Lab Session: Finite Fields and Number Theoretic Reference Problems

Chan Nam Ngo
channam.ngo@unitn.it
University of Trento, Trento, Italy

September 28, 2018

1 Installation of the libsnark library

All the lab sessions assume the Linux platform (e.g. Ubuntu). Students using other platforms should find equivalent alternatives. An easy solution would be VirtualBox[1] + Ubuntu[2].

Students should follow the instruction on `https://github.com/scipr-lab/libsnark` for installation. The libsnark library is useful because it provides almost all necessary dependencies for all our lab sessions.

- GMP for handling big integers
- libff for elliptic curve finite fields
- libsnark itself is a zk-SNARK library

Other dependencies such as libssl (which also includes libcrypto) are also useful for some crypto tasks, e.g. random number generations, etc.

2 Warm up with small integers

Students only need to use standard C/C++ library for doing this exercise. The goal is to implement the following 3 fundamental algorithms for small integers.

2.1 Extended Euclidean algorithm in $\mathbb{Z}$

The Extended Euclidean algorithm, takes as input two integers a and b, returns not only the gcd (greatest common divisor) of a and b, but also two other integers x and y such that $a \cdot x + b \cdot y = \gcd(a, b)$.

[1] `https://www.virtualbox.org/wiki/Downloads`
[2] `https://www.ubuntu.com/`

In case a coprimes b, i.e. $gcd(a, b) = 1$, the Extended Euclidean algorithm is very useful, as it yields almost no overhead but returns x, which is the modular multiplicative inverse of a modulo b, and y is the modular multiplicative inverse of b modulo a.

(gcd,x,y) = EGCD(a,b)

```
procedure EGCD(a, b)
    if b == 0 then
        return (a, 1, 0)
    end if
    x_2 = 1, x_1 = 0, y_2 = 0, y_1 = 1
    while b > 0 do
        q = a/b, r = a − q ∗ b, x = x_2 − q ∗ x_1, y = y_2 − q ∗ y_1
        a = b, b = r, x_2 = x_1, x_1 = x, y_2 = y_1, y_1 = y
    end while
    return (a, x_2, y_2)
end procedure
```

Students can check the algorithm output against the online implementation at `https://planetcalc.com/3298/` for correctness.

2.2 Computing multiplicative inverse in $\mathbb{Z}_n$

Students will use the implemented Extended Euclidean algorithm for computing the multiplicative inverse of an integer a in $\mathbb{Z}_n$.

$\boldsymbol{x = MultiplicativeInverse(a, n)}$

```
procedure MultiplicativeInverse(a, n)
    (d, x, y) = EGCD(a, n)
    if d > 1 then
        return null
    end if
    return x
end procedure
```

2.3 Repeated square-and-multiply algorithm for exponentiation in $\mathbb{Z}_n$

Let the binary representation of k be $\{k_i\}_{i=0}^{t}$, the modular exponentiation a^k for $a \in \mathbb{Z}_n$ can be efficiently computed as $a^k = \prod_{i=0}^{t} a^{k_i 2^i}$.

$\boldsymbol{b = RSM(a, \{k_i\}_{i=0}^{t}, n)}$

```
procedure RSM(a, {k_i}_{i=0}^t, n)
    if k == 0 then
        return 1
    end if
```

  $A = a$
  if $k_0 == 1$ **then**
    $b = a$
  end if
  for $(i = 1; i < t; i++)$ **do**
    $A = A^2 \mod n$
    if $k_i == 1$ **then**
      $b = A \cdot b \mod n$
    end if
  end for
  return b
end procedure

Students can check the output of the algorithm against the trivial version, i.e. k multiplications of $a \mod n$, for correctness.

Can you also explain the complexity difference between RSM and the trivial version?

3 Handle big integers using the GMP library

Students will use the GMP library (a dependency of the libsnark library) to handle big integers. Repeat the three algorithms in §2.

To use and link the GMP library, follow the instructions at `https://gmplib.org/manual/Headers-and-Libraries.html`. Below you can find a simple example. All arithmetic operations for Big Integer in GMP require function calls.

The *add_example.c* is as follows.

```
{
#include <stdio.h>/* for printf */
#include <gmp.h>

int main(int argc, char *argv[])
{
    mpz_t a, b; /* working numbers */
    if (argc<3)
    { /* not enough words */
        printf("Please supply two numbers to add.\n");
        return 1;
    }
    mpz_init_set_str (a, argv[1], 10);
    /* Assume decimal integers */
    mpz_init_set_str (b, argv[2], 10);
    /* Assume decimal integers */
    mpz_add (a, a, b); /* a=a+b */
```

```
    printf("%s + %s => %s\n", argv[1],
          argv[2], mpz_get_str (NULL, 10, a));
    return 0;
}
```

It can be built using the following command.

```
ubuntu:~$ gcc -o add_example add_example.c -lgmp -lm
```

See `https://gmplib.org/manual/Integer-Functions.html` for references.

Students can use the following functions of `openssl` to do some sub-tasks.

