UNIWERSYTET KOMISJI EDUKACJI NARODOWEJ
W KRAKOWIE

Instytut Bezpieczeństwa i Informatyki

SPRZĘTOWE ASPEKTY CYBERBEZPIECZEŃSTWA

Ćwiczenie numer: 5

Temat: Podstawy Simple Power Analysis (SPA) – symulacja

Autorzy:

Imię i Nazwisko: Emilia Zaręba

Imię i Nazwisko: Katarzyna Zieleniewska

Numer grupy: L3

Spis treści

1. Wstęp

1.1 Cel i zakres projektu

Celem niniejszego projektu jest zapoznanie się z podstawami ataków typu *side-channel*, a w szczególności z atakiem **Simple Power Analysis (SPA)**, poprzez wykonanie jego symulacji w środowisku laboratoryjnym. Projekt ma charakter praktyczny i polega na przeprowadzeniu symulowanego ataku mocy na prostą implementację algorytmu kryptograficznego oraz analizie uzyskanych przebiegów poboru mocy.

W ramach projektu zrealizowano pełny scenariusz obejmujący identyfikację podatności implementacyjnej, demonstrację ataku SPA oraz wdrożenie mechanizmów obronnych mających na celu utrudnienie lub uniemożliwienie skutecznej analizy poboru mocy. Projekt został wykonany na dwóch różnych systemach operacyjnych: **Windows oraz Linux**, co pozwala na pokazanie, że podatność na atak SPA wynika z implementacji algorytmu, a nie z właściwości konkretnego systemu operacyjnego.

1.2. Zakres ćwiczenia i przyjęte założenia

Zakres ćwiczenia obejmuje przygotowanie środowiska laboratoryjnego, implementację prostego algorytmu kryptograficznego podatnego na atak SPA oraz symulację poboru mocy w trakcie jego wykonywania. Pobór mocy został zasymulowany na podstawie uproszczonego modelu, w którym zakłada się zależność zużycia energii od liczby jedynek w przetwarzanych danych, z uwzględnieniem losowego szumu.

W ramach ćwiczenia nie wykonywano rzeczywistych pomiarów sprzętowych. Cała analiza została przeprowadzona w formie symulacji programowej, co pozwala skupić się na zrozumieniu mechanizmu ataku SPA oraz jego wpływu na bezpieczeństwo implementacji algorytmów kryptograficznych. Przyjęto założenie, że algorytm o tej samej logice działania, uruchamiany na różnych systemach operacyjnych, powinien wykazywać analogiczne charakterystyki przebiegów poboru mocy, mimo drobnych różnic implementacyjnych wynikających ze środowiska uruchomieniowego.

2. Podstawy teoretyczne

2.1 Ataki typu Side-Channel

Ataki typu *side-channel* (kanałów bocznych) to klasa ataków, które nie polegają na łamaniu samego algorytmu kryptograficznego, lecz na analizie dodatkowych informacji ujawnianych

podczas jego wykonywania. Informacje te mogą pochodzić m.in. z czasu wykonania operacji, poboru mocy, emisji elektromagnetycznej lub dźwięków generowanych przez urządzenie.

W przeciwieństwie do klasycznych ataków kryptograficznych, ataki typu side-channel wykorzystują fakt, że fizyczna realizacja algorytmu może nieświadomie ujawniać informacje o przetwarzanych danych lub kluczu kryptograficznym. Oznacza to, że nawet algorytmy uznawane za bezpieczne od strony matematycznej mogą być podatne na ataki, jeśli ich implementacja nie uwzględnia odpowiednich zabezpieczeń.

Ataki tego typu są szczególnie istotne w kontekście systemów wbudowanych, kart inteligentnych, urządzeń IoT oraz modułów kryptograficznych, gdzie ograniczenia sprzętowe często utrudniają stosowanie zaawansowanych mechanizmów ochrony.

2.2 Simple Power Analysis (SPA)

Simple Power Analysis (SPA) jest jednym z najprostszych ataków typu side-channel, który polega na bezpośredniej analizie pojedynczego przebiegu poboru mocy urządzenia w trakcie wykonywania operacji kryptograficznych. W odróżnieniu od bardziej zaawansowanych metod, takich jak Differential Power Analysis (DPA), atak SPA nie wymaga analizy statystycznej wielu przebiegów ani skomplikowanych obliczeń.

W ataku SPA analizowany jest kształt przebiegu poboru mocy w czasie. Różnice w wykonywanych instrukcjach, takich jak operacje logiczne, instrukcje warunkowe czy pętle, mogą powodować zauważalne zmiany w poborze mocy. Na tej podstawie atakujący może wyciągać wnioski dotyczące przebiegu algorytmu, a w niektórych przypadkach także wartości klucza kryptograficznego.

SPA jest szczególnie skuteczny w przypadku implementacji, które wykonują różne operacje w zależności od wartości klucza lub danych, np. poprzez instrukcje warunkowe. Z tego powodu atak ten jest często wykorzystywany jako przykład pokazujący, jak istotne znaczenie ma bezpieczna implementacja algorytmów kryptograficznych.

2.3 Model poboru mocy

Na potrzeby symulacji ataku SPA w projekcie zastosowano uproszczony model poboru mocy oparty na tzw. **Hamming Weight Model**. Model ten zakłada, że pobór mocy układu jest w przybliżeniu proporcjonalny do liczby bitów o wartości „1” w przetwarzanej danej lub w zawartości rejestru procesora.

Przykładowo, jeśli dana binarna zawiera większą liczbę jedynek, to jej przetwarzanie powoduje większy pobór mocy niż w przypadku danych zawierających mniej jedynek. W praktyce rzeczywisty pobór mocy jest znacznie bardziej złożony, jednak model Hamming

Weight jest często stosowany w analizach edukacyjnych i symulacjach ze względu na swoją prostotę i czytelność.

W projekcie do modelu poboru mocy dodano również losowy składnik szumu, który ma na celu lepsze odwzorowanie rzeczywistych warunków pracy układu elektronicznego oraz pokazanie wpływu zakłóceń na analizę przebiegów.

2.4. Ograniczenia i założenia symulacji

Przeprowadzona symulacja ataku SPA ma charakter edukacyjny i została oparta na uproszczonych założeniach. W ramach projektu nie wykonywano rzeczywistych pomiarów poboru mocy sprzętu, a wszystkie przebiegi zostały wygenerowane programowo na podstawie przyjętego modelu.

Symulacja nie uwzględnia wielu czynników występujących w rzeczywistych systemach, takich jak złożona architektura procesora, równoległe wykonywanie instrukcji, mechanizmy buforowania czy wpływ temperatury. Pomimo tych uproszczeń, symulacja pozwala w czytelny sposób zaprezentować podstawową ideę ataku SPA oraz jego wpływ na bezpieczeństwo implementacji algorytmu kryptograficznego.

