Lista de Exercícios
ECDLP e o Ataque Baby-Step Giant-Step

Curso de Introdução à Criptografia

Objetivo

Implementar o algoritmo **Baby-Step Giant-Step (BSGS)** para resolver o Problema do Logaritmo Discreto em Curvas Elípticas (ECDLP) e usá-lo para "quebrar"uma simulação da troca de chaves Diffie-Hellman (ECDH).

Contexto e Ferramentas

Nota: A Curva de Edwards

Nossos exercícios são baseados em uma forma específica de Curva de Edwards. Todos os parâmetros (a, d, p) referem-se à equação da curva definida sobre um corpo finito $\mathbb{F}_p$:

Forma da Curva de Edwards:

$$ax^2 + y^2 \equiv 1 + dx^2y^2 \pmod{p}$$

Onde:

- a e d são os coeficientes que definem a forma da curva.
- p é o módulo primo que define o corpo finito $\mathbb{F}_p$ onde a curva existe.

Funções do Pacote `edwards_crypto`

Para estes exercícios, você deve usar o pacote `edwards_crypto` que desenvolvemos. Você precisará importar as seguintes funções do módulo `curves.py`:

- `edwards_add(P1, P2, a, d, p)`: Adiciona dois pontos.
- `Double_and_Add(P, k, a, d, p)`: Multiplica um ponto por um escalar k.
- `scalar_multiplication(P, k, a, d, p)`: (Alternativa ao `Double_and_Add`).

Lembre-se que o ponto de identidade (neutro) $\mathcal{O}$ para esta curva é `(0, 1)`, e que os pontos são tuplas `(x, y)`, que podem ser usadas diretamente como chaves de dicionários.

Exercício 1: Função Auxiliar de Inverso

O algoritmo BSGS requer a subtração de pontos ($Q - P_m$), que é implementada como uma adição com o inverso ($Q + (-P_m)$).

Sua primeira tarefa é implementar uma função auxiliar que calcule o inverso de um ponto em uma Curva de Edwards.

- **Crie a função:** `ponto_inverso(P, p)`
- **Dica:** Para a Curva de Edwards na forma que usamos, o inverso do ponto $P = (x, y)$ é o ponto $-P = (-x, y)$. Não se esqueça de aplicar o módulo `p` ao novo valor de x.

Exercício 2: Implementação do Baby-Step Giant-Step

Implemente a função principal do algoritmo BSGS.

- **Crie a função:** `meu_baby_step_giant_step(P, Q, n, a, d, p)`
- **Argumentos:**
  - `P`: O ponto base (gerador).
  - `Q`: O ponto "alvo"(a chave pública que queremos quebrar).
  - `n`: A ordem do subgrupo gerado por `P`.
  - `a, d, p`: Os parâmetros da curva (definidos na seção anterior).
- **Retorno:** O logaritmo discreto `k` (um inteiro), ou `None` se não for encontrado.

Etapas Lógicas do Algoritmo

Sua função deve seguir estas etapas:

1. **Cálculo de m:** Calcule $m = \lceil \sqrt{n} \rceil$. Use `math.ceil(math.sqrt(n))`.
2. **Fase "Baby Steps":** Crie um dicionário (hash map). Itere j de 0 até $m - 1$ e armazene os pontos jP.
   - O dicionário deve mapear: `Ponto` → `j`.
   - Comece com $j = 0$: O ponto é `(0, 1)` (identidade).
   - A cada iteração, adicione `P` ao ponto anterior usando `edwards_add()`.
3. **Cálculo do "Passo Gigante":**
   - Calcule o ponto $P_m = mP$ usando `Double_and_Add()`.
   - Calcule o inverso deste ponto, $-P_m$, usando sua função `ponto_inverso()` do Exercício 1.
4. **Fase "Giant Steps":** Itere i de 0 até $m - 1$. O objetivo é encontrar uma colisão.
   - Calcule o ponto $Q_i = Q - i(P_m)$. (Dica: para fazer isso, comece com $Q_i = Q$ e, a cada iteração, adicione o inverso $-P_m$ a Q_i).

- Em cada iteração, verifique se o Q_i atual **está no dicionário** dos "baby steps".
- **Colisão Encontrada:** Se Q_i estiver na tabela, significa que $Q_i = jP$ para algum j.

5. **Cálculo Final:** Ao encontrar a colisão (onde $Q - i(mP) = jP$), o logaritmo discreto `k` é $k = (i \cdot m + j) \pmod{n}$. Retorne `k`.

Exercício 3: "Quebrando"o Diffie-Hellman

Agora, use sua implementação para descobrir as chaves privadas secretas do exemplo `main_diffie_hellman.py`.

- **Crie um script:** `ataque_bsgs.py`.
- **Parâmetros do Teste:** Use os parâmetros do exemplo Diffie-Hellman:
  - `p = 17`
  - `a = 1`
  - `d = 2`
  - `P = (3, 0)` (Ponto base)
  - `n = 16` (Ordem do subgrupo, necessária para o BSGS)
- **Chaves Secretas (para Verificação):** Os valores que você deve encontrar são:
  - `alice_pkey = 3`
  - `bob_pkey = 5`

Tarefas do Script `ataque_bsgs.py`

O seu script deve fazer o seguinte:

1. Importar as funções necessárias de `edwards_crypto` e suas funções dos Exercícios 1 e 2.
2. Definir todos os parâmetros (p, a, d, P, n, alice_pkey, bob_pkey).
3. **Simular o ECDH:**
   - Calcule a Chave Pública de Alice: $A =$ `alice_pkey`$\cdot P$ (use `scalar_multiplication` ou `Double_and_Add`).
   - Calcule a Chave Pública de Bob: $B =$ `bob_pkey` $\cdot P$.
   - Imprima os valores de A e B (ex: `Chave Publica de Alice (A): (X, Y)`).
4. **Executar o Ataque:**
   - Use sua função `meu_baby_step_giant_step()` para encontrar a chave de Alice. (ex: `k_alice = meu_baby_step_giant_step(P, A, n, ...)`)
   - Use sua função `meu_baby_step_giant_step()` para encontrar a chave de Bob. (ex: `k_bob = meu_baby_step_giant_step(P, B, n, ...)`)

5. **Verificar o Sucesso:**
   - Imprima os valores encontrados: `Chave de Alice encontrada: ...`
   - Verifique se `k_alice` é igual a `alice_pkey` e se `k_bob` é igual a `bob_pkey`.

O que Entregar

1. O arquivo `meu_bsgs.py` contendo as implementações de `ponto_inverso()` e `meu_baby_step_gian`
2. O arquivo `ataque_bsgs.py` (o script de validação do Exercício 3).
3. A saída (print do console) da execução do `ataque_bsgs.py`, mostrando que o script calculou as chaves públicas e, em seguida, encontrou com sucesso as chaves privadas `3` e `5`.