Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Cyberbezpieczeństwa

Raport z postępu pracowni dyplomowej

na kierunku Telekomunikacja
w specjalności Techniki Teleinformatyczne

Akceleracja sprzętowa kryptoanalizy algorytmów kryptograficznych

Andrzej Tłomak
Numer albumu 311450

promotor
dr. hab. inż. Mariusz Rawski

WARSZAWA 2024

Spis treści

1. Plan na semestr 2023Z

Celem tego etapu było zapoznanie się z literaturą opisującą aktualny State of the Art kryptoanalizy systemów opartych o krzywe eliptyczne w ciałach skończonych, zapoznanie się z teorią oraz podstawami matematycznymi zagadnienie krzywych eliptycznych w kryptografii oraz przygotowanie środowiska do pracy z wykorzystaniem technologii CUDA.

Status zaplanowanych zadań:

- Przegląd literatury - **zrealizowane**
- Zapoznanie się z podstawami matematycznymi- **zrealizowane**
- Konfiguracja środowiska pracy (CUDA oraz SageMath) - **zrealizowane**
- Implementacja i testowanie prototypu w technologii CUDA - **w trakcie**

2. Wstęp teoretyczny

W tym etapie jednym z zaplanowanych celów, było zapoznanie się z teorią stojącą za kryptografią krzywych eliptycznych. Aby jednak zrozumieć to zagadnienie, musiałem również zgłębić szersze pole tej dziedziny kryptografii, jaką jest kryptografia oparta o **problem logarytmu dyskretnego**, zarówno na krzywych eliptycznych jak i innych, odpowiednich grupach.

Każde wymienione poniżej zagadnienie, zaimplementowałem również w pakiecie obliczeniowym SageMath. Odpowiadające zaganieniom listingi kodu znajdują się w dalszej części raportu.

2.1. Problem logarytmu dyskretnego

Problem logarytmu dyskretnego (**DLP**) jest podstawą wielu kryptosystemów. Najbardziej znanym z nich jest kryptosystem ElGamala. Problem logarytmu dyskretnego można przedstawić zarówno na grupie multiplikatywnej $(\mathbb{G}, \cdot)$ oraz grupie addytywnej, przy odpowiednim zdefiniowaniu operacji dodawania na krzywej eliptycznej $(\mathbb{E}, +)$ [1].

2.2. DLP w grupie multiplikatywnej

Jeżeli $\mathbb{G}$ to (skończona) grupa multiplikatywna, $\alpha \in \mathbb{G}$ to element rzędu n oraz $\beta \in <\alpha>$ (jest w podgrupie generowanej przez α), to uważane za problematyczne jest znalezienie takiej liczby a:

$$a \in \mathbb{Z} \text{ oraz } 0 \leq a \leq n-1$$

że:

$$\alpha^a = \beta$$

Liczbę a można przedstawić jako:

$$\log_\alpha \beta$$

2.3. DLP w grupie addytywnej

W przypadku krypografii opartej o krzywe eliptyczne, DLP dotyczy grupy addytywnej $(\mathbb{E}, +)$ zdefiniowanej na krzywej eliptycznej. Niech α jest rzędu n. W takim przypadku, ponieważ operacją na grupie jest dodawanie modulo n, to działanie potęgowania przedstawia się jako:

$$\alpha \cdot a = \beta \pmod{n}$$

Przy odpowiednim wyborze grupy addytywnej, rozwiązanie problemu logarytmu dyskretnego, tj. znalezienie a, jest trudne [2][1].

2.4. Krzywe eliptyczne

Krzywą eliptyczną nieosobliwą nad ciałem $\mathbb{K}$ o charakterystyce różnej od 2 i 3 definiuje się za jako zbiór rozwiązań $(x, y) \in \mathbb{R} \times \mathbb{R}$ równiania: [1]