Celem ćwiczenia nie jest dokładne odwzorowanie fizycznych właściwości urządzeń, lecz zrozumienie mechanizmu ataku oraz pokazanie, w jaki sposób niewłaściwa implementacja może prowadzić do ujawnienia informacji wrażliwych. Przyjęte założenia umożliwiają łatwe przeprowadzenie analizy oraz skupienie się na istocie problemu.

3. Środowisko laboratoryjne

3.1 Wykorzystany sprzęt

Do realizacji projektu wykorzystano standardowy komputer przenośny przeznaczony do pracy studenckiej. Ze względu na symulacyjny charakter ćwiczenia nie było wymagane użycie specjalistycznego sprzętu pomiarowego ani dodatkowych urządzeń zewnętrznych. Wszystkie eksperymenty zostały przeprowadzone lokalnie na komputerze z wykorzystaniem środowiska programistycznego.

3.2 Wykorzystane oprogramowanie

Na potrzeby symulacji ataku Simple Power Analysis wykorzystano następujące oprogramowanie:

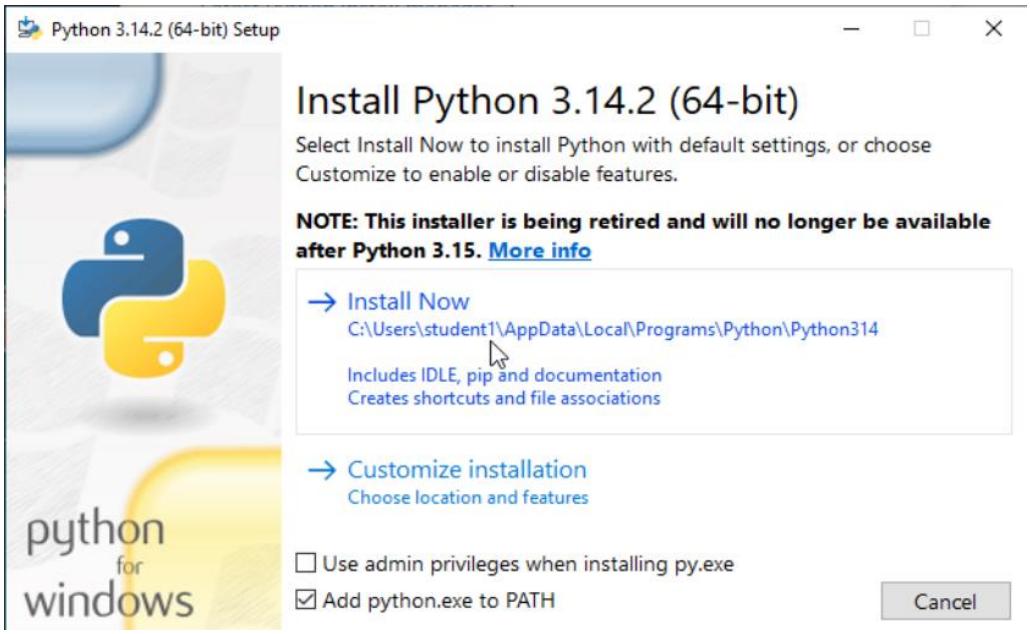
- język programowania **Python** (wersja 3.14.2),
- biblioteki:

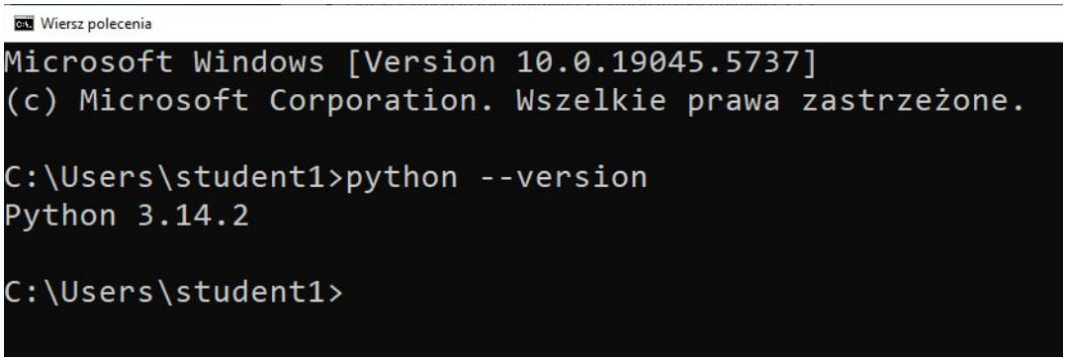
    - **NumPy** – do obliczeń numerycznych,
    - **Matplotlib** – do generowania wykresów przebiegów poboru mocy,
- terminal systemowy / środowisko IDE do uruchamiania skryptów.

To samo rozwiązanie programowe zostało uruchomione na dwóch systemach operacyjnych w celu porównania wyników i potwierdzenia niezależności podatności od platformy systemowej.

3.3 Konfiguracja środowiska

- Windows
    - Instalacja Pythona





    - Instalacja bibliotek

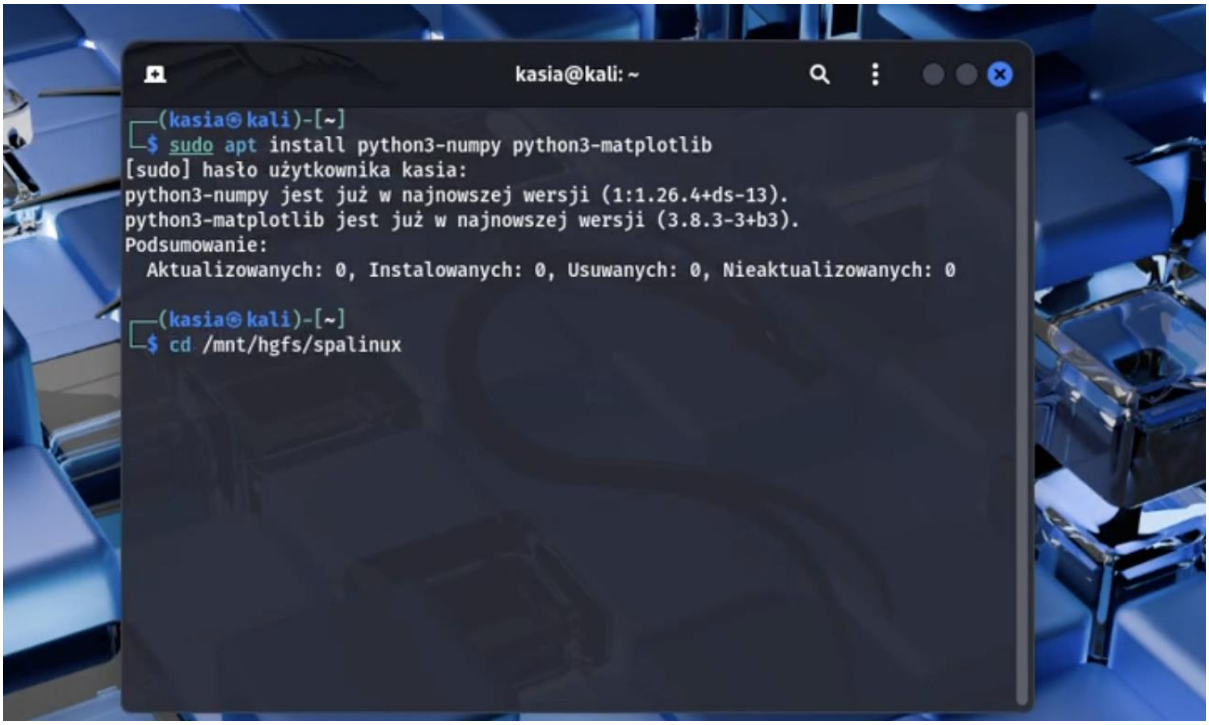
```
C:\Users\student1>pip install numpy matplotlib
Installing collected packages: six, pyparsing, pillow, packaging, numpy
, kiwisolver, fonttools, cycler, python-dateutil, contourpy, matplotlib

Successfully installed contourpy-1.3.3 cycler-0.12.1 fonttools-4.61.1 k
iwisolver-1.4.9 matplotlib-3.10.8 numpy-2.4.0 packaging-25.0 pillow-12.
1.0 pyparsing-3.3.1 python-dateutil-2.9.0.post0 six-1.17.0
```

```
C:\Users\student1>pip list
Package         Version
--------------- -----------
contourpy       1.3.3
cycler          0.12.1
fonttools       4.61.1
kiwisolver      1.4.9
matplotlib      3.10.8
numpy           2.4.0
packaging       25.0
pillow          12.1.0
pip             25.3
pyparsing       3.3.1
python-dateutil 2.9.0.post0
six             1.17.0
```

