The CEAL Book

Revision 8, draft 2 (23 Dec 2015)

Oleg Mazonka, 2015

Table of Contents

Disclaimer *I try to put down the ideas, solutions, and functionality of CEAL as quick as possible. Here I focus more on amount and completeness of information rather than proper structure and clarity of expression or good English.*

1. Preamble

Cryptoleq is a one instruction language (OISC). An enhanced assembly language designed for writing programs for Cryptoleq is **CEAL** (Cryptoleq Enhanced Assembly Language). The CEAL compiler is a program translating CEAL program into Cryptoleq code. The CEAL compiler, as a current program, has also built-in emulator and is able to emulate Cryptoleq. It is written in standard C++ with using elements of C++14.

Cryptoleq is an abstract machine consisting of one instruction. Cryptoleq is an OISC. The closest relative is Subleq.

This document should be considered as an addition to the first description of Cryptoleq [1].

Section 6 of this document a simple step by step introduction into CEAL. The rest of the document can be used as a reference material.

2. Elements of the CEAL compiler

CEAL is Cryptoleq Enhanced Assembly Language. CEAL compiler is a program for translation CEAL program into Cryptoleq code, which is a sequence of numbers for Cryptoleq execution. The numbers represent memory cells with their values and optionally addresses. The compiler works by creating a *tree* from the input program. The tree is a node tree representing an intermediate state of the data. The tree is processed and reorganised by Evaluator, and at the end the tree is converted to a stream of numbers ready to by run on Emulator. The CEAL compiler modules like Evaluator and Emulator are described below.

2.1 Modules

The CEAL compiler consists of the following modules:

Module	Synopsis
Ceal driver	C++ main function
Cloner	Instantiation of macros
Context object	Sequence of steps for processing
Emulator	Emulation of Cryptoleq processor
Evaluator	Reduction of the tree
Macros manager	Managing macro definitions and instantiations
Memory	Abstraction of memory
Nodes	Definition of different types of tree nodes
Parser	Conversion of token stream into a tree
Pragma object	Managing pragmas
Stat	Collecting of statistics while emulating
Tokenizer	Conversion of text into a token stream
Utilities	Collection of different functions
Compiler object	Database of encryption parameters
Processor object	Database of parameters for emulation
Memcell	Implementation of memory cell

Bignum library	Implementation of bignums and adaptor for 3rd party libraries

The **CEAL driver** is the main function of the program. The main creates Context and Pragma objects, initialises them; creates Compiler object; and then translates the program by running Tokenizer, Parser, and Evaluator. Depending on options it can proceed to emulation part without exiting.

Cloner is a simple module which clones a sub-tree from the tree. This module works during Macro instantiations.

Context object is a helper functionality to the main function. The text of the program is converted to a sequence of Tokens (a token stream), then to a tree; then the tree is converted back to the token stream, and then tokens converted to text. The job of Context object is to:

1. process command line options;
2. run Tokenizer;
3. run Parser;
4. run Evaluator;
5. compile the tree to a token stream;
6. write a stream of tokens;
7. save output to a file;
8. dump the tree to output; and
9. process crypter command line option (see 2.7).

Emulator is a module which emulates Cryptoleq processor. It can execute cryptoleq code using Processor object. This module is also used in a satellite program Emlrun[1].

Evaluator is a complex module in CEAL. Its job is to evaluate the tree by reducing high level CEAL concepts to lower level until the tree can be represented by pure Cryptoleq code. Evaluator executes the following steps in this particular order:

1. substitute (instantiate) macros;
2. move definitions to the root node
3. expand string literals and arrays;
4. complete 3-operand instructions;
5. assign addresses;
6. resolve names – build dictionary;
7. evaluate expressions; and
8. check memory overlaps.

Each step goes through and modifies Tree. The modifications can be viewed by using command line option ‘-t’ (see 2.7).