$$y^2 = x^3 + ax + b$$

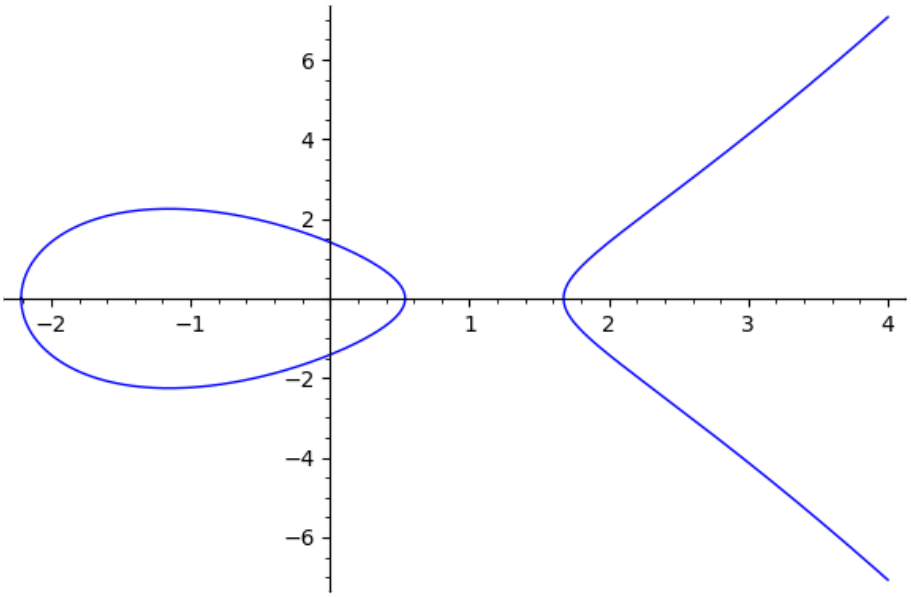
przy założeniu, że stałe a, b takie, że:

$$4a^3 + 27b^2 \neq 0$$

Jest to tak zwana forma *Weierstrassa* krzywej eliptycznej.

2.4.1. Krzywe eliptyczne na liczbach rzeczywistych

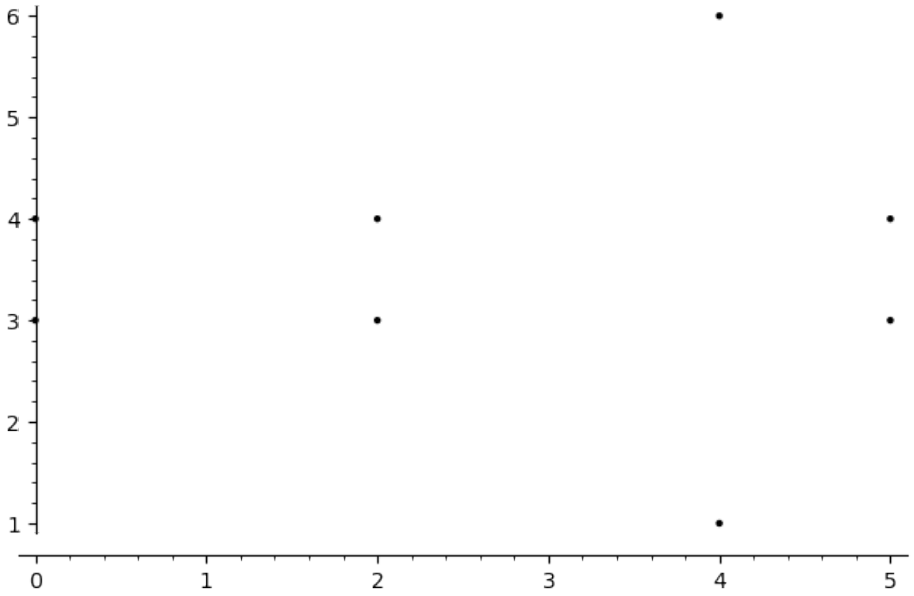
Krzywe eliptyczne zdefiniowane na liczbach rzeczywistych nie są kluczowe w systemach kryptograficznych[2][1], ale takie ustawienia pozwalają na prostsze przedstawienie niektórych zagadnień np. dodawnie punktów na krzywej.



Rys. 2.1. Krzywa eliptyczna $y^2 = x^3 - 4 + 2$

Krzywe eliptyczne na ciele skończonym

Krzywa eliptyczna na ciele skończonym jest stosowana w kryptografii. Z powodu charakterystyki ciała, jej wykres nie przypomina krzywej na liczbach rzeczywistych. Krzywa taka składa się z punktów, których współrzędne należą do ciała na którym jest opisana. Wszystkie operacje na krzywej, takie jak dodawanie, wykonuje się również z zastosowaniem operacji modulo rzędu ciała.