Pierwszym środowiskiem wykorzystanym w projekcie był system operacyjny **Windows**.
Na systemie zainstalowano interpretera Python oraz wymagane biblioteki z wykorzystaniem menedżera pakietów `pip`.

- Linux



W celu weryfikacji dostępności wymaganych bibliotek wykonano polecenie instalacji pakietów python3-numpy oraz python3-matplotlib z użyciem menedżera pakietów APT. System zwrócił informację, że obie biblioteki są już zainstalowane w najnowszych wersjach i nie wymagają dodatkowej instalacji ani aktualizacji.

Potwierdza to, że środowisko Linux (Kali Linux) posiadało domyślnie zainstalowany interpreter Python 3 wraz z niezbędnymi bibliotekami numerycznymi i graficznymi, co umożliwiło natychmiastowe uruchomienie symulacji ataku SPA bez potrzeby dodatkowej konfiguracji.

4. Opis algorytmu kryptograficznego

4.1. Założenia algorytmu testowego

W ramach projektu nie zastosowano pełnej implementacji konkretnego, standardowego algorytmu kryptograficznego (np. AES), lecz **uproszczony model algorytmu blokowego**, którego celem było umożliwienie przejrzystej analizy podatności na ataki typu Simple Power Analysis (SPA). Takie podejście jest często stosowane w celach dydaktycznych, ponieważ pozwala skupić się na mechanizmie wycieku informacji przez pobór mocy, a nie na złożoności samego algorytmu.

Algorytm operuje na **pojedynczym bajcie danych (8 bitów)** oraz **pojedynczym bajcie klucza**, co pozwala w prosty sposób analizować zależność pomiędzy wykonywanymi operacjami a obserwowanym poborem mocy.

4.2. Schemat działania algorytmu

Algorytm składa się z trzech głównych etapów, które odpowiadają kolejnym fragmentom symulowanego przebiegu poboru mocy:

1. **Załadowanie danych (LOAD)**
   W pierwszym kroku bajt danych wejściowych jest ładowany do rejestru roboczego. Operacja ta nie zależy od klucza kryptograficznego i generuje podobny pobór mocy niezależnie od jego wartości.
2. **Operacja XOR z kluczem**
   W drugim kroku wykonywana jest operacja XOR pomiędzy danymi wejściowymi a kluczem kryptograficznym. Jest to typowa operacja spotykana w algorytmach kryptograficznych, która wprowadza zależność danych od klucza. Pobór mocy w tym etapie może już pośrednio zależeć od wartości klucza.
3. **Operacja zależna od klucza (krok krytyczny)**
   W trzecim etapie wykonywana jest dodatkowa operacja logiczna, której charakter zależy od wybranego bitu klucza (w projekcie – najbardziej znaczącego bitu). W wersji podatnej algorytmu obecność tej zależności powoduje powstanie charakterystycznego wzorca w przebiegu poboru mocy, który może zostać wykorzystany w ataku typu SPA do odróżnienia wartości klucza.

4.3. Model algorytmu a podatność na SPA

Zastosowany algorytm celowo zawiera fragment, w którym:

- liczba lub charakter operacji zależy od wartości klucza,
- wynik pośredni przetwarzania zależy bezpośrednio od klucza.

Takie konstrukcje są **szczególnie podatne na ataki typu Simple Power Analysis**, ponieważ różnice w wykonywanych operacjach prowadzą do różnic w poborze mocy, możliwych do zaobserwowania nawet bez zaawansowanej analizy statystycznej.

Uproszczony charakter algorytmu umożliwia jednoznaczne wskazanie, w którym etapie następuje wyciek informacji oraz pozwala na czytelną demonstrację skuteczności zarówno ataku, jak i zastosowanych mechanizmów obronnych.

4.4 Uzasadnienie wyboru algorytmu

Wybór uproszczonego algorytmu zamiast pełnej implementacji standardu kryptograficznego był celowy i uzasadniony charakterem projektu. Pozwolił on na:

- łatwą identyfikację miejsca występowania wycieku informacji,
- czytelną wizualizację przebiegów poboru mocy,
- jednoznaczną ocenę skuteczności ataku SPA,
- prostą demonstrację wpływu zastosowanych mechanizmów obronnych.

Dzięki temu możliwe było skupienie się na istocie ataków typu side-channel, a nie na szczegółach implementacyjnych złożonych algorytmów kryptograficznych.

5. Symulacja ataku SPA

5.1 Scenariusz ataku i założenia

Celem przeprowadzonego ataku typu Simple Power Analysis (SPA) było odzyskanie informacji o kluczu kryptograficznym na podstawie analizy przebiegów poboru mocy podczas wykonywania algorytmu. Atakujący zakłada dostęp do pomiarów poboru mocy urządzenia realizującego operacje kryptograficzne, jednak nie posiada bezpośredniego dostępu do klucza ani do wewnętrznego stanu algorytmu.

W symulacji przyjęto następujące założenia:

- algorytm przetwarza jeden bajt danych oraz jeden bajt klucza,
- analizowany jest najbardziej znaczący bit klucza (bit 7),
- pobór mocy modelowany jest na podstawie liczby jedynek w reprezentacji binarnej przetwarzanej wartości (model Hamming Weight),
- atakujący porównuje przebiegi poboru mocy dla dwóch różnych wartości klucza, różniących się analizowanym bitem.

