

Raport z Projektu: Zaawansowana Analiza Keccak-f

Przedmiot: Kryptografia

Data: 22 stycznia 2026

Deadliny projektowe: 10 grudnia 2025 | 31 grudnia 2025 | 15 stycznia 2026

Zespół Projektowy

Rola	Imię i nazwisko	Odpowiedzialność
Lider zespołu	Mateusz Szelecki	Koordinacja projektu, moduł Avalanche, moduł Encryption
Członek zespołu	Kacper Rothkegel	Moduł Avalanche, moduł Attack
Członek zespołu	Radosław Chruściński	Moduł Attack

1. Wprowadzenie

1.1 Cel Projektu

Celem projektu była zaawansowana analiza kryptograficzna algorytmu **Keccak-f**, stanowiącego rdzeń standardu SHA-3. Projekt skupił się na trzech fundamentalnych aspektach bezpieczeństwa kryptograficznego:

- Efekt lawinowy (Avalanche Effect)** – analiza propagacji zmian bitowych przez rundy permutacji
- Wizualizacja szyfrowania** – obserwacja transformacji danych wizualnych przez operacje Keccak
- Ataki kryptograficzne** – badanie odporności na atak urodzinowy (birthday attack)

1.2 Architektura Rozwiązania

Projekt został zaimplementowany w języku **Python 3** z wykorzystaniem GUI opartego na **Tkinter**. Architektura składa się z:

```
kryptografia-projekt/
├─ main.py                # Launcher aplikacji z prezentacjami slajdów
├─ keccak.py              # Implementacja KeccakSponge (konstrukcja
gąbki)
├─ operations.py          # Operacje permutacji:  $\theta, \rho, \pi, \chi, \iota$ 
├─ modules/
│   └─ avalanche.py       # Moduł analizy efektu lawinowego
│   └─ encryption.py      # Moduł wizualizacji szyfrowania
│   └─ attack.py          # Moduł symulacji ataków
└─ assets/                # Grafiki i zasoby wizualne
```

2. Implementacja Rdzenia Keccak

2.1 Klasa KeccakSponge (keccak.py)

Centralnym elementem projektu jest implementacja konstrukcji gąbkowej (sponge construction) zgodna ze specyfikacją NIST FIPS 202.

Parametry konfiguracji:

- **rate** – liczba bitów wchłanianych w jednym cyklu
- **capacity** – liczba bitów "zarezerwowanych" dla bezpieczeństwa
- **w** – szerokość lane'a (typowo 64 dla SHA-3-256)
- **rounds** – liczba rund permutacji (24 dla pełnego Keccak)

Kluczowe metody:

Metoda	Opis
wchlanianie(input_bytes)	Faza absorpcji – XOR wiadomości ze stanem i permutacja
wyciskanie(output_length)	Faza wyciskania – ekstrakcja skrótu z odpowiednim paddingiem
keccak_f(callback)	Wykonanie pełnej permutacji z opcjonalnym callbackiem po każdym kroku
wykonaj_pojedyncza_runde(nr, callback)	Wykonanie jednej rundy ($\theta \rightarrow \rho \rightarrow \pi \rightarrow \chi \rightarrow \iota$)
encrypt_stream(key, data, iv)	Szyfrowanie strumieniowe w trybie Duplex

Fragment implementacji permutacji:

```
def keccak_f(self, callback=None):
    for i in range(self.rounds):
        self.wykonaj_pojedyncza_runde(i, callback)

def wykonaj_pojedyncza_runde(self, numer_rundy, callback=None):
    self.state = theta(self.state, self.w)
    if callback: callback("Theta", self.state)

    self.state = rho(self.state, self.w)
    if callback: callback("Rho", self.state)

    self.state = pi(self.state, self.w)
    if callback: callback("Pi", self.state)

    self.state = chi(self.state, self.w)
    if callback: callback("Chi", self.state)

    self.state = iota(self.state, numer_rundy, self.w)
    if callback: callback("Iota", self.state)
```

2.2 Operacje Permutacji (operations.py)

Implementacja pięciu operacji permutacji Keccak-f zgodnie ze specyfikacją NIST:

Operacja	Symbol	Funkcja kryptograficzna
Theta (θ)	Dyfuzja	XOR z sumą parytetu sąsiednich kolumn
Rho (ρ)	Przesunięcie	Rotacja bitowa wzdłuż osi Z
Pi (π)	Permutacja	Przestawienie pozycji (x,y) → (y, 2x+3y)
Chi (χ)	Konfuzja	Nieliniowa operacja: $A \oplus ((\neg B) \wedge C)$
Iota (ι)	Asymetria	XOR ze stałą rundy RC

3. Moduł 1: Efekt Lawinowy (avalanche.py)

Autorzy: Mateusz Szelecki, Kacper Rothkegel

3.1 Funkcjonalność

Moduł wizualizuje kluczową właściwość kryptograficzną – **efekt lawinowy** – gdzie zmiana pojedynczego bitu wejścia powinna skutkować zmianą ~50% bitów wyjścia.

Główne funkcje:

- **Symulacja wielowariantowa** – jednoczesne porównanie do 6 konfiguracji parametrów
- **Wykres Hamminga** – wizualizacja procentowej zmiany bitów runda po rundzie
- **Predefiniowane scenariusze badawcze** – gotowe zestawy testowe
- **Interaktywna konfiguracja** – modyfikacja liczby rund, szerokości stanu, wiadomości

3.2 Scenariusze Badawcze

Scenariusz	Cel	Parametry
1. Wpływ rozmiaru stanu	Analiza jak szerokość w wpływa na dyfuzję	$w \in \{8, 16, 32, 64, 128, 256\}$, rundy=24
2. Szybkość dyfuzji	Obserwacja propagacji przy różnej liczbie rund	$rundy \in \{3, 6, 8, 12, 18, 24\}$, $w=64$
3. Odporność na wzorce	Testowanie z różnymi typami wejść	Pusty, same zera, jedynki, wzorce, losowe

3.3 Implementacja Analizy

```
def run_simulation(self, rounds, w, msg):
    rate, cap = self.calculate_params(w)

    # Dwa stany różniące się 1 bitem
    s1 = KeccakSponge(rate, cap, w, rounds)
    s2 = KeccakSponge(rate, cap, w, rounds)

    m1 = bytearray(msg_bytes)
    m2 = bytearray(msg_bytes or b"\x00")
    m2[-1] ^= 1 # Flip ostatniego bitu

    s1.xorowanie_do_stanu(m1)
    s2.xorowanie_do_stanu(m2)

    dist = [self.calculate_hamming_diff(s1.state, s2.state, w)]

    for r in range(rounds):
        s1.wykonaj_pojedyncza_runde(r)
        s2.wykonaj_pojedyncza_runde(r)
```

```
dist.append(self.calculate_hamming_diff(s1.state, s2.state, w))

return dist, 25 * w, hexhash
```

3.4 Wizualizacja

Wykres przedstawia:

- **Oś X** – numer rundy (0 do liczby rund)
 - **Oś Y** – procent zmienionych bitów (0–65%)
 - **Linia referencyjna** – 50% (idealna dyfuzja)
 - **Zoom 40–60%** – szczegółowy widok obszaru konwergencji
-

4. Moduł 2: Szyfrowanie Obrazów (`encryption.py`)

Autor: Mateusz Szelecki

4.1 Funkcjonalność

Moduł oferuje wizualizację procesu szyfrowania krok po kroku na obrazie szachownicy, umożliwiając obserwację wpływu każdej operacji Keccak na dane wizualne.