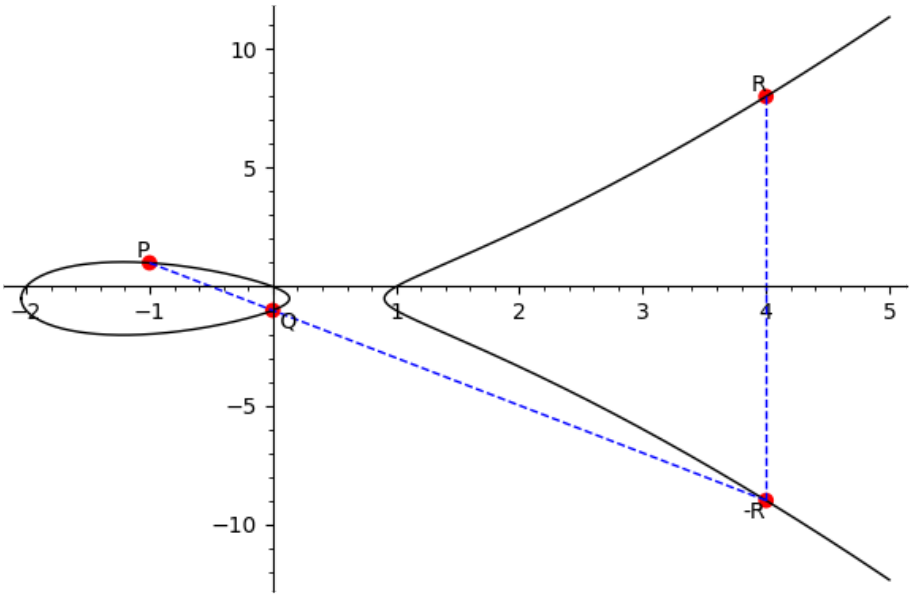


Rys. 2.2. Krzywa eliptyczna $y^2 = x^3 - 4 + 2$ nad $GF(7)$

2.4.2. Dodawanie punktów na krzywej eliptycznej

Przedstwienie krzywej eliptycznej na ciele liczb rzeczywistych, umożliwia proste zwizualizowanie geometrycznej interpretacji dodawania punktów leżących na krzywej.

Geometryczne dodawanie punktów na krzywej eliptycznej polega na połączeniu dwóch punktów P i Q prostopadłą linią, która przecina krzywą w trzecim punkcie, R'. Następnie, wynikowy punkt R, będący sumą $P + QP + Q$, znajdujemy przez odbicie punktu R' względem osi x. W przypadku dublowania punktu, czyli dodawania punktu PP do siebie samego, rysujemy styczną do krzywej w punkcie P, która przecina krzywą w nowym punkcie. Odbicie tego punktu względem osi x daje nam wynik $2P$.
Kod w SageMath użyty do wizualizacji dodawania: listing 4.1



Rys. 2.3. P + Q na krzywej eliptycznej $y^2 + y = x^3 - x^2 + 2x$

2.4.3. Dodawanie punktów na krzywej zdefiniowanej na ciele skończonym

Dodawanie punktów krzywej eliptycznej na ciele skończonym nie ma przejrzyjstej reprezentacji geometrycznej. W tym celu stosuje się podejście analityczne. Wtedy, dodawanie wygląda w następujący sposób:

1. Przypadek, gdy $P \neq Q$:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1},$$
$$x_3 = \lambda^2 - x_1 - x_2,$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$

2. Przypadek, gdy $P = Q$:

$$\lambda = \frac{3x_1^2 + a}{2y_1},$$
$$x_3 = \lambda^2 - 2x_1,$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$

Dodawanie punktów w SageMath listing 4.1

3. State of Art

3.1. GPU

Procesory graficzne są dedykowane do wykonywania wielu równoległych obliczeń. Dzięki temu, są bardzo wydajne w zadaniach które można łatwo zrónoleglić. Wiele algorytmów do kryptoanalizy pozwala na przetwarzanie równoległe, w szczególności algorytm **rho-Pollarda**.