Atak koncentruje się na identyfikacji fragmentu przebiegu, w którym wykonywane są operacje zależne od klucza, a następnie na porównaniu charakterystycznych wzorców poboru mocy.

5.2 Implementacja podatnej wersji algorytmu

Podatna wersja algorytmu została zaimplementowana w języku Python w sposób celowo uproszczony, aby wyraźnie uwidocznić zależność pomiędzy kluczem kryptograficznym a poborem mocy. W implementacji nie zastosowano mechanizmów ochronnych, takich jak stałoczasowość czy maskowanie danych.

Algorytm składa się z trzech etapów:

1. załadowanie danych wejściowych do rejestru roboczego,
2. wykonanie operacji XOR z kluczem kryptograficznym,
3. wykonanie operacji zależnej od wartości analizowanego bitu klucza.

W trzecim etapie algorytmu wykonywane są różne operacje logiczne w zależności od wartości bitu klucza, co prowadzi do powstania różnic w liczbie operacji oraz w przetwarzanych danych. Różnice te bezpośrednio wpływają na pobór mocy i stanowią podstawę skutecznego ataku SPA.

5.3 Generowanie przebiegów poboru mocy

Przebiegi poboru mocy generowane są w sposób symulowany. Dla każdej operacji algorytmu obliczana jest wartość modelowego poboru mocy na podstawie **modelu Hamming Weight**, czyli liczby jedynek w binarnej reprezentacji aktualnie przetwarzanej wartości.

Aby zwiększyć realizm symulacji, do każdej próbki poboru mocy dodawany jest losowy szum o rozkładzie normalnym. Pozwala to zasymulować niedoskonałości rzeczywistych pomiarów oraz zakłócenia występujące w fizycznych systemach.

Dla każdego etapu algorytmu generowana jest stała liczba próbek, co umożliwia jednoznaczne wyróżnienie poszczególnych segmentów przebiegu i ułatwia późniejszą analizę.

5.4 Wizualna analiza przebiegów - Windows

W celu oceny skuteczności ataku SPA przeprowadzono wizualną analizę przebiegów poboru mocy wygenerowanych dla dwóch różnych wartości klucza:

- klucz A – analizowany bit równy 0,
- klucz B – analizowany bit równy 1.

Na wykresach przedstawiono przebiegi poboru mocy w funkcji czasu dla obu przypadków. Szczególną uwagę zwrócono na trzeci segment przebiegu, odpowiadający operacjom zależnym od klucza.

Analiza wizualna wykazała wyraźne różnice pomiędzy przebiegami dla obu kluczy. W segmencie krytycznym widoczny jest charakterystyczny wzorzec poboru mocy, który pozwala na jednoznaczne rozróżnienie wartości analizowanego bitu klucza. Obserwacje te potwierdzają, że zaimplementowana wersja algorytmu jest podatna na atak typu Simple Power Analysis.

Dodatkowo, analiza statystyczna wykazała istotną różnicę średnich wartości poboru mocy pomiędzy oboma przebiegami ($\Delta \approx 1.69$), co jednoznacznie potwierdza skuteczność przeprowadzonego ataku.

Wyniki analizy statystycznej dla wersji podatnej algorytmu – widoczna istotna różnica średnich wartości poboru mocy.

```
Wiersz polecenia - python  spa_attack_w.py

C:\Users\student1\spa_project>python spa_attack_w.py

[METRYKI – WERSJA PODATNA]
Segment 3: próbki 160:240
Key A -> mean=3.989, std=0.240
Key B -> mean=2.299, std=0.279
Delta mean = 1.690
```

Przebiegi poboru mocy dla wersji podatnej algorytmu – widoczne różnice umożliwiające atak SPA.

5.5 Wizualna analiza przebiegów - Linux

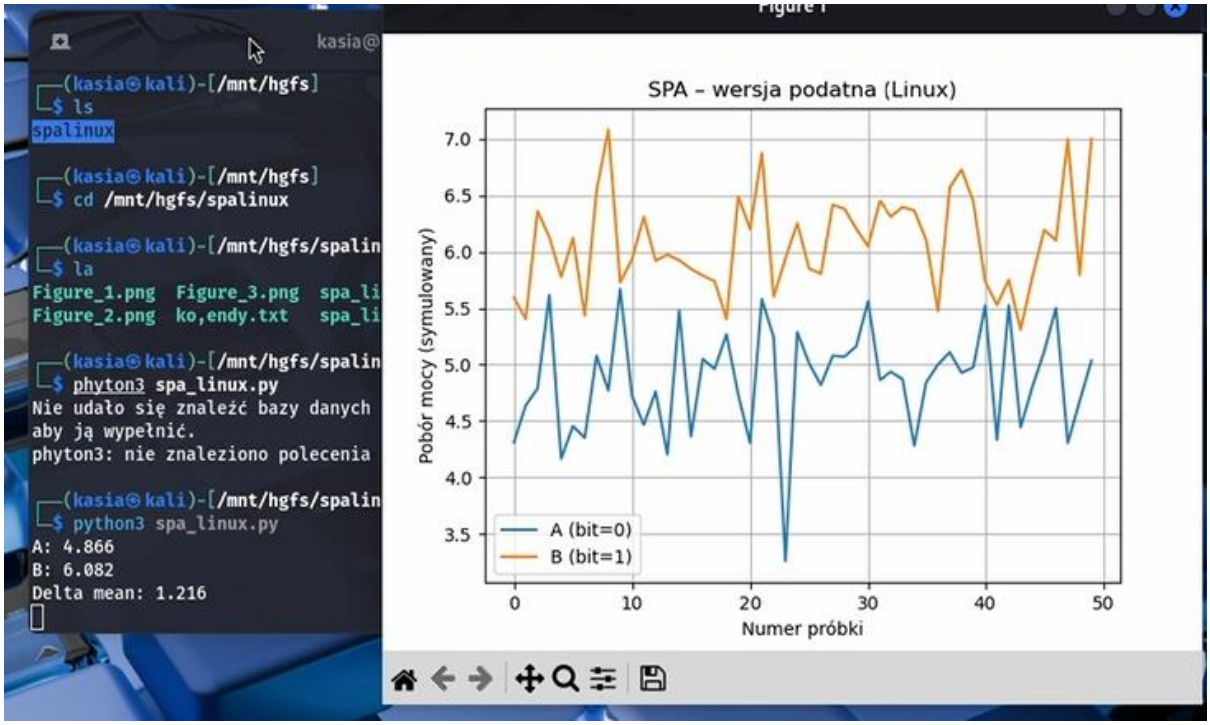
W systemie Linux przeprowadzono analogiczną symulację ataku SPA dla podatnej wersji algorytmu. Na wykresie przedstawiono przebiegi poboru mocy dla dwóch przypadków:
A – gdy analizowany bit klucza wynosi 0,
B – gdy analizowany bit klucza wynosi 1.

