Margaret Mallernee	mlm2299
Zachary Silber	zs2266
Michael Tong	mct2159
Richard Zhang	rz2345
Joshua Zweig	jmz2135

Programming Languages and Translators, Spring 2017

C% Language Reference Manual

Contents

1 Introduction

C% is a C like language purposed for applications of cryptography.The issue in the current state of the art cryptography is not in the theory of Diffie Hellman, but rather in the cryptology of how the protocol is implemented in specific applications and contexts. With this in mind, C% provides built in types, operators and functionality that are designed to remove the burden of carrying around large primes and keeping care over many modular calculations from the programmer.

The fore mentioned functionality is built into a familiar and powerful C like language specified in this manual. Subsequently, in the specification of this language and development of this manual many ideas and notions are purposed from the C Reference Manual [1] as well as Kernighan and Ritchie's *The C Programming Language*[2].

2 Types

Each data type will be given a name and stored in a variable. These names are case-sensitive and made up of alphanumeric characters (Including '_'). Certain keywords used elsewhere in the language will be reserved and unable to be used as variable names ("int", "if", "for"). (Cite K&R)

2.1 Basic Data Types

The basic data types listed will be implemented similarly to C.

2.1.1 `int`

An Int is a 4 byte representation of an integer. An Int ranges in value from -2,147,483,648 to 2,147,483,647. When an Int overflows in either direction behavior is not defined.

Ints can be declared with or without an initial value. If no initial value is given it will have an undefined value until something is defined.

```
int x;
int x = 5;
```

2.1.2 `char`

A Char is a 1 byte representation of any ASCII character.

Chars are declared by giving a charcter surrounded by apostrophes. If no initial value is given it will have an undefined value until something is defined.

```
char x1 = 'c';
char x2 = '\n';
```

Note that our language has string literals which will allow character arrays to be used similarly to Strings.

2.1.3 `void`

Void is reserved as a type to allow functions to return nothing.

2.2 Cryptographic types

These cryptographic types represent algebraic structures that are useful for the types of programs C% is intended for. These types of programs often deal with number much larger than a reasonably sized Integer type could handle. Thus, the building block of all of the cryptographic types is a Stone, which will be implemented similarly to BigNum. All following types are declared with stones in order to facilitate calculations on unbounded numbers.

[1]https://www.bell-labs.com/usr/dmr/www/cman.pdf

[2]Kernighan, Brian W., and Dennis M. Ritchie. The C Programming Language. Vol. 2. Englewood Cliffs: Prentice-Hall, 1988.

2.2.1 `stone`

A Stone is the building block of most of the other cryptographic types. Stones are represented as a linked list of integers and grow in size every time they run out of space to hold the current number. Thus, a Stone is effectively an unbounded integer. Stone's are used to represent the large numbers and primes that our programs are intended to work with.

A Stone is defined as simply as an Integer and many of the same operators will apply to it.

```
stone x = 5;
stone x = 999999999999999;
```

2.2.2 `mint`

A ModInt is a data type that holds two things; a current value and an immutable modulus. ModInt's are used to represent a number under a certain modulus to be used for all of its calculations. Because it only makes sense to perform operations on ModInt's with the same modulus at any given time, behavior is undefined if a user attempts to apply binary operations on two ModInt's with differing moduli.

A ModInt will only be able to be defined with two Stones to enable to work with unbounded integers. The value will be listed before the modulus in the declaration.

```
stone s1 = 3;
stone s2 = 7;
mint x = {s1, s2};
```

2.2.3 `curve`

Curves represent elliptic curves. Mathematically, the data of an elliptic curve is just its two coefficients a and b (see Appendix). Thus, Curves are declared with two ModInts. There is not any defined behavior if the two ModInt's have different moduli.

```
mint x1 = {5, 7};
mint x2 = {3, 7};
curve c = {x1, x2};
```