Macros manager is a module processing macro definitions and macro instantiations. It clones macro sub-tree into the appropriate point in the tree; and then resolves

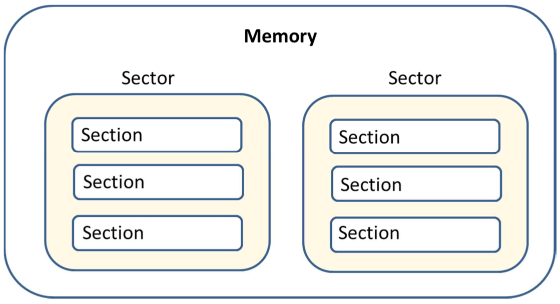
[1] Emlrun is a standalone program which can be shipped with Cryptoleq code for execution without disclosing the compilation method.

internally used names against the arguments and declared globals. It also implements Autobits, see 2.2. Macros are described in Section 7.

Memory module is responsible for organisation and access to memory cell during emulation. There are three types of memory implementation. The type is selected during build (see 2.8).

Cryptoleq value is X- or TS-number. X-number is obtained as a result of encryption. TS-number is X number broken into T and S parts. See more detail in Section 3.

> **Type 1**. The memory is organised as a collection of Sectors. Each Sector is a collection of Sections. Each Section is a collection of memory cells. All memory cell addresses within one Sector share the same S-value. All memory cell addresses within one Section are sequential in T-value. Incrementing T-value by unit gives the address of the next cell. Accessing a cell not belonging to any Section results in "Memory access violation" error and halts the execution.



> **Type 2**. The memory is a hash (C++ std::unordered_map) of X-values.

> **Type 3**. The memory is a red-black tree (C++ std::map) of X-values.

The default type is 3.

Nodes is a module defining different types of nodes used in the tree. Its role is descriptive and functionality is simplistic. All types of nodes are derived from class Node which has a sub-object Token. Token holds information about its type, position in the source file (for error reporting), and numeric and string values. Each node has a list of children nodes and a pointer to the parent. The following lists the classes derived from Node:

- Root – root node which does not have parent; it stores program wide data such as name definitions, macro definitions, reference to Compiler object.
- Item – an object representing memory cell
- Labels – list of labels for Item
- Instruction – list of Items
- Litem – Item and associated Labels
- Expr – arithmetic expressions
- Term – sub-expression; it can be another expression, unary minus, constant, identifier, or special function
- Idn – identifier: an alphanumeric word staring with a letter
- Cnst – constant value: TS-number, character, or question mark

- Unum – plain number
- Tsnum – TS-number
- Macuse – macro directive
- Macdef – macro definition

Parser converts token stream into the tree. It is organised as top-down recursive descend according to language Backus–Naur Form (BNF), section 9.3.

Pragma – see Subsection 2.4 below.

Stat – is a class for collecting statistics. For more detail see command line option '-d'.

Tokenizer is a module converting source text into token stream. Its function includes also reading *pragma* and *include* directives.

Utilities is a collection of useful functions.

Compiler object is a collection of data required for encryption and decryption. Compiler object also holds inside Processor object in the translation part of CEAL; the emulation part, on the other hand, has only Processor object as the emulator cannot know about encryption and decryption keys. Compiler object is a bad name for the module and is used only for historical reasons. See more at section 6.3.

Processor object is a collection of data required to run Cryptoleq program. See more at section 6.3.

Memcell is a module describing 5 different implementations of memory cell interface class. The object of class Cell can:

- be constructed from TS values, X values, or by default;
- be compared "less then" and "equal" with another Cell object;
- produce its TS value, or X value;
- increment and decrement;
- produce the reciprocal; and
- produce TS string representation.

The following Cell types are implemented:

Type	Name	Description
1	CellTs	encrypted value stored as two numbers: T and S
2	CellX	encrypted value stored as one X number
3	CellInvTS	encrypted value stored as TS number plus its inversion
4	CellInvX	encrypted value stored as X number plus its inversion
5	DoublePlug	synchronous storage of two different types for debug purposes

Default build memory type is 4.

Bignum library is an implementation of big number arithmetic. It also has an adaptor for including a 3rd party bignum library. GMP adaptor is implemented.

