna odporność. Dziś są one reliktem przeszłości, nie mającym zastosowania w praktycznych systemach bezpieczeństwa.

Implementacja w Pythonie z PyCryptodome wykazała dobrą wydajność dla algorytmów bazujących na kodzie C, jednakże ograniczenia środowiska wykonawczego (np. GIL) mogą wpływać na maksymalne wykorzystanie zasobów systemowych, zwłaszcza w kontekście wielordzeniowych procesorów.

Przyszłość kryptografii to nieustanne poszukiwanie nowych rozwiązań w obliczu pojawiających się zagrożeń, takich jak komputery kwantowe. Rozwój kryptografii postkwantowej oraz zaawansowanych technik zwiększających prywatność (homomorficzne szyfrowanie, ZKP) będzie kształtował krajobraz bezpieczeństwa w nadchodzących dekadach.

Wnioski te stanowią solidną podstawę dla zrozumienia i projektowania bezpiecznych systemów informatycznych, podkreślając jednocześnie, że dziedzina kryptografii jest dynamicznym obszarem, który wymaga ciągłej uwagi, badań i adaptacji do zmieniającego się środowiska technologicznego i zagrożeń. Inżynierowie i badacze muszą nieustannie ważyć kompromisy między wydajnością, bezpieczeństwem i funkcjonalnością, aby sprostać wyzwaniom cyfrowego świata.