Note that the numbers in the ModInt declarations will be interpreted as Stones and not Ints

2.2.4 `point`

Finally, Points are a fundamental data type for our language. Operations on Points are defined through a given curve. Mathematically, the data of a point consists of its two coordinates and the curve that it is on. In our language the two coordinates are represented with Stone's and the Curve will be given in the declaration as well. Note that operations will only have a defined behavior on two Point's under the same Curve.

```
point p = {c, 5, 6};
point pInf = {c, ~};
```

It is certainly possible that a point to may take on the value of "infinity" (see Appendix). In this case the user can define a point with the $\sim$ keyword for infinity.

2.2.5 `access()`

Many cryptographic types are made up of multiple stored values. The access function gives the user a method to read the individual values. For example, access of a ModInt would return an array with both Stones. Similarly, access on a Curve would break it down all the way to Stones and return a Stone array of length 4 (Curve is 2 ModInts which are 2 Stones each). Access of a Point, following the same logic, would return a Stone array of length 6. The Stones are in the same order as the type declarations.

```
mint m = {5, 7};
stone modulus = access(m)[1];
```

2.3 Grouping

Pointers and Arrays will both generally be handled as they are in C.

2.3.1 `pointer`

Pointers are stored with 8 bytes and represent a specific address in memory. Pointers can be incremented or decremented to look at adjacent blocks of memory. Pointers will specifically be useful in referencing Stone's which can be arbitrarily large.

Pointers are declared by naming the type of data to be pointed to and using the dereferencing (*) and referencing (&) unary operators.

```
int x = 7;
int* ptr = &x;

stone s = 17
stone* sPtr = &s;
```

2.3.2 `array`

Arrays can be used to store multiple instances of the same data type. An Array of any type can be declared with any specific size. Use square brackets to declare.

```
int[] arr[10];
arr[0] = 2;
arr[1] = 4;
```

3 Lexical Conventions

3.0.1 Keywords

The following keywords are reserved in the language. They are enumerated here by type and are each discusses in later sections.

- Types
  - `int`
  - `char`
  - `stone`
  - `void`
  - `mint`
  - `curve`
  - `point`
- Statements
  - `if`
  - `else`
  - `while`
  - `for`
  - `do`
  - `break`
  - `continue`
  - `return`
- Miscellaneous
  - `NULL`
  - ~

3.0.2 Comments

Single line comments are denoted by //. Any text on a line after a // will not be processed as part of the program.
Multiline comments are opened with `"(:"` and closed with `":)"`.

4 Expressions

4.1 Primary Expressions

Primary expressions are in many cases the fundamental expression type that is operated on. This section enumerates the class of primary expressions.

4.1.1 Identifiers

Identifiers are typed expressions that have been declared. For example in

```
int a = 5;
int *b = &a;
stone *c = {7, 12, a};
```

a, pointer to *b* and pointer to *c* are identifiers with types int, int pointer and stone pointer respectively. The array defined by *c* is specified by a pointer to it's first element. Additional identifiers include functions. In this case, the expression's type is that which the function returns.

4.1.2 Literals

Literals come in the form of numeric constants or string literals. If a constant fits into the size of an int, it will be of type int. If not, it will automatically be of type stone, as will be the case if a large prime is hard-coded (not recommended).

String literals will be expressed by any characters set between two double quotes. They will be of type `* char`.

All literals are strictly rvalues.

4.1.3 `NULL` and ∼

`NULL` is a reserved keyword and is a pointer to the memory address #0x0.

Additionally, ∼ is how infinity will be expressed in the language. It being mathematically valid for a point to have value of infinity per section 2.2.4, a point can be defined by `point p = {c, ∼}` where c is a Curve.

4.1.4 (*expression*)

A parenthesized expression. The expressions type and value is exactly that of the expression bound by the parentheses.

4.1.5 lvalue Expressions

There are exactly 3 forms of lvalues in this language. They are

1. `identifier` (non function pointers)
2. `( lvalue )`
3. `*expression`

All other expressions in this language are rvalues.

