AN EXPLORATION OF THE INDEX CALCULUS

KAZUYA ERDOS

1. Introduction

A central idea to many cryptographic protocols is the utilization of *trapdoor functions*, which are functions that are easy to compute in one direction, yet very difficult to invert. Perhaps the most ubiquitous trapdoor function in cryptography is discrete exponentiation: while it is very easy to compute $g^x \pmod{p}$ with methods such as binary fastpowering, computing the inverse, $\log_g(h)$, also known as taking the *discrete logarithm*, is quite difficult. Over finite fields, known algorithms such as *Baby Step Giant Step* run in exponential time [**?**]. For Elliptic Curves, this variant of algorithm is the best known for cracking the Discrete Log Problem (DLP), which allows Elliptic Curve-based cryptography to use shorter keys while maintaining the same security as their classical $\mathbb{F}_p$-based counterparts.

That begs the question, then, what algorithms exist to crack the DLP over $\mathbb{F}_p$? In this short paper we will explore the **Index Calculus**, an algorithm that utilizes the nature of B-smooth numbers to break the DLP into smaller, more manageable problems. A discrete logarithm used to be called an *index* [**?**], and the process of working over many smaller discrete logarithms, or indices, gives the algorithm its name. The algorithm runs in subexponential time, a strong improvement over the best-known exponential algorithms for Elliptic Curves.

2. Background

We assume that the reader is familiar with group theory, modular arithmetic, and the Extended Euclidean Algorithm. Below are some essential definitions and theorems to understand Index Calculus.

Definition 2.1. (B-smooth) [**?**]. A number is whose prime factors are all less than or equal to B is called a B-*smooth* number.

Definition 2.2. (Prime counting function) [**?**]. For any number X, let

$$\pi(X) = (\text{\# of primes } p \text{ satisfying } 2 \leq p \leq X)$$

Definition 2.3. (Square-free integers) [**?**]. An integer $n \in \mathbb{Z}$ is called *square-free* if it is divisible by no square number other than 1.

Theorem 2.4. *(Sun Tzu Remainder Theorem)* [**?**]. *If* $\gcd(m_i.m_j)$ *for* $i \neq j$, *then the system of congruencies*

$$\begin{aligned} x &\equiv b_1 \pmod{m_1} \\ x &\equiv b_2 \pmod{m_2} \\ &\vdots \\ x &\equiv b_l \pmod{m_l} \end{aligned}$$

Has a unique solution modulo $m = m_1 \cdot m_2 \cdots m_l$

We give a proof and describe an algorithm to implement Theorem **??** in section **??**. We can now move to our main result.

3. Mathematical Description of the Algorithm

Now, we describe the algorithm. Suppose we'd like to solve the Discrete Log Problem

$$g^x \equiv h \pmod{p} \tag{1}$$

First, we fix a bound B and randomly choose integer values for k until we find an instance of the quantity

$$h \cdot g^{-k}$$

that is B-smooth. For this value of k we can write

$$h \cdot g^{-k} \equiv \prod_{\ell \leq B} \ell^{e_\ell} \pmod{p} \tag{2}$$

For every prime $\ell \geq B$ and for some exponents e_ℓ. Taking the discrete logarithm base g of both sides of (**??**) yields

$$\log_g(h) \equiv k + \sum_{\ell \leq B} e^{\ell} \cdot \log_g(\ell) \pmod{p-1} \tag{3}$$

Note that exponents in $(\mathbb{Z}/p\mathbb{Z})^\times$ can be considered modulo $p-1$. Now, we must compute each discrete logs $\log_g(\ell)$ for smaller primes ℓ. To accomplish this, we randomly select exponents i and compute

$$g_i = g^i \quad \% \quad p$$

Until we find at least $\pi(B)$ instances of g_i that are B-smooth. For each g_i that is B-smooth we can write

$$g_i = g^i = \prod_{\ell \leq B} \ell^{u_\ell(i)}$$

Again, we can take the discrete logarithm base g of both sides to find

$$i \equiv \sum_{\ell \leq B} u_l(i) \cdot \log_g(\ell) \pmod{p-1} \tag{4}$$

This gives us a system of at least $\pi(B)$ equations with unknowns $\log_g(\ell)$. Note that since there are $\pi(B)$ values that ℓ can take on, it is essential for us to generate at least $\pi(B)$ equations. We can now solve this linear system to find each value of $\log_g(\ell)$. However, there is one small problem: Since $p-1$ is composite, we cannot use Gaussian elimination to solve the system modulo $p-1$. Instead, we must factor $p-1 = q_1^{e_1} \cdots q_n^{e_n}$ and solve the system for each prime factor q_j.