Podobnie jak w środowisku Windows, w trzecim segmencie przebiegu widoczne są wyraźne różnice pomiędzy oboma przypadkami. Przebieg odpowiadający bitowi równemu 1 charakteryzuje się większym średnim poborem mocy niż przebieg dla bitu równego 0.

Różnice te wynikają z wykonywania odmiennych operacji logicznych zależnych od wartości klucza.

Analiza statystyczna potwierdziła skuteczność ataku. Średnia wartość poboru mocy dla klucza A (bit = 0) wyniosła 4.866, natomiast dla klucza B (bit = 1) – 6.082. Różnica średnich Δ = 1.216 jest istotnie większa od poziomu szumu i umożliwia jednoznaczne rozróżnienie wartości analizowanego bitu klucza metodą Simple Power Analysis.

Uzyskane wyniki dowodzą, że implementacja algorytmu w wersji podatnej ujawnia informacje o kluczu również w środowisku Linux.



5.6 Porównanie wyników na różnych systemach operacyjnych

Porównanie wyników uzyskanych w systemach Windows i Linux wykazało bardzo zbliżony charakter przebiegów poboru mocy oraz podobny poziom podatności na atak SPA. W obu środowiskach:

- występują wyraźne różnice pomiędzy przebiegami dla bitu klucza równego 0 i 1,
- w segmencie zależnym od klucza widoczny jest charakterystyczny wzorzec umożliwiający jego identyfikację,
- różnica średnich wartości poboru mocy jest istotnie większa od poziomu szumu.

Potwierdza to, że podatność na atak Simple Power Analysis wynika z przyjętej logiki implementacji algorytmu, a nie z właściwości systemu operacyjnego, mimo że implementacje

w poszczególnych środowiskach różnią się szczegółami technicznymi. Zarówno Windows, jak i Linux umożliwiają skuteczne przeprowadzenie ataku przy zastosowaniu tej samej logiki algorytmu i porównywalnych parametrów symulacji.

6. Mechanizmy obronne

6.1 Identyfikacja podatności implementacyjnej

Analiza wyników przeprowadzonego ataku typu Simple Power Analysis wykazała, że podatność algorytmu wynikała z zależności pomiędzy wartością klucza kryptograficznego a wykonywanymi operacjami w trakcie działania algorytmu. W szczególności w trzecim etapie algorytmu wykonywane były operacje, których charakter zależał od wartości analizowanego bitu klucza.

Zależność ta powodowała powstanie wyraźnych różnic w przebiegach poboru mocy, zarówno widocznych wizualnie, jak i potwierdzonych analizą statystyczną. Wersja podatna algorytmu umożliwiała jednoznaczne rozróżnienie wartości bitu klucza na podstawie samej obserwacji poboru mocy, co świadczy o istnieniu istotnego wycieku informacji kanałem bocznym.

Zidentyfikowana podatność miała charakter **implementacyjny**, a nie kryptograficzny – wynikała ze sposobu realizacji algorytmu, a nie ze słabości samego schematu kryptograficznego.

6.2 Zastosowane mechanizmy zabezpieczeń

W celu ograniczenia skuteczności ataku SPA zastosowano zestaw klasycznych mechanizmów obronnych wykorzystywanych w ochronie przed atakami typu side-channel. Głównym celem było usunięcie bezpośredniej zależności pomiędzy kluczem kryptograficznym a obserwowanym poborem mocy w krytycznym etapie algorytmu.

Zastosowane mechanizmy obejmowały:

- **implementację stałoczasową (constant-time)** – zapewnienie, że w krytycznym fragmencie algorytmu wykonywana jest stała liczba operacji, niezależnie od wartości klucza,
- **maskowanie danych (boolean masking)** – wprowadzenie losowej maski do danych przetwarzanych w krytycznym segmencie, co powoduje, że obserwowany pobór mocy zależy od wartości zamaskowanej, a nie bezpośrednio od klucza,
- **dodanie losowego szumu pomiarowego**, symulującego zakłócenia występujące w rzeczywistych systemach sprzętowych.

Kluczowym elementem obrony było pozostawienie danych w postaci zamaskowanej w trakcie analizowanego segmentu algorytmu, co skutecznie utrudniło korelację przebiegów poboru mocy z wartością klucza.

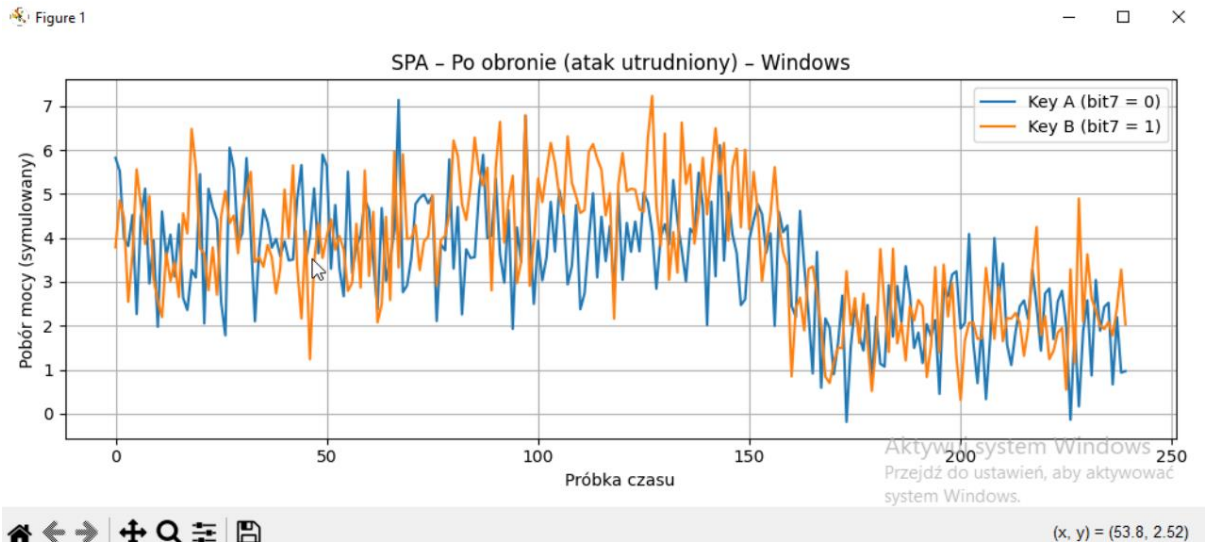
6.3 Implementacja zabezpieczonej wersji algorytmu

Zabezpieczona wersja algorytmu została zaimplementowana w języku Python poprzez modyfikację krytycznego etapu przetwarzania danych. W wersji tej wszystkie operacje wykonywane w trzecim etapie algorytmu realizowane są na **zamaskowanej wartości danych**, a liczba operacji pozostaje stała niezależnie od wartości klucza.