4.1.6 $primary - expression$ (*comma separated expression arguments*)

This primary expression takes the form of a function call, with the specified $primary - expression$ being a function as specified in 4.1.1. Functions can but do not need to accept a sequence of expressions, where each expressions type matches that specified in the function declaration. Each parameter/expression is to be separated from the previous by a "," (comma).

The function being executed receives the address of each parameter specified, making C% a *copy by reference* language.

4.1.7 Other Expression Types

Other expression types are built upon the Primary Expression types specified in the previous section. The remaining expression types are operations over the general expression types and are specified in section 5.

4.2 Order of Evaluation

There is exactly one operator on primary expressions, (). This operator has the highest priority in the language and groups left to right. For example, in

```
f(g(x, y))
```

function `f` receives the result of the evaluation of `g(x, y)` as its parameter.
The unary operators have the next highest precedence in the language and group right to left. Following, binary operators have the next highest priority, grouping left to right. The unary and binary operators have precedence following the order that they are presented in section 5, with the first operator presented having the highest precedence. The assignment operators have the next highest level of precedence, with equal precedence amongst all operators.

5 Operators

5.1 Unary operators

These operators work exactly as they do in C.

5.1.1 * *expression*

The unary $*$ operator represents dereference: *expression* must be a pointer, and this returns the object to which the expression points. Indeed, if the expression is a pointer to a type T, the type of the result will be T.

5.1.2 & $lvalue - expression$

The result of the & operator is a pointer to the object referred to by the lvalue-expression. If the type of $lvalue - expression$ is T, the type of the result of this operation is pointer to T.

5.1.3 - *expression*

The result of the unary $-$ operator is the negative of the expression. If *expression* is an integer type, this is the typical additive inverse. If *expression* is of type point, this returns the inverse of the point on the curve (as defined in **9.3.1**).

5.1.4 ! *expression*

The unary ! operator is the logical negation, i.e. it takes an integer type (or a pointer type) and returns (as an int) 0 if *expression* is non-zero and 1 otherwise.

5.2 Exponential operator

5.2.1 *expression* ** *expression*

The $**$ operator represents exponentiation and requires an integer type on both sides. Let the numerical value of the right expression be n (if it is of type modint, it returns the value, not the modulus). If n is non-negative, this finds the result of its value taken to the nth power (and takes the remainder if the left expression is a modint). Also, in the case that the left expression is a modint, this uses the squaring method for modular exponentiation to perform the operation efficiently.

If n is negative, this is only defined when the left expression is a modint with a multiplicative inverse, in which case this finds the inverse of the integer and takes that to the nth power (using the same algorithm as above).

5.3 Multiplicative operators

The multiplicative operators $*, /$, and % group left-to-right.

5.3.1 *expression* * *expression*

Depending on the type of the expressions, this operator has very different uses.

If the expressions are both of type modint, this returns the product of the two numbers under that modulus. The behavior is defined only when the two modints have the same modulus. If only one expression has type modint, it returns the product of two numbers under that modulus. Otherwise, it returns the typical numerical product of the two.

Now, if one expression has type point, then the other must be an integer type. In this case, if the integer type has value n, it returns the result of the point added to itself n times (see §9 for further explanation). No other type combinations are allowed.

5.3.2 *expression* / *expression*

The / operator only works when the expressions are of type int or stone. It implements integer division, returning the type of the first expression.

5.3.3 *expression* % *expression*

The % operator only works when the expressions are of type int or stone, and returns the remainder of the first expression when divided by the second, returning the type of the first expression.

5.4 Additive operators

The additive operators $+$ and $-$ group left-to-right.

5.4.1 *expression* $+$ *expression*

The $+$ operator returns the sum of the expressions. If the expressions are both integer or modint types, conditions are analogous to those applying to $*$.