Remark 3.1. For this implemention, we make the simplifying assumption that each $e_j = 1$, meaning that $p-1$ is square-free. To remove this assumption from our implementation, we would need to lift ecah of our solutions from $\mathbb{Z}/q_j\mathbb{Z}$ to $\mathbb{Z}/q_j^{e_j}\mathbb{Z}$, though this was determined to be out of scope. [**?**]

Finally, we combine all of the solutions modulo the relatively prime bases q_j to generate a single solution modulo $p-1$ via the Sun Tzu Remainder Theorem algorithm described in **??**. After finding each of our values of $\log_g(\ell)$, we plug them into (**??**) with value of k we chose to find $\log_g(h)$, and the algorithm is complete.

4. Implementation of the Algorithm

The full algorithm was implemented in Python, with Sagemath being used for Gaussian elimination. Some of the steps were chosen to be optimized, while others were implemented naively. The intention of this was to get a working implementation of the algorithm as a baseline, then adding optimizations on top. This section serves to break down some of the interesting implementation details.

4.1. **High-level Pseudocode.**

Here is a very high-level overview of the implementation. For complete details, please see the provided code.

```
Input: $g$: DLP base, $h$: power of $g$, $p$: large prime, $B$: bound
function solveDLP($g, h, p, B$)
    Find all primes $\ell \leq B$
    Generate $\pi(B)$ random powers $g^i$ that are $B$-smooth
    Factor $p - 1$ into $q_1^{e_1} \cdots q_n^{e_n}$
    if $e_i > 1$ then
        return Exception
    $success \leftarrow False$
    while $success == False$ do
        $aug \leftarrow$ Augmented matrix from random powers
        $R \leftarrow rref(aug)$
        $system\_solns \leftarrow$ solutions from $R$
        if $system\_solns$ not unique then
            Retry with new random powers
        else
            $success \leftarrow True$
    $\log_g(\ell) \leftarrow$ Solved STRT system
    while $True$ do
        $k \leftarrow randint(1, p - 1)$
        if $hg^{-k}$ is $B$-smooth then
            $pows \leftarrow$ powers of prime factors of $hg^{-k}$
            return $k +$ linear combination of $pows \cdot \log_g(\ell)$
```

4.2. **Finding and facotring B-smooth numbers.** A crucial part of the Index Calculus algorithm is determining whether a number n is B-smooth. Additionally, we must also express

$$n = \prod_{\ell \leq B} \ell^{e_\ell}$$

To find all of the primes $\ell \leq B$, we use the classic Sieve of Eratosthenes:

```
def find_prime_base(B):
    primes = [True for i in range(B + 1)]
    p = 2
    while (p ** 2 <= B):
        if primes[p]:
            for i in range (p ** 2, B + 1, p):
                primes[i] = False
        p += 1
    output_primes = []
    for p in range(2, B + 1):
        if(primes[p]):
            output_primes.append(p)
    return output_primes
```

With this power base of $\ell \leq B$, we can determine whether a number is B-smooth:

```
def find_smooth_powers(n, factor_base):
    output = []
    for b in factor_base:
        power = 0
        while n % b == 0:
            power += 1
            n = n // b
        output.append(power)
    if n == 1:
        return output
    else:
        return None
```

The idea of the algoritihm is to iterate through the factor base (previously referred to as the *prime base*) and pull off as many powers of each prime from n as possible, then see if the resulting quotient is 1 after completion. If so, then n is B-smooth and we can return the factorization, otherwise we return None.

