Proposal For C%: A Language For Cryptographic Applications

Maggie Mallernee, Zachary Silber,
Michael Tong, Richard Zhang, Joshua Zweig

UNIs: mlm2299, zs2266, mct2159, rz2345, jmz2135

1 Describe the language that you plan to implement

Our language is a C-like language designed for applications in cryptography and abstract algebra. The familiar and robust C-like nature of our language will provide programmers with customary and essential tools for building a wide range of applications and tools. Additionally, our selection of built in types coupled with a lightweight syntax designed to handle modular operations will suit this language for implementing cryptographic protocols.

In today's world, modular operations underpin many important applications, ranging from hashing to cryptography. However, most programming languages offer very little support for the rich world of moduli. Our language will offer this support in the form of built in types and operators to remove the programmer's burden of writing difficult series of modular expressions. In doing so, the workflow of writing programs for modular heavy arithmetic will become streamlined, and the syntax natural and intuitive. With additional support for arithmetic on elliptic curves, this language ought to be a cryptologists'delight.

2 Explain what sorts of programs are meant to be written in your language

2.1 General background

We start with some general background and motivation for the subject. Most common cryptographic protocols are in the following setting: you start with some prime number p, say 7, and consider the number system of positive integers less than this prime where the operation is modular multiplication (i.e., take two numbers less than 7, multiply them as usual, and then take their remainder when divided by 7). By general theory, there exists some integer α less than p such that, if you continually multiply it by itself and then take its remainder when divided by 7, it will at some point equal every positive integers less than 7; in this case 3 satisfies this property. We call 3 a primitive mod 7. Now, the goal behind cryptographic protocols is to design a procedure in which two parties first generate random private keys (typically just random numbers less than p) so that they can share obfuscated information and be confident that, even if someone sees their interaction, they won't be able to read the underlying message. Thus, modular arithmetic is at the heart of such protocols and is widely applicable throughout cryptography.

From a more abstract point of view, however, the core of what we are working with here is a set of things (in this case, positive integers less than p) with a binary operation (in this case, modular multiplication). In particular, the binary operation of modular multiplication enjoys nice properties: it is associative (that is, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$), it has an identity element (for every a, we have $1 \cdot a = a$), and it can be reversed (for every a,

there is some other element b so that $a \cdot b = 1$). Indeed, in abstract algebra, a set equipped with a binary operation satisfying these properties is called a group, and cryptography can be done with general groups instead of just on modular arithmetic. Historically, the latter has been the standard for real-life implementation for various reasons, however our language will offer tremendous support for a newer brand of cryptography where the group in question consists of points lying on an object called an elliptic curve. How this works is highly technical, but essentially elliptic curves offer a new system for doing cryptography, with its own various intricacies when compared to the standard model.

2.2 Programs

Our language will be meant for implementing these cryptographic protocols, either for practical or educational purposes. For simplicity, the following examples will be confined to the modular arithmetic case, as opposed to the elliptic curve system. That said, with our language's support for elliptic curves, the knowledgeable user will be able to easily implement a protocol in this system without having to manually define the rather technical rules which pervade these techniques.

For practical purposes, if one wanted to implement an RSA cryptosystem for receiving messages, one could publicize a large product of primes pq (these primes would not be automatically generated by our program, but chosen by the user) and exponent e. Then one could use our language to accept incoming encrypted messages and decrypt them using the decryption exponent d. This can easily be modified to implement a more realistic protocol that depends on the theory of RSA, such as an SSL handshake.

For educational purposes, having a language which allows for abstract cryptographic protocols to be easily implemented can help make the concepts more concrete. It can even help convince a student that a protocol actually works! As an example, there is the so-called coin-flipping protocol which allows two people, Alice and Bob, to essentially flip a coin with full trust without either of them being in the same room (the only flaw is that one of the players can choose to lose without the other knowing). The protocol takes some background to explain, and even with the knowledge it may not be convincing that it actually works. One could, using our language, easily implement the protocol and see that, indeed, about half of the time Alice wins the coin toss and about half of the time Bob wins the coin toss. And indeed, going through the process of coding this protocol will make the logic involved more clear, helping the student understand the concepts involved.

3 Explain the parts of your language and what they do

We are going to maintain the standard primitives of Integer, Pointer, Array, and Struct. These are all things that will be relevant to the use cases of our language. Our language will also allow the user to write For and While loops as well as If statements. These control flow tools will allow the user flexibility in utilizing the main types of our language. We have determined that the most relevant building blocks for solving problems in the domain of cryptography and abstract algebra are ModIntegers, Curves, and Points, so we will include these as built-in types.