- openssl-prime[3] for generating prime numbers for testing purposes.

```
ubuntu:~$ openssl prime -generate -safe -bits 128
329720161372808576669651325053333255543
```

- openssl-dhparam[4] for generating Diffie-Hellman parameters (big prime and generator) for testing purpose.

```
ubuntu:~$ openssl dhparam -text 128
Generating DH parameters, 128 bit long safe prime,
generator 2
This is going to take a long time
...
DH Parameters: (128 bit)
prime:
00:94:42:54:bd:bc:ee:37:f5:81:e2:0c:c6:10:a5:
6d:8b
generator: 2 (0x2)
```

The online implementation at `https://planetcalc.com/3298/` is also applicable for big integers.

4 The Discrete Logarithm problem

Students will try to solve the DLOG problem using two algorithms (1) Exhaustive Search and (2) Baby-Step Giant-Step.

*** SHOULD TEST WITH SMALL NUMBERS FIRST***

[3] `https://www.openssl.org/docs/manmaster/man1/openssl-prime.html`
[4] `https://www.openssl.org/docs/manmaster/man1/dhparam.html`

4.1 Exhaustive Search using Repeated square-and-multiply algorithm in $\mathbb{Z}_p^*$

procedure ExhaustiveDLOG1(p, g, y)
    for $(k = 0; k < p; k++)$ **do**
        Let the binary representation of k be $\{k_i\}_{i=0}^{t}$
        if $y == \mathsf{RSM}(g, \{k_i\}_{i=0}^{t}, p)$ **then**
            return k
        end if
    end for
end procedure

Can you suggest another variant instead of deterministically going through k from 0 to p? Can we pick k in a better way (by a heuristic)?

Can the following variant provide the correct answer? Explain why.

procedure ExhaustiveDLOG2(p, g, y)
    for $k = 0; \boldsymbol{k < \sqrt{p}}; k++$ **do** ▷ Only consider k up to $\sqrt{p}$
        Let the binary representation of k be $\{k_i\}_{i=0}^{t}$
        if $y == \mathsf{RSM}(g, \{k_i\}_{i=0}^{t}, p)$ **then**
            return k
        end if
    end for
end procedure

4.2 Baby-Step Giant-Step

procedure BSGSDLOG(p, g, y)
    $m = \sqrt{n}$
    $J = \emptyset$
    for $(j = 0; j < m; j++)$ **do**
        $J = J \cup (j, g^j)$ ▷ g^j should be evaluated using RSM
    end for
    for $(i = 0; i < m; i++)$ **do**
        $z = y \cdot g^i$
        if $((j, z) \in J)$ **then**
            return $i * m + j$
        end if
    end for
end procedure

How should J be stored and looked-up? Explain why.

5 The Diffie-Hellman Key Exchange protocol

Students need to implement the DHKE protocol as follows.

Alice		**Bob**
$x \leftarrow_{\$} \mathbb{Z}_p$		
$X \leftarrow g^x$		
	$\xrightarrow{\mathbb{G}, p, g, X}$	
		$y \leftarrow_{\$} \mathbb{Z}_p$
		$Y \leftarrow g^y$
	$\xleftarrow{Y}$	
$\mathsf{k}_A \leftarrow Y^x$		$\mathsf{k}_B \leftarrow X^y$

Write the algorithm $(x, g^x) \leftarrow \mathsf{KeyGen}(p, g)$ and $(g^{xy}) \leftarrow KeyAgree(p, g, x, g^y)$.

5.1 Breaking DH with DLOG

Write an algorithm to use a DLOG solving algorithm (e.g. $\mathsf{BSGSGLOG}$) to solve the following Diffie-Hellman problems.

procedure CDH(p, g, a, b)
    Let $a = g^x$, $b = g^y$
    return g^{xy}
end procedure

procedure DDH(p, g, a, b, c)
    Let $a = g^x$, $b = g^y$, $c = g^z$
    if $(z = xy)$ **then**
        return true
    else
        return false
    end if
end procedure

Can we also use CDH or DDH to solve DLOG? Write such algorithms.

5.2 Man In The Middle attack

Alice		**Eve**		**Bob**
$x \leftarrow_{\$} \mathbb{Z}_p$				
$X \leftarrow g^x$				
	$\xrightarrow{\mathbb{G}, p, g, X}$			
		$z \leftarrow_{\$} \mathbb{Z}_p$		
		$Z \leftarrow g^z$		
	$\xleftarrow{Z}$		$\xrightarrow{\mathbb{G}, p, g, Z}$	
				$y \leftarrow_{\$} \mathbb{Z}_p$
				$Y \leftarrow g^y$
			$\xleftarrow{Y}$	
$\mathsf{k}_{AE} \leftarrow Z^x$		$\mathsf{k}_{AE} \leftarrow X^z$		$\mathsf{k}_{BE} \leftarrow Z^y$
		$\mathsf{k}_{BE} \leftarrow Y^z$		

Write a test program using KeyGen and KeyAgree above to simulate the MITM attack.