University College London
Department of Computer Science

Cryptanalysis Lab 4

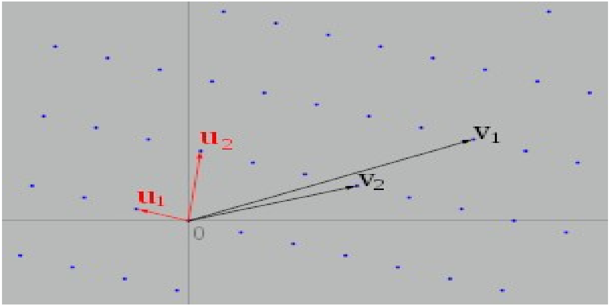
J. P. Bootle

May 27, 2018

jonathan.bootle.14@ucl.ac.uk
Version 2.0

The LLL Algorithm

Given a set of basis vectors $\mathcal{S} = \mathbf{x}_1, \ldots, \mathbf{x}_n$ with integer entries, we can form a lattice $\mathcal{L}$ by taking all integer linear combinations of vectors in $\mathcal{S}$. The picture below shows a lattice generated by vectors in $\mathbb{Z}^2$. The same lattice can be generated by $\mathbf{u}_1, \mathbf{u}_2$, or by $\mathbf{v}_1, \mathbf{v}_2$.



The LLL algorithm takes a collection of 'bad' basis vectors for lattice, such as $\mathbf{v}_1, \mathbf{v}_2$, and tries to generate a set of 'good' basis vectors for the lattice, such as $\mathbf{u}_1, \mathbf{u}_2$, which are much shorter, and close to being perpendicular.

In the next two questions, you will use the LLL algorithm to solve problems related to cryptography.

A Knapsack-Based Hash Function

We can try to construct a hash function based on the hardness of solving the Knapsack problem.

Let $a_1, \ldots, a_n$ be positive integers. Given a positive integer s, we might ask whether there exist $x_1, \ldots, x_n \in \{0, 1\}$ such that $\sum_{i=1}^n a_i x_i = s$. This is a special case of the knapsack problem, and it is NP-complete, which suggests that the function $(x_1, \ldots, x_n) \mapsto \sum_{i=1}^n a_i x_i$ might be difficult to invert, and have some of the properties of a good hash function.

Concretely, we create a hash function by choosing random values for the $a_1, \ldots, a_n$. We hash values $(x_1, \ldots, x_n) \in \{0, 1\}^n$ to $\{0, 1\}^k$ by computing $s = \sum_{i=1}^n a_i x_i$, and then taking the binary digits of s as output.

Click on the green letter before each question to get a full solution. Click on the green square to go back to the questions.

Exercise 1.

(a) Implement a function 'Parameters' which takes integers n, μ as input, and generates n random μ-bit integers $a_1, \ldots, a_n$ for use in a knapsack-based hash function.

(b) Create a function 'KnapsackHash' which implements the knapsack-based hash function described above. Your function should take the output of part a) and a value to hash, and produce a hash value.

(c) Consider the lattice generated by the rows of the following matrix, for some large value K.

$$\begin{pmatrix} Ka_1 & 1 & 0 & \cdots & 0 \\ Ka_2 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ Ka_n & 0 & 0 & \cdots & 1 \end{pmatrix}$$

How might finding a short vector in this lattice help to find a collision in the knapsack hash function?

Hint: What happens if we find a short vector with first component zero, and the other components 1 or -1?

(d) The LLL algorithm can be applied to a matrix M by writing M.LLL(). On input a square matrix of row vectors, the LLL algorithm produces a new matrix, where the first row is a short vector in the lattice. Write a program which uses the LLL algorithm to break a knapsack hash function with `params = Parameters(n,mu)` for $(n, mu) = (10, 10), (20, 20), (40, 40)$. What are the largest values of (n, mu) for which your program finds a collision?

Finding Polynomials with Small Coefficients from Approximate Roots

Taken from Algorithmic Cryptanalysis, Chapter 13, Exercise 1: Consider the floating point number:

$$x = 8.44311610583794550393138517.$$

Show that x is a close approximation of a real root of a polynomial of degree 3, with integer coefficients bounded by 20 (in absolute value).