4.3. **Sun Tzu Remainder Theorem.** Conveniently, the proof of the Sun Tzu Remainder Theorem gives us a very natural way to implement it with code. We provide a high-level proof here:

Proof of Theorem **??**. [**?**] From the given system of congruencies, we let $m = m_1 \cdots m_l$, and set $n_i = \frac{m}{m_i}$ for all i. Notice that $\gcd(n_i, m_i) = 1$ and $\gcd n_i, m_j = m_j$ if $j \neq i$. By the Extended Euclidean Algorith, $\exists r_i, s_i \in \mathbb{Z}$ such that $r_i m_i + n_i s_i = 1$. Set $e_i = n_i s_i$. Observe that $e_i \equiv 1 \pmod{m_i}$ and $e_i \equiv 0 \pmod{m_j}$. So, to satisfy all of the congruencies, we set our unique solution to

$$x \equiv \sum_{i=1}^{l} b_i e_i \pmod{m}$$

□

This leads to a very natural programming implementation:

```
def solve_single_strt(m, strt_in):
    soln = 0
    for m_i, b_i in strt_in:
        n_i = m // m_i
        _, s_i = ext_gcd(m_i, n_i)
        e_i = n_i * s_i
        soln += b_i * e_i
    return soln % m
```

4.4. **Solving Linear Systems with Sagemath.** We utilized Sagemath to solve the linear systems described in the algorithm:

```
prime_factorization = prime_factor(p - 1)
sun_tzu_consts = []
for q_i, e_i in prime_factorization:
    # NOTE: Simplifying assumption that p-1 is square-free
    if e_i > 1: raise Exception("Out of scope: p-1 is not square-free")
    coeffs_matrix = Matrix(GF(q_i), coefficients)
    consts_matrix = vector(GF(q_i), powers_i)
    aug = coeffs_matrix.augment(consts_matrix)
    R = aug.rref()
    system_solns = []
    for i in range(pi_B):
        if R[i, i] != 1: # Extract from pivot rows
            continue
        system_solns.append(R[i, pi_B])
    sun_tzu_consts.append((q_i, system_solns))
sun_tzu_solved = solve_strt_system(sun_tzu_consts, pi_B)
```

The strategy is as follows: We iterate through the prime factors of $p-1$, raising an exception if our assumption of $p-1$ being square-free is violated. Then, we construct an augmented matrix modulo $\mathbb{F}_{q_i}$ of the coefficients and constants found in equation **??**. We solve them by Gaussian Reduction through Sagemath's **rref()** function, then combine the solutions with the Sun Tzu Remainder Theorem algorithm described in **??**.

Remark 4.1. It is possible for the system generated to not have unique solutions, due to linear dependence between rows of the augmented matrix. The full version of the code implementation checks for this case, and naively retries with new randomly generated powers. An optimization of this would be to ensure that the system is solvable as it is being generated.

5. Short Discussion on Runtime

As seen earlier, our implementation is not fully optimized, but it does meet the **subexponential** runtime expectations of the Index Calculus. The main bottleneck is finding $\pi(B)$ powers $g^i \pmod p$ that are B-smooth. As we use a sieve method to quickly check if numbers are B-smooth, from [**?**] we can estimate that our running time is on the order of

$$e^{\sqrt{2(\ln p)(\ln \ln p)}}$$

6. Running and testing the code

The program was developed and tested locally on a device running an Apple ARM processor. The following tools were used:

(1) **python 3.12.0**, installed from [**?**]
(2) **sage 10.3**, installed from Homebrew, a MacOS package manager [**?**]
(3) **unittest**, which is included as a part of Python

Three files exist as part of the program: **index_calc.py** for the main functionality, **helpers.py** for support functions, and **test_index_calc.py** for testing. Each of these files imports sage as follows:

```
from sage.all import *
```

The main **index_calc.py** file can be run directly with the sage binary from the command line. This prompts the user to enter values for g, h, p, and B, and solves the DLP with Index Calculus (if possible):

```
sage index_calc.py
```

We use the Python **unittest** library for testing. To run tests, you must first enter the Sagemath shell to gain access to all of its modules:

```
sage -sh
```

Once inside of the shell, we can run all tests:

```
python3 -m unittest test_index_calc.UnitTests
```

or a single test:

```
python3 -m unittest test_index_calc.UnitTests.test_textbook
```