Solving Discrete Logarithms in Smooth-Order Groups with CUDA

W roku 2012 na karcie graficznej NVIDIA Tesla M2050 osiągnięto wydajność na poziomie 51.9 miliona operacji mnożenia modularnego 768-bit na sekundę. Implementacja opierała się głównie na języku C z CUDA framework wraz z jednostkowymi segmentami w języku PTX który jest zbiorem instrukcji dla CUDA GPU. Praca ma dla mnie szczególną na tym etapie, ponieważ razem z pracą udostępniono kod implementacji na prawach open-source, dodatkowo opisuje ograniczenia i założenia jakie należy uwzględnić przy implementacji algorytmu rho-Pollarda na GPU[3].

ECC2K-130 on NVIDIA GPUs

Artykuł opisuje implementację algorytmu rho-Pollarda na karcie graficznej NVIDIA GTX 295. Autorzy wybrali krzywą Koblitza ECC2K-130. Opisano decyzje związane z wyborem bazy (w tym przypadku wybrano bazę normalną). Przedstawiono również szczegóły związane z zarządzaniem pamięcią oraz problem związany z DRAM'em karty (przy pełnej utylizacji GPU w pamięci brakowało miejsca na input) Wynik: Średnio obliczenie ECDLP na tej krzywej zajełoby 2 lata przy 534 kartach.

3.2. FPGA

Solving Discrete Logarithms in Smooth-Order Groups with CUDA

W 2014 opublikowano pracę przedstawiającą implementację FPGA na platformie Virtex-6. dedykowaną do rozwiązania logarytmu dyskretnego na 113-bitowej krzywej Koblitza. Opisano zastosowane zabiegi poprawiające optymalizację, oraz design poszczególnych moduł Na przykład w celu lepszej optymalizacji, wykorzystano bazę normalną F_{2^m} w jednym z modułów do liczenia automorfizmu punktów. Wynik po ekstrapolacji to 28 dni na rozwiązanie logarytmu na krzywej Koblitza 113 bit.

3.3. CPU

CPU nie są najwydajniejszą architekturą do wykonywania równoległych obliczeń. Zazwyczaj charakteryzują się znacznie wydajniejszymi jednostkami obliczeniowymi (rdze-

niami) niż na przykład GPGPU, ale jest ich również znacznie mnniej niż w GPGPU. CPUs są najlepiej przystosowane do przetwarzania potokowego.

A Review on solving ECDLP over Large Finite Field using Parallel Pollard's Rho (p) Method

Praca przedstawia wyniki czasowe przy obliczaniu ECDLP na ciele skończonym rzędu p do 85-bitów. Zastosowano do tego cluster CPU o 256 rdzeniach octa-core. Artykuł również jest interesujący ponieważ zwięźle opsuje background matematyczny oraz przejrzyście przedstawia wersję równoległą algorytmu rho Pollarda[4]. Wynik to 52 godziny dla krzywej na ciele rozmiaru p = 85-bitów.

4. Code listings

Dodawnie punktów na krzywej eliptycznej nad $\mathbb{R}$

Wydruk 4.1. Dodawnie punktów na krzywej eliptycznej na liczbach rzeczywistych

```
# SageMath kernel

sage: R = P + Q
sage: R_inv = -R

# Plot elliptic curve
sage: plot_curve = plot(E, rgbcolor="black", xmax=5)

# Plot points
sage: plot_p = plot(P, marker="o", rgbcolor="red", size=50)
sage: p_label = text(
    "P",
    P.dehomogenize(2),
    horizontal_alignment="right",
    vertical_alignment="bottom",
    color="black",
)
sage: plot_q = plot(Q, marker="o", rgbcolor="red", size=50)
sage: q_label = text(
    "␣Q",
    Q.dehomogenize(2),
    horizontal_alignment="left",
    vertical_alignment="top",
    color="black",
)
sage: plot_r = plot(R, marker="o", rgbcolor="red", size=50)
sage: r_label = text(
    "R",
    R.dehomogenize(2),
    horizontal_alignment="right",
    vertical_alignment="bottom",
    color="black",
)
sage: plot_r_inv = plot(R_inv, marker="o", rgbcolor="red", size=50)
sage: r_inv_label = text(
```

```
    "–R",
    R_inv.dehomogenize(2),
    horizontal_alignment="right",
    vertical_alignment="top",
    color="black",
)

sage: p6 = line2d(
    [P.dehomogenize(2), R_inv.dehomogenize(2)], linestyle="--", rgbcolor="blue"
)
sage: p7 = line2d(
    [R.dehomogenize(2), R_inv.dehomogenize(2)], linestyle="--", rgbcolor="blue"
)

sage: plot_curve + plot_p + plot_q + plot_r + plot_r_inv + p6 + p7 + p_label + q_la
```