Istotnym elementem implementacji było niewykonywanie odmaskowania w segmencie, dla którego analizowany jest pobór mocy. Dzięki temu obserwowany przebieg poboru mocy zależy od losowej maski, a nie bezpośrednio od klucza kryptograficznego.

Pełna implementacja zabezpieczonej wersji algorytmu została przedstawiona w **Załączniku B**. Skuteczność zastosowanych mechanizmów potwierdzono poprzez ponowną analizę SPA, która wykazała istotne zmniejszenie różnicy średnich wartości poboru mocy ($\Delta \approx 0.11$), co uniemożliwia jednoznaczne rozróżnienie kluczy.

Przebiegi poboru mocy po zastosowaniu obrony – brak wyraźnych różnic zależnych od klucza.



Wyniki analizy statystycznej po zastosowaniu mechanizmów obronnych.

```
C:\Users\student1\spa_project>python spa_defense_w.py

[METRYKI - PO OBRONIE]
Segment 3: próbki 160:240
Key A -> mean=2.031, std=0.963
Key B -> mean=2.140, std=0.878
Delta mean = 0.109
```

7. Ocena skuteczności obrony

7.1 Ponowna symulacja ataku po wprowadzeniu zabezpieczeń – Windows

Po zaimplementowaniu mechanizmów obronnych przeprowadzono ponowną symulację ataku typu Simple Power Analysis w środowisku systemu Windows. Wykorzystano tę samą metodologię analizy co w przypadku wersji podatnej algorytmu, w szczególności tę samą liczbę próbek oraz ten sam model poboru mocy.

Analiza wizualna wygenerowanych przebiegów poboru mocy wykazała brak wyraźnych różnic pomiędzy przebiegami odpowiadającymi różnym wartościom klucza. W krytycznym segmencie algorytmu, który w wersji podatnej umożliwiał jednoznaczne rozróżnienie kluczy, przebiegi po zastosowaniu obrony były do siebie bardzo zbliżone.

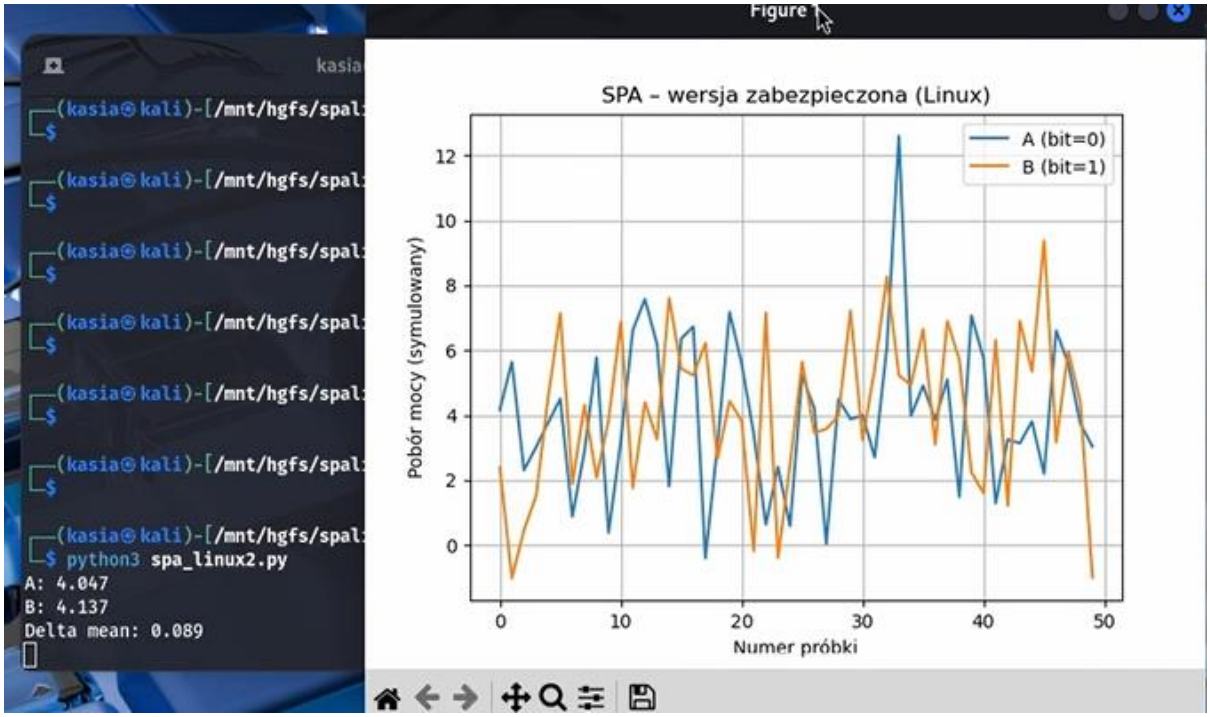
Dodatkowo analiza statystyczna potwierdziła istotne ograniczenie skuteczności ataku. Różnica średnich wartości poboru mocy w analizowanym segmencie spadła do poziomu $\Delta \approx 0.11$, co jest porównywalne z poziomem szumu i uniemożliwia jednoznaczne wnioskowanie o wartości analizowanego bitu klucza metodą SPA.

7.2 Ponowna symulacja ataku po wprowadzeniu zabezpieczeń – Linux

Po wprowadzeniu mechanizmów obronnych przeprowadzono ponowną symulację ataku SPA w środowisku Linux. Na wygenerowanych wykresach przebiegi poboru mocy dla obu wartości analizowanego bitu klucza stały się do siebie bardzo podobne, a charakterystyczne różnice widoczne w wersji podatnej praktycznie zanikły.

Analiza statystyczna potwierdziła skuteczność zabezpieczeń. Średnie wartości poboru mocy dla obu przypadków różniły się nieznacznie, a różnica średnich wyniosła jedynie 0.089. Jest to wartość porównywalna z poziomem szumu losowego, co uniemożliwia jednoznaczne rozróżnienie wartości bitu klucza na podstawie pojedynczego przebiegu mocy.

Oznacza to, że zastosowanie implementacji stałoczasowej oraz maskowania danych skutecznie ograniczyło wyciek informacji kanałem bocznym również w środowisku Linux.



7.3 Porównanie wyników: przed i po zabezpieczeniu

- Windows

Porównanie wyników uzyskanych dla wersji podatnej oraz wersji zabezpieczonej algorytmu jednoznacznie wskazuje na wysoką skuteczność zastosowanych mechanizmów obronnych.

W wersji podatnej algorytmu:

- obserwowano wyraźne różnice w przebiegach poboru mocy,
- możliwe było jednoznaczne rozróżnienie wartości analizowanego bitu klucza,
- różnica średnich wartości poboru mocy wynosiła około $\Delta \approx 1.69$.

Po zastosowaniu mechanizmów obronnych:

- przebiegi poboru mocy stały się trudne do odróżnienia,
- brak było charakterystycznych wzorców zależnych od klucza,
- różnica średnich wartości poboru mocy spadła do poziomu $\Delta \approx 0.11$.