2.2 Autobits

Autobits ("._autobit") is a special macro defined inside the compiler. It takes three arguments: constant expression, macro name 0 and macro name 1. When the constant expression is evaluated into a number and broken down into bits, the compiler builds a list of macro instructions with macro name 0 for every 0-bit and macro name 1 for every 1-bit. The purpose for this special macro is to generate G module. See more at section 6.17.

2.3 Special functions

The following functions are constant expressions and defined inside the compiler:

Name	Description
`$unit(`*`z`*`)`	a value $(1+s)_N^{-1}$ calculated from ***z*** which can be used to increment and decrement ***z***
`$X(`*`z`*`)`	construct value given X representation of ***z*** (default is TS)
`$peekrnd()`	give the current random value without advancing it
`$random()`	give the current random value and advance it
`$B2()`	B_2
`$beta()`	β
`$fkf()`	$s(\varphi k)_N^{-1}\varphi \bmod N\varphi$, where s is a "sneak" number (see pragma sneak)
`$enc(`*`m`*`)`	Encrypt value *m*
`$halfN()`	$\lfloor N/2 \rfloor$
`$TS(`*`x`*`,`*`y`*`)`	*x.y* – both arguments should evaluate to open values
`$k()`	k
`$phi()`	φ
`$invN(`*`x`*`)`	$(x)_N^{-1}$
`$T(`*`z`*`)`	T part of ***z***
`$S(`*`z`*`)`	S part of ***z***
`$powN(`*`x`*`,`*`y`*`)`	$x^y \bmod N$

If a function does not take arguments, then the brackets are optional. Functions '**`unit`**' and '**`invN`**' are related:

```
$unit(z) ≡ $invN($S(z)+1)
$invN(z) ≡ $unit($T(z)-1)
```

2.4 Usage

CEAL compiler accepts the command line options shown in the table below.

Name	Description
-	input/output using stdin/stdout
-o file	specify output file ('stdout' for console output)
-p pragma	pragma string; overrides cryptoleq.pgm and input
-s	show compiler parameters and exit
-t nnn	show the tree at different stages (0 to 8), e.g. 0237

-r seed	random seed: either number or word 'time'
-E	preprocess only, no parse or evaluate; make output after Tokenizer’s job
-x	execute; default for .sce files
-e	translate only; default for .sct files
-a	translate and execute; default for .sca files
-I	add include path, see also 2.5
-b ascii	set space separator; default ' ' for io=ts, '\n' for io=x
-c n f	run crypter; n={x|ts}{enc|dec[_r]|ord} f={file|@num}
-d file	collect statistics, slower; file is input and output
-m [expr]	Run bignum calculator, see Section 11.2

‘-c’ crypter is a helper function to encrypt and decrypt numbers from the command line or stored in files. The prefix ‘ts’ or ‘x’ specifies the format of the values. The postfix ‘_r’ tells to print *r* part. Function ‘ord’ calculates order.

Example: `$ ceal -p "PQ=7.11 r=17 k=5" -c tsdec @1.71`
Output: `16`.

Example: `$ ceal -p "PQ=7.11" -c xord @1814`
Output: `2`.

‘-d’ options initiates collection of statistics. File is both input (may exist or not) and output; input is optional – if it does not exists, it will be created. The format of the file is the following:

```
IP watch list { _G_start _omul_start _seq_start _smul_start }

IP pass counters
================
_G_start[2656604]   = 612
_omul_start[1704679]= 0
_seq_start[4938295] = 6
_smul_start[3664180]= 6

Instruction stat
================
Input/Output   = 8
Open           = 108342
Secure         = 55691
Mixed          = 35929
----------------
Total          = 199970
```

IP watch list is a list of labels (if alphanumeric) or addresses (if numeric) which are being passed by IP. From the example above (PIR 204) we can see that G function was called 612 times; open multiplication has never been executed; secure equal operation and secure multiplication was executed 6 times each (6 entries in the Database). Square brackets show the X-value address of the corresponding label. The addresses have to be specified explicitly in the *IP watch list* if emulation is run separately without compilation. The following lines in the example show the instruction count.
Option ‘-m [expr]’ runs bignum calculator tool described in section 11.2. Argument ‘expr’ is optional. If it is not specified, then interactive mode is initiated. If ‘expr’ is

one continuous string, it is assumed to be name of input file with bignum calculator commands. If ‘expr’ is several separated strings, then it is assumed to be one command.