If the expressions are both of type point, this computes the elliptic-curve addition of the two points (see §9), the behavior of which is only defined when the two points are relative to the same elliptic curve. No other type combinations are allowed.

5.4.2 *expression* - *expression*

The binary $-$ operator returns the difference of the expressions. It calculates exactly *expression* $+$ ($-$*expression*) and is valid for all types for which this expression is valid.

5.5 Relational operators

5.5.1 *expression* $<$ *expression*

5.5.2 *expression* $>$ *expression*

5.5.3 *expression* $>=$ *expression*

5.5.4 *expression* $<=$ *expression*

These expressions return 0 or 1 based on whether the specific relation is false or true, respectively. The operators are only defined when the expressions are of integer type or are a pointer, though both expressions need not be of the same type. In the case of a pointer, it uses the memory location that it is pointing to (interpreted as a number) to compare, and in the case of a modint it uses its numerical value (without the modulus).

Note that the statement $a < b < c$ does not seem to mean what it does, even though the operators do group left-to-right.

5.6 Equality operators

5.6.1 *expression* == *expression*

5.6.2 *expression* != *expression*

These are analogous to the relational operators, except they have lower precedence. Note that this implies that they implement structural equality, not physical equality.

Note that if the two expressions are of type modint, it does not check that the moduli are equal. To do so, call `access()` (see **2.2.5**). Also, == is defined for points – it checks that the curve they are defined over is the same and that its x and y coordinates are equal as modints.

5.6.3 *expression* && *expression*

This is the logical AND operator. It returns 1 if both values are non zero and 0 otherwise.

5.6.4 *expression* || *expression*

This is the logical OR operator. It returns 0 if both arguments are 0, 1 otherwise.

5.7 Assignment operator

5.7.1 *lvalue* = *expression*

At the end of the evaluation, the value of the right operand is stored in the left operand. The value of this operation is exactly the value of the left operand after the assignment has been completed. The operand on the left of the assignment operator must be an lvalue.

5.7.2 *lvalue* =% *expression*

At the end of the assignment, lvalue will store the result of *lvalue % expression.*

5.8 Other

5.8.1 *expression*, *expression*

An expression of this format is equal to the value of the right-most expression. This is a unique use of the "," as compared to the usages in separating parameters in a function call, as per section 4.1.6 and in type declarations, as per section 2.2.

6 Statements

Statements allow the user to control when expressions are executed in their programs.

6.1 Statement Terminator & Blocks

An expression is turned into an expression statement when it is terminated by a semicolon:

expression ;

Any expression becomes an *expression statement* when terminated by a semicolon, however expression statements are only useful for their side effects, such as a calling functions or assigning a value to a variable.

Statements may be grouped together into a *compound statement* or *block* using braces { }. Blocks are used to group multiple statements together, and the resulting block is syntactically equivalent to a single statement. Blocks are commonly seen in the definition of a function body, or with many of the control flow statements defined in this section, such as for, while, and if. Unlike normal statements, blocks do not need to be terminated by a semicolon. Blocks may be defined within other blocks, and variables may be declared within any block. When a variable is declared within a block, its scope is defined by that block, meaning that the variable is local to the block in which it is defined.

6.2 Control flow

Control flow statements determine the order in which expressions are evaluated.

6.2.1 `if`, `else`

The keywords `if` and `else` are used to create conditional statements, which selectively execute statements based on the truth of a given expression. The simplest constructions are

`if` (*expression*) *statement*$_1$

`if` (*expression*) *statement*$_1$ `else` *statement*$_2$

In each of these expressions, *statement*$_1$ is executed if (and only if) *expression* evaluates to true (a non-zero value). If *expression* `== 0`, nothing happens as a result of the `if` statement unless there is an `else` following *statement*$_1$, in which case *statement*$_2$ is executed instead of *statement*$_1$. For example,

```
if (n > 3)
        x = 1;
else
        x = 5;
```

If `n > 3` evaluates to true, then the variable `x` is assigned the value 1, otherwise `x` is assigned the value 5. Note that the expression `n > 3` is not terminated by a semicolon, as it is not a *statement*, while both `x = 1` and `x = 5` are semicolon terminated, in accordance with the expected syntax of `if` statements listed above.