Uzyskane wyniki potwierdzają, że zastosowanie implementacji stałoczasowej oraz maskowania danych skutecznie ogranicza możliwość przeprowadzenia ataku typu Simple Power Analysis. Atak, który w wersji podatnej algorytmu był jednoznacznie skuteczny, po wprowadzeniu zabezpieczeń przestał dostarczać użytecznych informacji o kluczu kryptograficznym.

- Linux

Dla systemu Linux porównanie wersji podatnej i zabezpieczonej algorytmu prowadzi do analogicznych wniosków jak w przypadku Windows.

W wersji podatnej:

obserwowano wyraźne różnice pomiędzy przebiegami dla bitu 0 i 1,
różnica średnich wartości poboru mocy wynosiła 1.216,
atak SPA umożliwiał jednoznaczne odtworzenie wartości analizowanego bitu klucza.

Po zastosowaniu zabezpieczeń:

przebiegi poboru mocy dla obu wartości klucza niemal się pokrywają,
brak jest widocznych wzorców zależnych od klucza,
różnica średnich spadła do 0.089.

Wyniki te potwierdzają wysoką skuteczność zastosowanych mechanizmów obronnych oraz fakt, że ochrona przed atakami SPA ma charakter niezależny od platformy systemowej, o ile zachowana zostanie ta sama logika zabezpieczeń.

8. Podsumowanie i wnioski końcowe

Celem projektu było zapoznanie się z mechanizmem ataków typu Simple Power Analysis oraz praktyczna demonstracja podatności implementacyjnej algorytmu kryptograficznego na tego typu ataki. W ramach pracy zrealizowano pełny cykl laboratoryjny obejmujący identyfikację podatności, przeprowadzenie ataku, wdrożenie mechanizmów obronnych oraz ocenę ich skuteczności.

Przeprowadzona symulacja ataku SPA wykazała, że nawet uproszczona implementacja algorytmu kryptograficznego, w której występują operacje zależne od klucza, może prowadzić do istotnego wycieku informacji poprzez pobór mocy. Analiza wizualna przebiegów oraz metryki statystyczne jednoznacznie potwierdziły skuteczność ataku w wersji podatnej algorytmu.

W kolejnym etapie projektu zaimplementowano mechanizmy obronne oparte na stałoczasowości oraz maskowaniu danych. Zastosowane zabezpieczenia skutecznie ograniczyły zależność poboru mocy od wartości klucza kryptograficznego. Wyniki ponownej analizy SPA wykazały znaczące obniżenie różnicy średnich wartości poboru mocy, co uniemożliwiło jednoznaczne wnioskowanie o kluczu na podstawie obserwowanych przebiegów.

Projekt pozwolił na praktyczne zrozumienie, że bezpieczeństwo kryptograficzne nie zależy wyłącznie od poprawności matematycznej algorytmu, lecz w dużej mierze od jego implementacji. Nawet proste operacje zależne od klucza mogą prowadzić do wycieku informacji, jeśli nie zostaną zastosowane odpowiednie mechanizmy ochronne.

Uzyskane rezultaty potwierdzają, że ataki typu side-channel stanowią realne zagrożenie dla systemów kryptograficznych, a ich skuteczna obrona wymaga świadomego projektowania implementacji oraz uwzględniania aspektów związanych z fizycznym sposobem realizacji obliczeń.

9. Załączniki

A. Pełny kod źródłowy SPA atak Windows

```python
import numpy as np
import matplotlib.pyplot as plt
import os

def hamming_weight(x: int) -> int:
    return bin(x & 0xFF).count("1")

def segment_stats(trace: np.ndarray, seg_start: int, seg_end: int):
    seg = trace[seg_start:seg_end]
    return float(np.mean(seg)), float(np.std(seg))

def save_metrics_csv(path: str, rows: list):
    with open(path, "w", encoding="utf-8") as f:
        f.write("wariant,mean_keyA,std_keyA,mean_keyB,std_keyB,delta_mean\n")
        for r in rows:
            f.write(",".join(str(x) for x in r) + "\n")


def simulate_vulnerable_trace(
    data_byte: int,
    key_byte: int,
    noise_std: float = 0.25,
    n_points_per_step: int = 80
):
    trace = []

    # Krok 1: LOAD
    x = data_byte
    for _ in range(n_points_per_step):
        trace.append(hamming_weight(x) + np.random.normal(0, noise_std))

    # Krok 2: XOR
    x = data_byte ^ key_byte
    for _ in range(n_points_per_step):
        trace.append(hamming_weight(x) + np.random.normal(0, noise_std))

    # Krok 3: WARUNKOWA operacja (wyciek!)
    if (key_byte & 0b10000000) != 0:
        x = x ^ 0xAA
        amp_boost = 1.3
    else:
        amp_boost = 0.0

    for _ in range(n_points_per_step):
        trace.append(hamming_weight(x) + amp_boost + np.random.normal(0, noise_std))

    return np.array(trace)


def plot_and_save(trace_a, trace_b, title, out_png):
    plt.figure(figsize=(10, 4))
    plt.plot(trace_a, label="Key A (bit7 = 0)")
    plt.plot(trace_b, label="Key B (bit7 = 1)")
    plt.title(title)
    plt.xlabel("Próbka czasu")
    plt.ylabel("Pobór mocy (symulowany)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(out_png, dpi=180)
    plt.show()
```

```python
def main():
    os.makedirs("out", exist_ok=True)

    data = 0x3C
    key_a = 0x12  # bit7 = 0
    key_b = 0x92  # bit7 = 1

    trace_a = simulate_vulnerable_trace(data, key_a)
    trace_b = simulate_vulnerable_trace(data, key_b)

    n = len(trace_a)
    seg3_start = 2 * n // 3
    seg3_end = n

    mean_a, std_a = segment_stats(trace_a, seg3_start, seg3_end)
    mean_b, std_b = segment_stats(trace_b, seg3_start, seg3_end)
    delta = abs(mean_b - mean_a)

    print("\n[METRYKI - WERSJA PODATNA]")
    print(f"Segment 3: próbki {seg3_start}:{seg3_end}")
    print(f"Key A -> mean={mean_a:.3f}, std={std_a:.3f}")
    print(f"Key B -> mean={mean_b:.3f}, std={std_b:.3f}")
    print(f"Delta mean = {delta:.3f}")

    save_metrics_csv("out/metrics_vulnerable.csv", [
        ("vulnerable_seg3",
         f"{mean_a:.3f}", f"{std_a:.3f}",
         f"{mean_b:.3f}", f"{std_b:.3f}",
         f"{delta:.3f}")
    ])

    plot_and_save(
        trace_a,
        trace_b,
        title="SPA - Wersja podatna (atak dziala) - Windows",
        out_png="out/attack_vulnerable_windows.png"
    )

    print("Zapisano:")
    print("- out/attack_vulnerable_windows.png")
    print("- out/metrics_vulnerable.csv\n")

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

B. Pełny kod źródłowy SPA obrona Windows