First, we discuss ModIntegers. Many of the problems we want to solve require working with integers under a certain modulus. This stems from the concept of finite fields

in abstract algebra, which forms a theoretical basis for many cryptographic protocols. Essentially, our language will be able to easily support operations within a given finite field, meaning that we will override the addition and multiplication operators (along with their inverses) to work on integers under a given modulus. For example, under the modulus of 7,

$$\begin{aligned} 4 \cdot 3 &\equiv 5 \pmod 7 \\ 6 + 5 &\equiv 4 \pmod 7 \\ 4 - 6 &\equiv 5 \pmod 7 \end{aligned}$$

It is of course possible to do this arithmetic with regular Int's and a built-in remainder operation; however, it becomes cumbersome to include the remainder operation in every step that changes any values. Additionally, the mod operator and the remainder calculator are actually slightly different, and different programming languages deal with this in different ways. C, for example, defines the % operator as the remainder and not the mod operator, which becomes clear when dealing with negative numbers. Consider the following code snippet:

```c
#include <stdio.h>

int rem(int a, int b)
{
    int r = a - ((a/b) * b);
    return r;
}

int mod(int a, int b)
{
    int r = rem(a, b);
    return (r > 0) ? r : r + b;
}

int main()
{
    printf("The remainder rem(-5, 3) = %d\n", rem(-5, 3));
    printf("The mod answer to mod(-5, 3) = %d\n", mod(-5, 3));
    printf("The percent sign operator -5 % 3 = %d\n", (-5 % 3));
    return 0;
}
~
~
```

The output of this snippet demonstrates the difference between the remainder and mod operators:

```
-bash-4.1$ gcc mod_test.c
-bash-4.1$ ./a.out
The remainder rem(-5, 3) = -2
The mod answer to mod(-5, 3) = 1
The percent sign operator -5 % 3 = -2
```

Some languages, like Haskell, do define both a remainder and a mod operator. We would like to maintain C-like syntax and the % operator as it is in C, while adding the ability to easily work with integers under a given modulus. Instead of having to write a separate function to correct for the % operator as one would have to in C, our ModIntegers will automatically give the correct results for operations defined in modular arithmetic.

ModIntegers will be a built-in type defined with two integers; a current value and an immutable modulus. In theory, dealing with the extremely large primes used in modern cryptography would necessitate a different method of storing this information, such as storing the int and a separate pointer (like a smart pointer) to a single stored copy of the prime. That way, the process of verifying that two ModIntegers have the same modulus could be done by comparing pointers rather than comparing the very large primes. This will be a frequent comparison, as operations can only apply to ModIntegers under the same modulus.

Next, we have Elliptic Curves and Points on that curve. Elliptic curves may be defined by three integers, or two ModIntegers of the same type. Most standard operations like addition and multiplication do not make sense between two elliptic curves. However, Curves are going to be essential in working with Points. We will define each of our Points with one Curve that can't be changed as well as two Int's that represent the current value of the Point. When you consider operations between points, they are all under a certain curve. Above we explained that it only makes sense to consider operations between two ModIntegers under the same modulus. Similarly, we only consider operations between two points on the same elliptic curve. Standard operations like addition, multiplication, etc. are mathematically defined between two points on a curve and we can implement them through built-in operators.The built in syntax for operating on points and elliptic will allow programmers to implement elliptic curve cryptographic functions with ease.

This unique and concise set of types will serve as the building blocks for any number of applications, especially those of cryptography.

4 Include the source code for an interesting program in your language

This program (really two programs, put together in the same source file for simplicity) implements the Elliptic Curve variant of the famous Diffie-Hellman protocol between two parties Alice and Bob.

```
1  /*  Public keys:   E = elliptic curve mod p, i.e. equation y^2 = x^3 + ax + b (mod p) --
       curve defined by a, b, p
2                     P = (x, y), a point on E of high order.
3
4      Private keys:  Alice's randomly chosen integer m, Bob's randomly chosen integer n
5
6      Protocol:      Alice sends the point A = mP = (P + ... + P) (m times) to Bob
7                     Bob sends the point B = nP to Alice
8
9                     Alice computes the point Q = mB, Bob computes the point Q = nA. Q is
                          their shared private key, and it
10                    will be hard for an eavesdropper Eve to calculate Q given A and B
                          without knowledge of m or n.
11
12
13                    int prime = 18218358123758917867
14                    Curve E = prime(a, b)
15                    Point P = E(x, y) */
16
```

```
17 //Alices Program
18 /*
19 Select E, P (public)
20
21 Pick m
22 Compute A = mP
23 Send A to Bob
24 (Now Alice has B)
25 Compute Q = mB
26 */
27
28 int main() {
29     int prime = 2**74207281 - 1; //some large, user generated prime --
30     mint a = (5, prime);    //Could alternatively define a, b as normal ints and then
           convert in E
31     mint b = (3, prime);
32     curve E = (a, b); //Since a and b are defined by the same prime, it need not be
           specified
33     point P = (E, x, y); //where x, y are coordinates on the elliptic curve. To be
           verified at compile time
34     printf("%P", P);    // make public
35
36     int m = rand();
37     point A = m * P; //int times a point
38     point B;
39
40     printf("%P", A);
41     scanf("%P", &B);
42
43     point Q = m * B;
44     return 0;
45 }
46
47
48
49 //Bobs Program
50 /*
51 Pick n
52 Compute B = nP
53 Send B to Alice
54 (Now Bob has A)
55 Compute Q = nA
56 */
57
58 int main() {
59     point P;
60     scanf("%P", &P);
61
62     int n = rand();
63     point B = n * P;
64     point A;
65
66     printf("%P", B);
67     scanf("%P", &A);
68
69     point Q = n * A;
70     return 0;
71 }
```