600.445/600.646 Practical Cryptographic Systems February 28, 2025

Assignment 3

Instructor: Matthew Green Due: 11:59 pm, 16th March 2025

The assignment must be completed individually. You are permitted to use the Internet and any printed references, though your answers and code must be your own!

Supplementary material. As part of this assignment, you are provided with two Python source files: • `problem.py` implements the challenges and the test harness. • `solution.py` provides a starter template for your solution. Please refer to these files for more details.

Submission instructions. This assignment includes both written and programming components. Submit your answers to the written portion as a single PDF file under "Assignment 3 - Written" on Gradescope. For the programming part, submit your completed `solution.py` file under "Assignment 3 - Programming" on Gradescope.

1 Programming

Problem 1

In this problem, you are given an ElGamal public key $\mathsf{pk} = g^{\mathsf{sk}} \in \mathbb{Z}_p^*$, where p is a large prime[1]. However, the public key has two potential issues: (1) the order of $\mathbb{Z}_p^*$, which is $\varphi(p) = p-1$, has small prime factors[2] that are exponentially smaller than $p-1$, and (2) to ensure efficient decryption[3], the scheme uses a "short" secret key sk e.g., sk is only 128-bits long even when p is 1024-bits (note that standard ElGamal requires sampling the secret key from $\mathbb{Z}_{p-1}$). It turns out the combination of these two issues is catastrophic, allowing an attacker to completely recover the ElGamal secret key from the public key.

This was, in fact, a real vulnerability in some implementations of TLS, as discussed by Adrian et al. (See Section 3.5, "Attacks on composite-order subgroups"). These implementations failed to carefully sample the prime p and used short secrets to allow efficiently computing ephemeral keys, ultimately compromising security.

Recall that the security of the ElGamal encryption scheme is based on the Decisional Diffie-Hellman (DDH) assumption, which, in turn, is closely related to the hardness of computing discrete logs. However, as we discussed in class, ensuring the hardness of the discrete log problem requires careful selection of the group used for computations. In particular, the Pohlig-Hellman algorithm can efficiently compute the discrete log when the group order is smooth i.e., $p - 1 = \prod_{i=1}^{n} p_i^{e_i}$, where each p_i is relatively small. In this problem, you will progressively work through the steps required to implement the Pohlig-Hellman algorithm and then use it to recover the secret key from the ElGamal public keys affected by the vulnerabilities described above.

[1] Recall that p is at least 1024 bits, and modern protocols use 3072-bit primes.

[2] Note that $p-1$ may have large prime factors as well.

[3] Decryption requires computing an exponentiation h^{sk} where $h \in \mathbb{Z}_p^*$. Thus, a short secret key allows for faster decryption.

Additional resources. Before answering the questions below, you may find it helpful to review the basics of number theory covered in class. We also recommend reading through the comments in `problem.py`.

Some additional references include Section 2.7 of A Computational Introduction to Number Theory and Algebra by Victor Shoup (Shoup-NTB) and Section 2.5 of the Handbook of Applied Cryptography by Menezes, Oorschot and Vanstone (HAC). Additional references specific to each problem are provided below. You might also find Wikipedia articles on these algorithms a helpful starting point.

1. **Brute Force Discrete Log:** Implement the `brute_force_dl` function in `solution.py`. This function takes as input the modulus p, a group element $g \in \mathbb{Z}_p^*$, the order q of the subgroup generated by g, and a target element $h \in \langle g \rangle$. It performs a brute force search to find the discrete log of h with respect to g by iterating through every possible value in $\mathbb{Z}_q$ until it finds an x such that $g^x = h$.

   Your solution will only be tested on inputs where the subgroup order q is small enough for the brute force algorithm to complete within a few seconds.

   [5 points]

2. **Baby-Step Giant-Step Algorithm:** Implement the `baby_step_giant_step_dl` function in `solution.py`.

   This function takes the same inputs as `brute_force_dl` but uses the baby-step giant-step algorithm to compute the discrete log. Compared to the brute force algorithm, the baby-step giant-step giant step improves the runtime at the cost of using more space.

   Your solution will only be tested on inputs where the subgroup order q is small enough for the baby-step giant-step algorithm to complete within a few seconds.