Example: `$ ceal -m print factors 12345`
Output: `3,5,823`.

2.5 Include

Include directive has the format: *.include func “file”*. Function *func* is optional and can be one of “asis” (default) or “datax”. The later converts values from X representation.

A file is searched first in the current directory. If it is not found, then the list of include directories (specified by command line option ‘-I’ and pragma option ‘incdir’) is used in the order as directories added to the list. If file is not found in any directory, error is issued. When the file is found, the search ends.

A special directive is *.pragma once*, which instructs the include algorithm to include the current file only once in the program. For example, if A includes B and C, and B includes C, and file C has “pragma once” directive, then file C will be included only the first time.

2.6 Pragma

Pragma is a set of environment variables for Compiler and Processor objects. It can be specified in three places:

1. ‘cryptoleq.pgm’ file
2. Pragma directive in the program
3. Command line option

Each pragma specification is overridden in the above order – so a variable set in the command line overrides the one set in the program or pragma file. Pragma parameters are defined in format *name*=*value*.

List of pragma parameters

Name	Description
N	Value N (X.Y also allowed then used as an alias of PQ)
P	First prime of N
Q	Second prime of N
u	Minimal beta value the program requires. This does not affect the program except that the check may fail if N is selected so beta is not big enough.
k	k value
r	Seed for the random engine
entry	Entry address, default is 0
beta	See section 3
PQ	Same as P and Q but in format X.Y
io	One of (ascii|a|ts|x). Format for input/output

cqtype	One of (ts|x). Format of cryptoleq code
id	Informational parameter – name
ver	Informational parameter – version
incdir	Same as the command line option –I
sneak	“sneak” number

Sneak number is a number which adds extra security to “fkf” function (see built-in functions). This number is by default 1. When beta parameter specified explicitly and is it is less than maximal (default) beta, then any number can be multiplied by some coefficient *s* without affecting Leq test. This coefficient must be less than $2^{\beta_{\max}-\beta}$.

2.7 X-optimisation

The purpose for this optimisation is to eliminate division operation when memory cell is represented as X value. There are two places where division is required for X representations: 1) Leq testing; and 2) advance in the address.

The former case can be restructured as following. Suppose that $t^{+}_{\max}$ is the maximal T-value which tests positive, so $t^{+}_{\max}+1$ tests negative, i.e.

$$\begin{aligned}&\mathrm{Leq}\left(X\left(t^{+}_{\max}+1\right)\right)\Rightarrow \mathrm{True}\\&\mathrm{Leq}\left(X\left(t^{+}_{\max}\right)\right)\Rightarrow \mathrm{False}\end{aligned}$$

Correspondingly $t^{+}_{\min}=1$

$$\begin{aligned}&\mathrm{Leq}\left(X\left(t^{+}_{\min}-1\right)\right)\Rightarrow \mathrm{True}\\&\mathrm{Leq}\left(X\left(t^{+}_{\min}\right)\right)\Rightarrow \mathrm{False}\end{aligned}$$

Hence, their equivalent values in X are

$$\begin{aligned}X^{+}_{\min}&=1+Nt^{+}_{\min}+(s=0)=1+N\\X^{+}_{\max}&=1+Nt^{+}_{\max}+(s=N-1)=N\left(t^{+}_{\max}+1\right)\end{aligned}$$

Testing Leq becomes as simple as testing whether X-value lays between $X^{+}_{\min}$ and $X^{+}_{\max}$:

$$\mathrm{Leq}(X)=\begin{cases}\mathrm{True}, & \text{if}\quad X^{+}_{\min}\le X\le X^{+}_{\max}\\ \mathrm{False}, & \text{otherwise}\end{cases}$$

In the latter case to avoid calculation of T part when advancing the address one can use an associated map instead of continuous array in memory. Its efficiency depends on particular algorithm and implementation. CEAL uses two such memory types 2 and 3 (see 2.1, Memory).