Dodawnie punktów na krzywej eliptycznej nad $\mathbb{GF}(7)$

Wydruk 4.2. Dodawnie punktów na krzywej eliptycznej zdefiniowanej na GF(7)

```
Gf = GF(7)

E = EllipticCurve(Gf,[3,4])

def ell_add(E, P1, P2):
    a, b, p = E
    if P1 == "inf": return P2
    if P2 == "inf": return P1
    x1, y1 = P1; x2,y2 = P2
    x1 %= p; y1 %= p; x2 %= p; y2 %= p;

    if x1 == x2 and y1 == p-y2: return "inf"

    if P1 == P2:
        if y1 == 0: return "inf"
        lam = (3*x1^2+a) * inverse_mod(2*y1,p)
    else:
        lam = (y1-y2) * inverse_mod(x1-x2,p)
    x3 = lam^2 - x1 - x2
    y3 = -lam*x3 - y1 + lam*x1
```

```
    return (x3%p, y3%p)

P1 = E.random_point()

p1 = (2,5)
print(p1)
for _ in range(9):
    p1 = ell_add(e, p1, p1)
    p1 = (Integer(p1[0]),Integer(p1[1]))
    print(p1)
```

Kryptosystem El-Gamala na $(\mathbb{G},\cdot)$

Wydruk 4.3. Kryptosystem El-Gamala

```
G = Integers(11)

# Bob:

# Private
x = G(3)

## Public
g = G(5)
h = g^x

# Alice:

# Message
m = G(5)

# Private
k = G(8)

# Public
a = g^k
b = (h^k)*m

# Decrypt Bob:
b*(a^(-1))^x
```

Kryptosystem El-Gamala na krzywej $y^2 = x^3 + x + 6 \in \mathbb{GF}(11)$

Wydruk 4.4. Kryptosystem El-Gamala na krzywej

```
Gf = GF(11)
E = EllipticCurve(Gf, [1, 6])

# Alice -> Bob

# Bob private
m = 7

# Bob public
P = E(2, 7)
Q = m * P


def alice_encrypt():
    k = 6  # random secret variable
    x = 9  # message

    kP = k * P
    kQ = k * Q

    secret_hash = safe_hash_function(kQ)

    y1 = kP
    y2 = (x + secret_hash) % 11
    return (y1, y2)


# just mock, should be secure one way hash fucntion
def safe_hash_function(q):
    if q == E(8, 3):
        return 4
    return 3


# Alice sends cypher
cypher = alice_encrypt()
```

```
def bob_decrypt(cypher):
    y1 = cypher[0]
    y2 = cypher[1]

    kQ = m * y1   # m * (kP) / but mP = Q so its kQ

    secret_hash = safe_hash_function(kQ)

    plain_message = (y2 - secret_hash) % 11

    return plain_message


bob_decrypt(cypher)
```