`if` statements can be nested and combined to test for multiple conditions:

```
if (n > 3)
        x = 1;
else if (n == 3)
        x = 3;
else
        x = 5;
```

In nested statements such as these, each `else` matches the closest preceding `else`-less `if`. This rule clarifies the ambiguity resulting from the fact that not every `if` must have an `else`. For example, in

```
if (n > 3)
        if (n < 10)
                x = 1;
        else
                x = 5;
```

`x` will be assigned the value 5 if `n` is greater than 3 but not less than 10, i.e. the `else` is matched with the most recent `if` which tests if `n < 10`. If you want to match the `else` with the first `if` instead, you must use braces:

```
if (n > 3) {
        if (n < 10)
                x = 1;
}
else
        x = 5;
```

6.2.2 Loops – `while`

The `while` statement is used to create a loop that runs so long as a given expression is true. The formal syntax is:

> `while` (*expression*)
>   *statement*

In a `while` statement, *expression* is evaluated first. If it evaluates to a non-zero value, *statement* is executed and then *expression* is re-evaluated. This cycle continues until *expression* evaluates to 0, or is false, at which point the `while` statement is over, and *statement* will not be executed anymore. A `while` statement may also be exited using a `break` statement, as discussed in **5.2.5**.

The following example computes the sum of the first 9 integers:

```
int sum = 0;
int n = 1;
while(n < 10) {
        sum = sum + n;
        n = n + 1;
}
```

6.2.3 Loops – `for`

The `for` statement has the syntax

> `for` ($expr_1$; $expr_2$; $expr_3$)
>   *statement*

and is equivalent to a `while` statement of the form

> $expr_1$;
> `while` ($expr_2$) {
>   *statement*
> $expr_3$;
> }

except only in relation to `continue`, discussed in **5.2.6**.

The three elements of a `for` loop are any expressions, and any or all of these expressions may be omitted. Specifically, $expr_1$ is executed once at the beginning of the loop, as an initialization step; $expr_2$ is a test expression that is executed before each iteration of the loop, such that if $expr_2$ evaluates to false the loop exits; and $expr_3$ is called after each iteration of the loop body, as an incremental step. Omission of $expr_1$ or $expr_3$ simply means that the loop runs without calling those expressions. In the event that $expr_2$ is omitted, the loop runs as if $expr_2$ is true permanently, as in an 'infinite' loop (`while(1)`). This makes sense only if the loop is to be broken in some other way, such as with a `break` or `return` statement. It is also important to note that, in accordance with the behavior of the comma operator (see **5.9.1**), the use of a comma separated expression for $expr_2$ will have the effect of only using the rightmost expression as a check for termination of the loop. In order to check multiple conditions, you must combine the expressions using `&&`.

For example, the following loop will terminate when `x` reaches 10, not when `y` reaches 5:

```
int sum = 0;
for(int y = 0, x = 0; y < 5, x < 10; y = y + 1, x = x + 1)
        sum = sum + x + y;
```

To actually check both conditions, replace $expr_2$ with `y < 5 && x < 10`.

The `for` loop makes more sense than the `while` loop in cases where there are present and simple initialization and incremental steps, as when you want to count the number of times the loop runs.

Our previous example is one of those cases:

```
int sum = 0;
for (int n = 1; n < 10; n = n + 1)
        sum = sum + n;
```

6.2.4 Loops – `do-while`

The `do-while` statement is a loop that checks the condition for termination after running the loop body. Unlike a `while` or `for` loop, the body of the loop will always be executed at least once, regardless of the evaluation of the expression. The syntax is as follows:

```
do
   statement
while (expression) ;
```

The `do` statement first executes *statement*, and then evaluates *expression*. If *expression* evaluates to true, the cycle repeats, executing *statement* repeatedly until *expression* evaluates to false (0). A `do-while` statement may also be exited using a `break` statement, as discussed in **5.2.5**.