Główne funkcje:

- **Interaktywny suwak** – nawigacja przez 15 kroków szyfrowania (3 rundy × 5 operacji)
- **Wizualizacja stanu wewnętrznego** – obraz przed/po każdej operacji
- **Klucz szyfrowania** – możliwość wprowadzenia hasła
- **Szczegółowe opisy** – wyjaśnienie każdej operacji wizualnie

4.2 Architektura Przetwarzania

Obraz szachownicy (100×100 px, 8-bit grayscale)

↓

Podział na bloki 25 bajtów (= rozmiar stanu 5×5×8)

↓

XOR z kluczem (jeśli podany)

↓

Kolejne operacje Keccak:

Stan 0: Inicjalizacja

Runda 0: Theta → Rho → Pi → Chi → Iota

Runda 1: Theta → Rho → Pi → Chi → Iota

Runda 2: Theta → Rho → Pi → Chi → Iota

↓

Wizualizacja stanu jako obraz 500×500 px

4.3 Opisy Operacji dla Użytkownika

Moduł zawiera szczegółowe, edukacyjne opisy każdej operacji:

Operacja	Efekt wizualny	Wyjaśnienie
Theta	Szachownica zanika/"szarzeje"	Każdy piksel "zaciąga" informację od 10 sąsiadów
Rho	Zmiana odcieni (kształty zostają)	Rotacja bitów wewnątrz bajtu
Pi	"Teleportacja" kwadratów	Fizyczne przeniesienie bajtu w inne miejsce macierzy
Chi	Całkowita utrata przewidywalności	Nieliniowa funkcja – obraz zaczyna przypominać szum
Iota	Minimalna zmiana (często niewidoczna)	XOR ze stałą rundy w rogu macierzy

4.4 Mechanizm Callback

Wykorzystano system callback do zatrzymania przetwarzania w dowolnym momencie:

```
def process_image(self):
    class StopProcessing(Exception): pass

    def my_callback(name, state):
        nonlocal current_step_counter
        current_step_counter += 1
        if current_step_counter >= target_steps:
            raise StopProcessing

    try:
        k.keccak_f(callback=my_callback)
    except StopProcessing:
        pass # Zatrzymano w żądanym kroku
```

5. Moduł 3: Atak Kolizyjny (attack.py)

5.1 Funkcjonalność

Moduł symuluje **atak urodzinowy** (birthday attack) w celu znalezienia kolizji – dwóch różnych wiadomości dających identyczny skrót.

Główne funkcje:

- **Konfigurowalne parametry** – szerokość stanu, liczba rund, rozmiar wejścia/wyjścia
- **Wykres prawdopodobieństwa** – wizualizacja teoretycznego vs rzeczywistego prawdopodobieństwa kolizji
- **Log w czasie rzeczywistym** – szczegóły znalezionej kolizji
- **Asynchroniczne przetwarzanie** – atak działa w osobnym wątku

5.2 Parametry Ataku

Parametr	Domyślna wartość	Opis
Szerokość (w)	64	Szerokość lane'a w bitach
Rundy	24	Liczba rund permutacji
Wejście (B)	16	Rozmiar losowej wiadomości w bajtach
Wyjście (B)	2	Rozmiar skrótu w bajtach (dla szybkiej kolizji)

5.3 Algorytm Ataku

```
def attack_worker(self, cfg):
    seen_hashes = {}
    attempts = 0

    while self.is_running:
        attempts += 1
        rand_input = os.urandom(cfg['in_size'])

        k = KeccakSponge(rate, cap, cfg['w'], cfg['rounds'])
        k.wchlanianie(rand_input)
        digest = k.wyciskanie(cfg['out_size'])

        if digest in seen_hashes:
            if seen_hashes[digest] != rand_input:
                # KOLIZJA ZNALEZIONA!
```

```
self.finish_attack(rand_input, seen_hashes[digest], diges
return
```

```
seen_hashes[digest] = rand_input
```

5.4 Prawdopodobieństwo Kolizji (Birthday Paradox)

Moduł oblicza teoretyczne prawdopodobieństwo kolizji według wzoru:

$$P(n, N) \approx 1 - e^{-\frac{n^2}{2N}}$$

gdzie:

- **n** – liczba prób
- **N** – przestrzeń możliwych skrótów ($2^{(\text{hash_bytes} \times 8)}$)

Dla 2-bajтового skrótu ($N = 65536$), 50% prawdopodobieństwo kolizji występuje przy ~300 próbach.

6. Interfejs Użytkownika (`main.py`)

6.1 Launcher

Główne okno prezentuje trzy moduły jako duże przyciski z opcjonalną prezentacją slajdów przed uruchomieniem każdego modułu.