Algorytm rho-Pollarda dla grupy multiplikatywnej

Wydruk 4.5. Algorytm rho-Pollarda dla grupy multiplikatywnej

```
class Group_parameters:
    def __init__(self, p, n, alpha, beta) -> None:
        self.p = p
        self.n = n
        self.alpha = alpha
        self.beta = beta

class Triple:
    def __init__(self, x, a, b) -> None:
        self.x = x
        self.a = a
        self.b = b
    def __str__(self) -> str:
        return f"x = {self.x}, a = {self.a}, b = {self.b}"


def f(tripe: Triple, group: Group_parameters) -> Triple:
    if tripe.x % 3 == 1:
        x = group.beta * tripe.x % group.p
```

```
        a = tripe.a
        b = tripe.b + 1
        return Triple(x, a, b)
    if tripe.x % 3 == 0:
        x = tripe.x ** 2 % group.p
        a = tripe.a * 2
        b = tripe.b * 2
        return Triple(x, a, b)
    else:
        x = group.alpha * tripe.x % group.p
        a = tripe.a + 1
        b = tripe.b
        return Triple(x, a, b)

g = Group_parameters(809, 101, 89, 618)

t1 = f(Triple(1, 0, 0), g)
t2 = f(t1, g)



i = 1
print('%s %s | %s %s' % (i, t1, 2*i, t2))

i = 2
while(t1.x != t2.x):
    t1 = f(t1, g)
    t2 = f(f(t2, g), g)
    print('%s %s | %s %s' % (i, t1, 2*i, t2))
    i=i+1

print(f"Znaleziona kolizja: x: {t1.x}")
```

Algorytm rho-Pollarda znajdowania punktów na krzywej eliptycznej

Wydruk 4.6. Algorytm rho-Pollarda znajdowania punktów na krzywej eliptycznej

```
class Group_parameters:
    def __init__(self, p, alpha, beta) -> None:
        self.p = p
        self.alpha = alpha
```

```
        self.beta = beta


class Triple:
    def __init__(self, x, a, b) -> None:
        self.x = x  # Point at Elliptic Curve
        self.a = a  # just a number
        self.b = b  # just a number

    def __str__(self) -> str:
        return f"x = {self.x}, a = {self.a}, b = {self.b}"


def f(tripe: Triple, group: Group_parameters) -> Triple:
    #print("Function run")
    #print(f"Input: {tripe}")

    x_of_xpoint = tripe.x[0]
    y_of_xpoint = tripe.x[1]

    p = group.p

    if int(y_of_xpoint) % 3 == 1:
        x = group.beta + tripe.x
        a = tripe.a
        b = tripe.b + 1
        # check
        if (x != a * g.alpha + b * g.beta):
            pass
            print(f"1 xab: {x} {a} {b}")
        return Triple(x, a, b)
    if int(y_of_xpoint) % 3 == 0:
        x = 2 * tripe.x
        a = (tripe.a * 2)
        b = (tripe.b * 2)
        if (x != a * g.alpha + b * g.beta):
            pass
            print(f"2 xab: {x} {a} {b}")
        return Triple(x, a, b)
```

```
    else:
        x = group.alpha + tripe.x
        a = tripe.a + 1
        b = tripe.b
        if (x != a * g.alpha + b * g.beta):
            pass
            print(f"3 xab: {x} {a} {b}")
        return Triple(x, a, b)


def main(g: Group_parameters, t1):
    i = 1

    t2 = f(t1, g)

    print("%s %s | %s %s" % (i, t1, 2 * i, t2))

    i = 2
    while t1.x != t2.x:
        t1 = f(t1, g)
        t2 = f(f(t2, g), g)
        print("%s %s | %s %s" % (i, t1, 2 * i, t2))
        i = i + 1

    print(f"Found:\nt1: {t1}\nt2: {t2}")

    x = -((t2.a - t1.a) / (t1.b - t2.b))

    print(x)
```

Bibliografia

[1] M. B. P. Dauglas R. Stinson, „Kryptografia w teorii i praktyce, Wydanie IV”, w PWN, 2021, s. 274.

[2] A. Chrzęszczyk, „Algorytmy teorii liczb i kryptografii”, w btc, 2010, s. 279.

[3] R. Henry i I. Goldberg, „Solving Discrete Logarithms in Smooth-Order Groups with CUDA 1”, adr.: http://cacr.uwaterloo.ca/.

[4] K. A. Chavan, I. Gupta i D. B. Kulkarni, „A Review on Solving ECDLP over Large Finite Field Using Parallel Pollard's Rho (p) Method”, t. 18, s. 1–11, 2. DOI: 10.9790/0661-1802040111. adr.: www.iosrjournals.org.

Wykaz symboli i skrótów

EiTI – Wydział Elektroniki i Technik Informacyjnych
PW – Politechnika Warszawska
FPGA – Field Programmable Gates Array
DLP – Discrete Logarithm Problem
GF – Galois Field (ciało skończone)

Spis rysunków

Spis tabel

Spis wydruków

Spis załączników