```python
import numpy as np
import matplotlib.pyplot as plt
import os


def hamming_weight(x: int) -> int:
    return bin(x & 0xFF).count("1")

def segment_stats(trace: np.ndarray, seg_start: int, seg_end: int):
    seg = trace[seg_start:seg_end]
    return float(np.mean(seg)), float(np.std(seg))

def save_metrics_csv(path: str, rows: list):
    with open(path, "w", encoding="utf-8") as f:
        f.write("wariant,mean_keyA,std_keyA,mean_keyB,std_keyB,delta_mean\n")
        for r in rows:
            f.write(",".join(str(x) for x in r) + "\n")

def simulate_protected_trace(
    data_byte: int,
    key_byte: int,
    noise_std: float = 1.0,
    n_points_per_step: int = 80
):
    trace = []

    # Krok 1: LOAD
    x = data_byte
    for _ in range(n_points_per_step):
        trace.append(hamming_weight(x) + np.random.normal(0, noise_std))

    # Krok 2: XOR
    x = data_byte ^ key_byte
    for _ in range(n_points_per_step):
        trace.append(hamming_weight(x) + np.random.normal(0, noise_std))

    # Krok 3: MASKOWANIE + stałoczasowo (tu bronimy przed SPA)
    m = np.random.randint(0, 256)
    x_masked = x ^ m

    x_masked = x_masked ^ 0xAA
    x_masked = ((x_masked << 1) & 0xFF) | ((x_masked >> 7) & 0x01)  # rotacja bitowa
    x_masked = x_masked ^ 0x55

    for _ in range(n_points_per_step):
        trace.append(hamming_weight(x_masked) + np.random.normal(0, noise_std))

    return np.array(trace)


def plot_and_save(trace_a, trace_b, title, out_png):
    plt.figure(figsize=(10, 4))
    plt.plot(trace_a, label="Key A (bit7 = 0)")
    plt.plot(trace_b, label="Key B (bit7 = 1)")
    plt.title(title)
    plt.xlabel("Próbka czasu")
    plt.ylabel("Pobór mocy (symulowany)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(out_png, dpi=180)
    plt.show()
```

```
def main():
    os.makedirs("out", exist_ok=True)

    data = 0x3C
    key_a = 0x12  # bit7 = 0
    key_b = 0x92  # bit7 = 1

    trace_a = simulate_protected_trace(data, key_a)
    trace_b = simulate_protected_trace(data, key_b)

    n = len(trace_a)
    seg3_start = 2 * n // 3
    seg3_end = n

    mean_a, std_a = segment_stats(trace_a, seg3_start, seg3_end)
    mean_b, std_b = segment_stats(trace_b, seg3_start, seg3_end)
    delta = abs(mean_b - mean_a)

    print("\n[METRYKI - PO OBRONIE]")
    print(f"Segment 3: próbki {seg3_start}:{seg3_end}")
    print(f"Key A -> mean={mean_a:.3f}, std={std_a:.3f}")
    print(f"Key B -> mean={mean_b:.3f}, std={std_b:.3f}")
    print(f"Delta mean = {delta:.3f}")

    save_metrics_csv("out/metrics_protected.csv", [
        ("protected_seg3",
         f"{mean_a:.3f}", f"{std_a:.3f}",
         f"{mean_b:.3f}", f"{std_b:.3f}",
         f"{delta:.3f}")
    ])

    plot_and_save(
        trace_a,
        trace_b,
        title="SPA - Po obronie (atak utrudniony) - Windows",
        out_png="out/defense_protected_windows.png"
    )

    print("Zapisano:")
    print("- out/defense_protected_windows.png")
    print("- out/metrics_protected.csv\n")

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

C. Pełny kod źródłowy SPA atak Linux

```python
import numpy as np
import matplotlib.pyplot as plt
import random

DATA = 0b10101100
KEY_A = 0b00000001  # A = bit 0
KEY_B = 0b00000011  # B = bit 1
NOISE_LEVEL = 0.5
SAMPLES = 50

def hamming_weight(x: int) -> int:
    return bin(x).count("1")

def vulnerable_trace(data: int, key: int, samples: int) -> list[float]:
    trace = []
    for _ in range(samples):
        result = data ^ key

        # podatność: różna ścieżka zależna od bitu klucza
        if key & 1:
            temp = result ^ 0b11110000
        else:
            temp = result ^ 0b00001111

        power = hamming_weight(temp) + random.gauss(0, NOISE_LEVEL)
        trace.append(power)
    return trace

trace_A = vulnerable_trace(DATA, KEY_A, SAMPLES)
trace_B = vulnerable_trace(DATA, KEY_B, SAMPLES)

A = float(np.mean(trace_A))
B = float(np.mean(trace_B))
delta_mean = abs(B - A)

print(f"A: {A:.3f}")
print(f"B: {B:.3f}")
print(f"Delta mean: {delta_mean:.3f}")

plt.plot(trace_A, label="A (bit=0)")
plt.plot(trace_B, label="B (bit=1)")
plt.title("SPA – wersja podatna (Linux)")
plt.xlabel("Numer próbki")
plt.ylabel("Pobór mocy (symulowany)")
plt.legend()
plt.grid(True)
plt.show()
```

D. Pełny kod źródłowy SPA obrona Linux

```python
import numpy as np
import matplotlib.pyplot as plt
import random

DATA = 0b10101100
KEY_A = 0b00000001
KEY_B = 0b00000011
NOISE_LEVEL = 2.0   # większy szum (obrona)
SAMPLES = 50

def hamming_weight(x: int) -> int:
    return bin(x).count("1")

def protected_trace(data: int, key: int, samples: int) -> list[float]:
    trace = []
    for _ in range(samples):
        # maskowanie danych
        mask = random.randint(0, 255)
        masked_data = data ^ mask

        # wyrównanie ścieżek wykonania: zawsze te same operacje
        temp1 = masked_data ^ 0b11110000
        temp2 = masked_data ^ 0b00001111
        result = temp1 ^ temp2 ^ key

        power = hamming_weight(result ^ mask) + random.gauss(0, NOISE_LEVEL)
        trace.append(power)
    return trace

trace_A = protected_trace(DATA, KEY_A, SAMPLES)
trace_B = protected_trace(DATA, KEY_B, SAMPLES)

A = float(np.mean(trace_A))
B = float(np.mean(trace_B))
delta_mean = abs(B - A)
```

```python

print(f"A: {A:.3f}")
print(f"B: {B:.3f}")
print(f"Delta mean: {delta_mean:.3f}")

plt.plot(trace_A, label="A (bit=0)")
plt.plot(trace_B, label="B (bit=1)")
plt.title("SPA – wersja zabezpieczona (Linux)")
plt.xlabel("Numer próbki")
plt.ylabel("Pobór mocy (symulowany)")
plt.legend()
plt.grid(True)
plt.show()

```

E. Nagranie wideo (dołączone w kolejnym pliku)