   References: Section 16.1.1 of Boneh/Shoup Textbook, Section 3.6.2 of HAC, Notes by William Gasarch.

   [10 points]

3. **Chinese Remainder Theorem:** Implement the `crt` function in `solution.py`.

   This function takes as input a list of integers $(a_1, \ldots, a_n)$ and a list of moduli $(m_1, \ldots, m_n)$. The moduli are *pairwise coprime* i.e., for all $i, j \in \{1, \ldots, n\}$, where $i \neq j$, m_i and m_j are co-prime. It outputs an integer z such that

   $$\begin{aligned} z &\equiv a_1 \bmod m_1, \\ z &\equiv a_2 \bmod m_2, \\ &\vdots \\ z &\equiv a_n \bmod m_n. \end{aligned}$$

   References: Section 2.4 (Proof of Theorem 2.6) in Shoup-NTB, Stanford Crypto Article on CRT. You can use `pow(a, -1, m)` to compute the modular inverse of a modulo m in Python.

   [15 points]

4. **Pohlig-Hellman Algorithm:** Implement the `pohlig_hellman` function in `solution.py`.

   This function takes as input the modulus p, a group element $g \in \mathbb{Z}_p^*$, the *prime factorization* of the order of the subgroup q generated by g, and a target element $h \in \langle g \rangle$. It uses the Pohlig-Hellman algorithm to compute the discrete log of h with respect to g.

   Your solution will only be tested on inputs where the subgroup order q is a smooth integer.

   References: Wikipedia article, Section 16.1.2 of Boneh/Shoup Textbook, Video by Jim Fowler, Section 3.6.4 of HAC.

   [25 points]

5. **ElGamal Attack:** Implement the `elgamal_attack` function in `solution.py`.

   This function takes as input the modulus p, a generator g of $\mathbb{Z}_p^*$, the prime factorization of the group order $p-1$, a bound B on the value of the secret key, and a public key $\mathsf{pk} \in \mathbb{Z}_p^*$. It outputs the secret key sk corresponding to pk.

   Approach: At a high level, you will use your Pohlig-Hellman implementation from part (4) to compute the discrete log of pk.

   - The Pohlig-Hellman algorithm is efficient only when the subgroup order is *smooth*. However, the full group $\mathbb{Z}_p^*$ may contain very large prime factors in its order $p-1$, making direct application inefficient.
   - It is guaranteed that $\mathbb{Z}_p^*$ contains a smooth-order subgroup whose order is at least B. Identify this subgroup of $\mathbb{Z}_p^*$ whose order is large enough to *uniquely* recover the secret key.
   - Run Pohlig-Hellman over this chosen subgroup to compute sk.

   [25 points]

Written

Problem 2

Answer the following questions based on your solutions to Problem 1.

1. Summarize how the baby-step giant-step and Pohlig-Hellman algorithms solve the discrete logarithm problem in a few sentences each.

   [5 points]

2. How many steps do the brute force, baby-step giant-step and Pohlig-Hellman algorithms take in the worst case, when computing discrete log over a group of order 1031^4 (note that 1031 is a prime number)? In this analysis, consider the number of loop iterations as the measure of steps. Assume that each loop iteration, including group exponentiations and other computations, takes constant time.

   [5 points]

3. Let p be the following 1389-bit prime,

$$p = 2^{32} \cdot 1031^{46} \cdot 184467440737095516291^{4} + 1.$$

Let pk be an ElGamal public key computed over $\mathbb{Z}_p^*$, with the secret key sampled uniformly at random from $\{0, \ldots, 2^{128}\}$. Briefly explain how your solution to Problem 1.5 uses the Pohlig-Hellman algorithm as a subroutine to compute the secret key corresponding to pk.

[5 points]

4. Let p be a large *safe* prime i.e., $p = 2q + 1$, where q is also a prime number. Let pk be an ElGamal public key computed over $\mathbb{Z}_p^*$, with the secret key sampled uniformly at random from $\{0, \ldots, 2^{128}\}$. Will your solution to Problem 1.5 successfully recover the secret key corresponding to pk? Why or why not?

[5 points]