The following example also computes the sum of the first 9 integers:

```
int sum = 0;
int n = 1;
do  {
        sum = sum + n;
    n = n + 1;
} while(n < 10);
```

6.2.5 break

A `break` statement is used to terminate a `while`, `for`, or `do` loop statement aside from the built in termination condition. In the case of nested loops, `break` causes only the innermost loop to exit. As a trivial example, the following loop simply increments `i` until it is a multiple of 3:

```
for (int i = 1; i <= 10; i = i + 1)
        if (x % 3 == 0)
                break;
```

6.2.6 continue

A `continue` statement is like a `break` statement, except that instead of exiting the entire loop, the `continue` statement causes the loop to continue to the next iteration without running through any remaining statements in the loop body. As with `break`, `continue` only affects the innermost loop. Both `do` and `while` loops move directly to the test expression upon encountering a `continue` statement, while a `for` loop passes control to the incremental expression ($expr_3$ in **5.2.3**). For example, the following computes the sum of even numbers between 0 and 10:

```
int sum = 0;
for (int i = 0; i < 10; i = i + 1)  {
        if (x % 2 == 1)
                continue;
        else
                sum = sum + i;
}
```

6.2.7 Null

The *null statement* is a single semicolon.

;

A null statement does nothing, and has no impact on your program, but be useful as the body of a loop statement. For example, the following snippet sets `x` to the largest integer less than the square root of `a`:

```
int x;
for (x = 2; x*x < a; x = x + 1)
        ;
```

6.2.8 return

The `return` statement is used to end the execution of the surrounding function body and return control back to the calling function. The statement has the form

`return` *value* ;

where *value* is optional, and only valid if the function has a return type other than `void`. If the function has a return type other than `void`, the function return type and the type of *value* must match. In this case, it is also valid to omit *value* so long as the calling function does not require a return value. It is generally not a good idea to omit *value* unless the function return type is `void`.

7 Program Structure

A Program is composed of variables and functions. Information can be shared between functions by parameters, return values and global variables. The functions can occur in any order in the source file, so long as a function's prototype or definition is provided before or at the time of its use.

Programs require a `main()` function, and all other functions used within `main()` must be declared before it, but may be defined afterwards, like in the following example:

```
int otherFunction();
int main()
{
    declarations and statements
}
int otherFunction()
{
    declarations and statements
}
```

How functions interact with the declarations and statements of other functions will be discussed in the section 7.2.

7.1 Functions

7.1.1 Basic Functions

Functions have the following form:

```
return-type function-name ( argument declarations )
{
    declarations and statements
}
```

Functions need a `return` statement to return a value from the called function to its caller, with the exception of functions that have `void` as the return-type (refer to section 2.1.3). If a return type is not explicit, then it will be assumed to be an `int` [2].

7.1.2 Returning Expressions

Functions may return expressions as follows:

```
return expression;
```

The *expression* will be converted to the function's return type if they do not match.

7.1.3 Void Functions

Functions may return what appears to be the return-type `void` like in the following:

```
void function-name ( argument declarations )
{
    declarations and statements
}
```

In `C%`, a `void` function type indicates that the function does not return a value.

7.2 Scope

7.2.1 Lexical Scope

The lexical scope of names declared in external definitions (in a file, but not within a function or other specified scope) extends from their definitions to the end of their respective file. Within functions, the lexical scope of names is the body of the function itself [2].

```
main() { int a; ...}
//a falls out of scope

int mainCantSeeThis = 0;

void mainCantSeeThisEither(int a) {...}
```

In the above example, `main()` cannot access the int declared after it, nor can it access the `void mainCantSeeThisEither`. In the same example, `void mainCantSeeThisEither` can access the declared `int mainCantSeeThis`. Furthermore, a is not accessible outside of the scope of its definition.
It is illegal to redeclare a name within its context unless it is of the same type and class it previously held [2].

