PRIME

a programming language designed for cryptography

Alexander Liebeskind (al3853), Nikhil Mehta (nm3077), Pedro B T Santos (pb2751), Thomas Tran (tkt2120)

General Language Overview (describe the language that you plan to implement):

We plan to implement a language with syntax stylistically similar to components of C++ and Java, as demonstrated below. Many current cryptographic algorithms are implemented in this style, so this implementation would facilitate user experience and transitioning from other coding languages. Our language will be static scoped and weakly typed.

The main purpose of our language is to expedite cryptographic algorithms and modular arithmetic with a particular focus on the use of the mathematical structures of fields, rings, and groups (sets of elements with unique operations). These data types and their associated operations are fundamental components of cryptography but are often not optimized in typical programming languages. We aim to remedy this through built-in language types to allow more straightforward implementation of cryptographic/security applications without the need for extensive mathematical knowledge. However, if the user does have more mathematical knowledge, we intend to allow for operator overloading.

Proposed Functionality (explain what sorts of programs are meant to be written in your language):

The basis for all modern cryptography is mathematical trapdoor functions. That is, mathematical operations that are easy to do in one direction but very difficult to do in the backwards (inverse) direction. Modular arithmetic using large primes is at the heart of many trapdoor functions used in industry today so our language will focus on making this kind of arithmetic easy to implement.

Given that our language is targeted towards prevalent cryptography algorithms, we aim to optimize the implementation of hash functions, asymmetric and symmetric key encryption and decryption, and digital signature procedures. More specifically, we are planning to optimize PRIME for the following algorithms and general topics:

- Key exchange protocols
    - Diffie Hellman Key Exchange (classical and elliptical)
- Cryptosystems
    - RSA, El Gamal, etc.
- Signature Schemes and authentication methods
    - RSA signature, El Gamal digital signature, Remote coin flipping
- Hash functions
- Binary Linear Codes (Probably not as this is a lot of matrix mult)
    - Parity check codes, Cyclic Codes
- Exploits
    - Pollard's P-1 attack, Pohlig Hellman Attack
- Cryptography suited mathematical operations
    - Modular arithmetic, polynomial arithmetic, solving systems of equations mod n

These topics center around the use of modular arithmetic and mathematical groups, and function on the computational complexity (time demand) of factoring large products of primes. Programs written in prime will therefore be used to implement encryption and decryption, as well as many of the commonly used attacks, in order to allow users to create, implement, and test their cryptography algorithms in an optimized environment. We also anticipate the language to aid in the implementation of other more complex algorithms similar to those

mentioned above (due to similarities in underlying mathematical properties), in addition to further uses of modular arithmetic and large number processing.

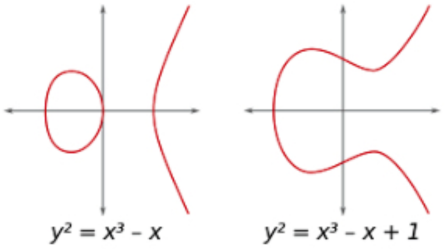
Language Components (explain the parts of your language and what they do):

Our proposed language aims to promote the implementation of cryptography algorithms that all draw on modular arithmetic as the basis for a mathematical ring, through which two standard operators (+, *) are defined. These operators serve as the building blocks for more complex computations that may involve modular exponentiation or taking the multiplicative inverse with respect to a prime number.

As referenced above, large prime numbers lie at the heart of modern cryptographic systems so we plan to design our language around and eventually include a big number library (GNU) to ease the process of writing code and carrying out computations on large numbers.

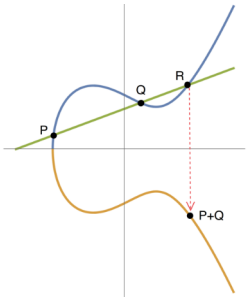
Our language will support modular arithmetic dealing with integers, large integers, polynomials, and n-dimensional points. These types serve as the building blocks for modern cryptographic methods and computations. Users will define a *ring* to perform operations on a specific type with respect to specific modulus. The declaration of a *ring* will come with a set of inbuilt operators for standard cases- integer arithmetic modulo a large integer, polynomial arithmetic modulo a prime polynomial, etc. However, just as how mathematical rings can take elements and perform operations of any kind, our language will allow the user to choose any combination of types or make their own to perform operations on with respect to a modulus of any type. Every operator can be overloaded to the user's specifications. The result of this flexibility means an elliptic curve mod p with distinctive operators can be easily coded up from integer and polynomial rings.

An elliptic curve (mod p) is defined as a curve of the form $y^2 \equiv ax^3 + bx + c \pmod p$ where points on the curve are (x, y). Elliptic curves are an extremely recent phenomena in the context of modern cryptographic technology due their conduciveness for building trapdoor functions.