6.2 Funkcje Dodatkowe

- **Tryb timera** (`--time`) – 90-sekundowy licznik dla prezentacji
 - **Prezentacje slajdów** – wprowadzenie do każdego modułu
 - **Dokumentacja PDF** – przycisk szybkiego dostępu
-

7. Wykorzystane Technologie i Biblioteki

Biblioteka	Zastosowanie
<code>tkinter</code> / <code>ttk</code>	Interfejs graficzny
<code>matplotlib</code>	Wykresy (Avalanche, Attack)
<code>PIL</code> (Pillow)	Przetwarzanie obrazów (Encryption)

Biblioteka	Zastosowanie
<code>threading</code>	Asynchroniczne ataki
<code>os.urandom</code>	Generowanie losowych wiadomości
<code>math</code>	Obliczenia prawdopodobieństwa

8. Proces Prac i Harmonogram

8.1 Etap I (do 10 grudnia 2025)

- Implementacja rdzenia `keccak.py` i `operations.py`
- Testy jednostkowe operacji permutacji
- Szkielet GUI

8.2 Etap II (do 31 grudnia 2025)

- Moduł Avalanche – pełna funkcjonalność
- Moduł Encryption – podstawowa wizualizacja
- Integracja z launcherem

8.3 Etap III (do 15 stycznia 2026)

- Moduł Attack – symulacja ataków
- Scenariusze badawcze
- Dokumentacja i prezentacje slajdów

9. Napotkane Problemy i Rozwiązania

9.1 Problem: Wydajność przy dużych stanach

Opis: Dla $w=256$ symulacje trwały zbyt długo.

Rozwiązanie: Cache wyników (`results_cache`) i optymalizacja pętli.

9.2 Problem: Wizualizacja kroków pośrednich

Opis: Trudność zatrzymania permutacji w dowolnym momencie.

Rozwiązanie: System callback z wyjątkiem `StopProcessing`.

9.3 Problem: Blokowanie GUI podczas ataków

Opis: Długotrwałe poszukiwanie kolizji zamrażało interfejs.

Rozwiązanie: Przeniesienie ataku do osobnego wątku (`threading.Thread`).

9.4 Problem: Skalowanie fontów na wykresach

Opis: Nieczytelne etykiety przy małych oknach.

Rozwiązanie: Dynamiczne skalowanie fontów (`scale_fonts`).

10. Wnioski

10.1 Efekt Lawinowy

- Pełna dyfuzja (~50% zmian) osiągnięta po **3-5 rundach** dla standardowych parametrów
- Mniejsza szerokość stanu (`w`) przyspiesza dyfuzję, ale zmniejsza bezpieczeństwo
- Wzorce wejściowe (same zera, jedynki) nie wpływają znacząco na szybkość dyfuzji

10.2 Wizualizacja Szyfrowania

- Operacja **Chi** ma największy wpływ wizualny (wprowadza nieliniowość)
- Operacja **Pi** fizycznie "tasuje" piksele, niszcząc strukturę
- Po 2-3 rundach obraz jest nierozpoznawalny od szumu

10.3 Odporność na Ataki

- Dla pełnych 24 rund z 256-bitowym skrótem atak jest niepraktyczny
 - Zredukowane wersje (2-bajtowy skrót) pozwalają na demonstrację ataku urodzinowego
 - Teoretyczne prawdopodobieństwa zgadzają się z wynikami eksperymentalnymi
-

11. Możliwe Rozszerzenia

1. **Dodatkowe warianty SHA-3** – SHAKE128, SHAKE256 z różnymi długościami wyjścia
 2. **Analiza różnicowa** – śledzenie propagacji różnic przez operacje
 3. **Wizualizacja 3D stanu** – macierz $5 \times 5 \times w$ jako sześćcian
 4. **Porównanie z innymi hashami** – SHA-2, MD5 dla kontekstu
 5. **Atak preimage** – wyszukiwanie wiadomości dla danego skrótu
-

Przygotował: Zespół projektowy

Kontakt: Mateusz Szelecki (lider)