8 File I/O

8.1 I/O Channels

`C%` supports three I/O channels: standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`). They are represented by the `int`s that describe their system file descriptors in `C%`'s standard I/O functions. `stdin` is represented by `0`, `stdout` is represented by `1`, and `stderr` is represented by `2`.

8.2 `write()`

`write` directly outputs a program's data by system call:

```
int  write(  int  channel,  char  *buffer,  int  nbyte );
```

The write() function attempts to write nbytes from buffer to the channel associated with the file descriptor passed as a parameter.
The function returns the number of bytes written to the file. A return value of -1 indicates an error.

8.3 `read()`

`read` directly reads from the `stdin` by system call:

```
int read( char *buffer, int nbyte );
```

The read() function attempts to read nbytes from `stdin`, and places the characters read into `buffer`. It removes carriage returns.
The function returns the number of bytes read. On carriage return, 0 is returned and on error it returns -1.

9 Appendix on cryptography

9.1 Modular arithmetic

Most readers are likely familiar with modular arithmetic. It is the arithmetic of integers where only the remainder of the number when divided by a fixed modulus is considered, so for example one would take $3 * 5 = 15$ and take the remainder when divided by the modulus 7, which is 1.

Modular arithmetic lies at the heart of cryptography, which concerns itself with sending information securely such that it cannot be read by any ordinary person unless they have access to some other private information (typically referred to as a key). Indeed, the security of modern cryptography protocols relies on various properties (or at least properties which are widely believed, but yet unproven) of the multiplication of integers when taken with a large (usually prime) modulus. For example, many security protocols rely on the *hardness of the discrete log problem*: given a prime modulus p and two integers α and β taken mod p, find an integer k for which $\alpha^k \equiv \beta \pmod{p}$.

9.2 General setting

Much of cryptography has historically been studied and implemented within this framework of modular arithmetic. However, from a more abstract point of view, the core of what we are working with here is a set of things (in this case, non-negative 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 (there is some element e so that for every a, we have $e \cdot a = a \cdot e = a$), and it can be inverted (for every a, there is some other element b so that $a \cdot b = b \cdot a = e$). 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. This ties nicely into our implementation of elliptic curves.

9.3 Elliptic curves

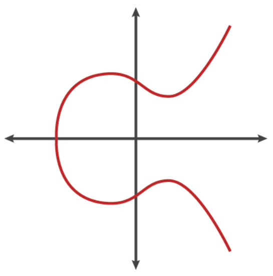
Our language will offer tremendous support for a newer brand of cryptography where the group in question consists of points lying on an object called an elliptic curve. How this works is highly technical, and we aim to explain the intricacies here.

9.3.1 Background and definitions

A typical elliptic curve is an equation

$$y^2 = x^3 + ax + b$$

where the variables in this equation take values in the real numbers $\mathbb{R}$. So one can visualize this as a set of points (x, y) in the plane for which this equation is true. Here is an example:



The curve $y^2 = x^3 - x + 1$

The elliptic curve has the following remarkable property: if a line passes through two points on the curve, then it passes through three points. Well, not exactly. The points of intersection are counted *with multiplicity*, which practically means that if a line is tangent to a curve at a point, then this intersection counts as two. Also, one might notice that in the picture, a vertical line only intersects the curve in two points. Vertical lines are said to intersect the curve *at infinity* – this sounds like cheating, but there is a (rather technical) way to have this make perfect sense (and our implementation deals with this behind the scenes). Furthermore, notice that if a point $P = (x, y)$ is on an elliptic curve then $(x, -y)$ is also on it, and we call this point $-P$.