Click on the green letter before each question to get a full solution.

Click on the green square to go back to the questions.

Exercise 2.

(a) With the collision matrix from the previous question in mind, design a matrix containing the powers of x, where a short vector in the lattice is likely to produce a polynomial of degree 3, with x as a root.

(b) Apply the LLL algorithm to your matrix to find the polynomial.

Solutions to Exercises

Exercise 1(a) The following code implements the 'Parameters' function.

```
def Parameters(n,mu):
    A = list();
    for i in range(0,n):
        A.append(randint(0,2**mu-1))
    return [vector(A),n,mu]
```

□

Exercise 1(b) The following code implements the knapsack hash function.

```
def KnapsackHash(params,x):
    A = params[0]
    n = params[1]
    mu = params[2]
    k = ceil(log(n,2))+mu
    if n != len(x):
        return "The input vectors are not the same length!"
    z = 0;
    for i in range(0,len(x)):
        z = z + A[i]*x[i];
    z = z.bits()
    while len(z) < k:
        z.append(0)
    return z
```

□

Exercise 1(c) Following the hint, since K is large, short vectors in the lattice are likely to have first component equal to zero. Otherwise, the first component would be a large number, as a multiple of K. This means that a short vector is likely to involve finding a linear combination of the a_i which is equal to zero. The other components of the vector tell us the coefficients in this linear combination. If the other components are all 1 or -1, then we can rearrange the linear combination to find two binary inputs which hash to the same output value. □

Exercise 1(d) The following code implements a collision finder.

```
def BreakKnapsackHash(params,K):
#Choose large positive integer K
    A = params[0]
    n = len(A)
    B = matrix(A).transpose()
    C = matrix.identity(n)
    M = block_matrix([[K*B,C]])
    L = matrix(list(M.LLL()))
    L = L[0]
    if L[0] != 0:
        return 'fail'
    L = list(L)          #(continued on next page)
    L.remove(0)
    for entry in L:
```

```
        if abs(entry) > 1:
            return 'fail'
    X1 = matrix([[abs(L[i]>0) for i in range(0,len(L))]])
    X2 = matrix([[abs(L[i]<0) for i in range(0,len(L))]])
    return matrix(list(block_matrix([[X1],[X2]])))
```

□

Exercise 2(a)

$$\begin{pmatrix} \lfloor K \rceil & 1 & 0 & 0 & 0 \\ \lfloor Kx \rceil & 0 & 1 & 0 & 0 \\ \lfloor Kx^2 \rceil & 0 & 0 & 1 & 0 \\ \lfloor Kx^3 \rceil & 0 & 0 & 0 & 1 \end{pmatrix}$$

Apply the LLL algorithm to the lattice. The first element in our reduced basis has the form $(\epsilon, a_0, a_1, a_2, a_3)$, where $\epsilon = a_0\lfloor K \rceil + a_1\lfloor Kx \rceil + a_2\lfloor Kx^2 \rceil + a_3\lfloor Kx^3 \rceil$ and ϵ is quite small. Dividing by K, this suggests that x is a close approximation to a root of the polynomial with coefficients a_i. □

Exercise 2(b) Section 13.1.2.2 of Algorithmic Cryptanalysis suggests using $K \geq (\max|a_i|)^{2d}$ where d is the degree of the polynomial. So in our example we can take $K = (20)^6 = 64 \times 10^6$. This gives us a matrix:

$$\begin{pmatrix} 64000000 & 1 & 0 & 0 & 0 \\ 540359431 & 0 & 1 & 0 & 0 \\ 4562317413 & 0 & 0 & 1 & 0 \\ 38520175629 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Apply the LLL algorithm. For your own sanity, use SAGE rather than trying to do LLL by hand. Create a matrix A as above, do A.LLL() and look at the first row. This is $(-3, -10, -11, -7, 1)$, corresponding to $x^3 - 7x^2 - 11x - 10$. Check for yourself that $x^3 - 7x^2 - 11x - 10$ is extremely close to 0. □