Two examples of elliptic curves

In elliptic curve cryptography, an operation + on two points p, q is defined as the third point r where the line generated by p and q intersects the elliptic curve reflected about the x axis.



Our language is written in such a way that while elliptic curves are not a ready-to-use type, they can be coded up along with the + operator in less than 10 lines of code. This speaks to our language's design as a set of building blocks that can be combined to build powerful and relevant applications.

In addition to standard operations offered in C++ (i.e. +, -, *, /, etc) which are not fully enumerated here for the sake of brevity, our language will offer the following operators and types to accommodate cryptography algorithms. Operators will be distinguished based on input variable types.

Operators:
+: Arithmetic Modular Addition
-: Arithmetic Modular Subtraction or Additive Inverse
*: Arithmetic Multiplication
/: Arithmetic Division
^: Arithmetic Exponentiation
%: Modulus
`: Multiplicative Inverse

Types:
int: A signed 4-byte integer, with values ranging from -2,147,483,648 to 2,147,483,647, closed under five basic operators.
lint: A very large integer (approx. 200 digits, which is state of the art), based on a large number library (GNU C Library). The primary purpose of the *lint* type is to represent large prime numbers. The products of large prime numbers are computationally expensive to compute, and thus are the basis of cryptography functions.

lint primeOne = "67280421310721"

pt: A defined coordinate, typically defined within a ring to perform algebraic operations with respect to the ring.

pt generator[2] = (1,5)
generator[0] = 3

ring: Set of elements with their own operations addition and multiplication, that satisfy the following axioms:

(1) The set R is a group for addition. The unique identity element for
the addition operation is denoted by 0.
(2) The addition operation is commutative.
(3) The multiplication operation is associative. There exists an identity
element denoted by 1. We assume that 1 6= 0.
(4) The two operations are related by the distributive law: a · (b + c) = a · b + a · c

ring tinyprime = (int, 11)
tinyprime.comp(4+10)

The comp() function computes the arithmetic statement within the ring following the rules of PEMDAS.

poly quadratic = <1x^2 + 3x + 5, 7>
ring polymodRing = (12, quadratic)

poly: Polynomial defined with a modulus. Determining the roots of a polynomial function with respect to a modulus is often used within cryptography.

poly = <1x^2 + 3x + 5, 7>

for, while loop: For loops and while loops will work with similar syntax to that of Java or C

Sources:
Numbers, Groups and Cryptography - Gordan Savin
Introduction to Cryptography with Coding Theory - Wade Trape, Lawrence C. Washington
An Introduction to Mathematical Cryptography - Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman

Sample Code (include the source code for an interesting program in your language):
(note that syntax specifications are subject to update during implementation of the language)

Diffie Hellman Key Exchange (between communicators Alice and Bob) A primitive root and modulo a large prime are agreed upon beforehand. Alice and Bob choose secret integers randomly.

Alice's part

```
int diffiehellman(int primroot, lint bigprime){
	ring difhel = (lint, bigprime);
	//generate random number
	int alice_secret = int.rand();
	lint alice_pub = difhel.comp(primroot^alice_secret);
	print(alice_pub);
	//alice scans bob's public integer
	lint bob_pub = scan();
	lint secret = difhel.comp(bob_pub^alice_secret);
}
```

Bob's part

```
int diffiehellman(int primroot, lint bigprime){
	ring difhel = (lint, bigprime);
	//generate random number
	int bob_secret = int.rand;
	lint bob_pub = difhel.comp(primroot^bob_pub);
	//bob prints his public key
	print(bob_pub);
	//bob scans alice's public integer
	lint bob_pub = scan();
	lint secret = difhel.comp(alice_pub^bob_secret);
}
```

Elliptic Curve Creation

```
void main(){
	poly primepoly = <1x^3 + 3x + 5, 7>;
	// modulo polynomial defined with a base integer modulus
	ring curve = (pt[2], primepoly);
	// elliptic curves are of the form y^2 = ax^3 + bx + c, mod p
	// so we can define our curve as a ring that takes 2d points
	// modulo a polynomial with base p

	//overload the + operator for the curve ring
	curve:operator+(pt rhs){
		//define base ring with polynomial's base modulus for integer //modular
		arithmetic
		ring base(lint, curve.modulo.modulo);
		//calculate slope
		lint m = base.comp((rhs[1]-this[1])*(rhs[0] - this[0])');
		lint sum_x = base.comp(m^2 - this[0] - rhs[0]);
		pt sum = (sum_x, base.comp(m*(this[0] - sum_x) - this[1]));
		return sum;
	}
	// now we can sum two points on an elliptic curve to get a third
	pt p = (1, 5);
	pt q = (2, 3);
	pt r = curve.comp(pt1+pt2);
}
```