Using these two facts, we have the following binary operation, called addition and denoted $+$: it takes two points P and Q as input. It finds the unique line passing through both points (if the points are the same, it takes the tangent line) and finds the third point on the line R. The operation then returns $-R$. In short, $P + Q = -R$.

It takes some work to check, but it turns out that this addition $+$ satisfies the properties stated above in §9.2 so that the points on an elliptic curve form a group.

9.3.2 Addition formula

The following formula describes the addition law mentioned above. Let E be an elliptic curve given by $y^2 = x^3 + ax + b$. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be points on E. The point $P_1 + P_2 = P_3 = (x_3, y_3)$ is defined as follows:

$$\begin{aligned} x_3 &= m^2 - x_1 - x_2 \\ y_3 &= m(x_1 - x_3) - y_1 \end{aligned}$$

where

$$m = \begin{cases} (y_2 - y_1)/(x_2 - x_1) & \text{if } P_1 \neq P_2 \\ (3x_1^2 + a)/(2y_1) & \textit{if } P_1 = P_2. \end{cases}$$

and if the denominator of the relevant m is zero, then the sum is the point infinity and we write $P_3 = \infty$.

9.3.3 Translation to cryptography

Now, in the above exposition we took elliptic curves defined over the real numbers, but it turns out that this can be defined over any algebraic object known as a *field* (simplified, this is a set with *two* binary operations $+$ and $\times$ where both operations enjoy the nice properties of a group and both operations distribute over each other, as they do in $\mathbb{R}$).

Furthermore, the set of integers when taken with a prime modulus p, equipped with the typical modular operations $+$ and $\times$, is a field and is typically denoted $\mathbb{F}_p$. Typically, elliptic curves in cryptography are defined over this object $\mathbb{F}_p$ instead of $\mathbb{R}$, so that a point a curve E is really an equation

$$y^2 \equiv x^3 + ax + b \pmod p$$

and the points on it are pairs of (modular) integers (x, y) for which this equation holds. Indeed, even the addition law is the same, except the formula for m "changes" (we put "changes" in quotation marks since this is really just a generalization of the above) to

$$m = \begin{cases} (y_2 - y_1)(x_2 - x_1)^{-1} & \text{if } P_1 \neq P_2 \\ (3x_1^2 + a)(2y_1)^{-1} & \textit{if } P_1 = P_2. \end{cases}$$

where x^{-1} denotes the *multiplicative inverse* of x in whatever field it is defined in. For example, the multiplicative inverse of 5 with respect to the modulus 7 is 3, since $5 \cdot 3 \equiv 1 \pmod 7$, so we write $5^{-1} \equiv 3$. As above, the sum of two points is ∞ if the "denominator", i.e. the term being inverted, of the relevant m is equal to zero.

9.3.4 Comparison with modular arithmetic

The discussion in §9.2 motivates the analogy between the two cryptosystems. Indeed, while in modular arithmetic we use integers mod p as building blocks and modular multiplication as an operation relating these elements, in elliptic curve cryptography we use the points on an elliptic curve as the elements and use the aforementioned addition law to relate these elements together. With this analogy, one can translate many cryptographic protocols over the modular integers into an exact protocol over an elliptic curve.

Indeed, recall the discrete log problem from §9.1. The elliptic curve analogy is this: given a curve E and two points on it P and Q, can you find an integer k for which $kP = Q$? (Here kP denotes P added to itself k times). The analogy should be becoming clear.

References

[1] Dennis M. Ritchie *C Reference Manual.* https://www.bell-labs.com/usr/dmr/www/cman.pdf

[2] Brian W. Kernighan and Dennis M. Ritchie *The C Programming Language Volume 2.* Englewood Cliffs: Prentice-Hall, 1988.

[3] Travis Rothwell, James Youngman, et al. *GNU C Reference Manual.* Free Software Foundation, 2008.