2.8 Build

To build CEAL use the command *make*. Arguments are presented in the table below:

Name	Description
CELLDEF=n	defines the type of memory cell to build in (see 2.1, Memcell)
PLAT=msc	use: MS cl C++ compiler, Windows
PLAT=unx	use: GCC g++ C++ compiler, Windows and Unix
MEMDEF=n	defines the type of memory (see 2.1, Memory)
GMP=1	See 2.9

2.9 Building with GMP bignum library

CEAL comes with its own portable bignum library set by default to 4096-bit precision arithmetic. This built-in library is portable and simple, at the same time may not be efficient. To get more efficiency, especially working with larger N's, one can connect defacto standard bignum library GMP. There are two downsides:

1) A dependency on a 3rd party tool; and
2) GMP can be used only in Unix-like systems (Cygwin with GCC also works)

To build CEAL with GMP. Do the following.

First, try to build test programs linking GMP – goto 'unumber\test_gmp' and run 'run.sh'. If it builds 2 executables, then GMP is installed. If not, then:

1) Test that 'm4' is installed: 'm4 –help'. If not:
    a. Unpack 'm4', then 'sh configure', then 'make'
    b. 'make' may give some error, but it may still have done the job
    c. copy executable from './src/m4' to '/bin'
    d. test that m4 works
2) Unpack GMP and configure with C++ 'sh configure --enable-cxx'
3) 'make'
4) Copy 'gmp.h' and 'gmpxx.h' to include directory (can be /usr/include)
5) Directory '.libs' must have 'libgmp.a' and 'libgmpxx.a'.
6) Copy content of '.libs' to lib directory (can be '/lib' or '/usr/lib')

If you have build GMP as described above try to run 'run.sh' again and make sure it builds. If not, seek professional help.

Next, we build CEAL with GMP:

```
'make PLAT=unx GMP=1'
```

3. Cryptoleq Abstract Machine

3.1 Description of the Model

What is memory layout

Cryptoleq Abstract Machine is a model of a processor operating on sequence of memory cells. Each memory cell has an address and value, and any value can be considered to be an address, i.e. a value of a memory cell is always an address to existing or not existing memory cell. And the opposite is true: any address can be stored as value in a memory cell.

What is instruction

The processor has an instruction pointer IP and executes instructions read from the memory. Each instruction consists of three operands, named A B C. Let [*X*] denote the value of memory cell with address *X*. Each operand is fetched from the memory sequentially:

$$A = [\mathrm{IP}]$$
$$B = [\mathrm{IP}+1]$$
$$C = [\mathrm{IP}+2]$$

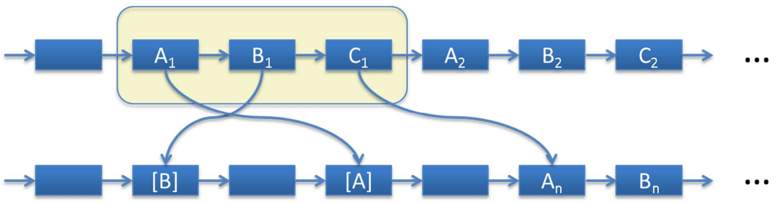
And after that two other memory cells are accessed [*A*] and [*B*]. The instruction modifies [*B*] and IP according to the following two steps:

$$[B] = O_1([A],[B])$$
$$\mathrm{IP} = \begin{cases} C, & \text{if} \quad O_2([B]) \le 0 \\ \mathrm{IP}+3, & \text{otherwise} \end{cases}$$

where O_1 and O_2 are some operations which are to be defined later. O_1 results in a value of cell type, and O_2 results in an integer. For future convenience it's handy to introduce a Boolean function "Less or equal to zero" (Leq) with a cell value argument:

$$\mathrm{Leq}(x) \overset{\text{def}}{=} (O_2(x) \le 0)$$

How exactly IP is incremented and how addresses are organized in a sequence described in Encryption Scheme section after an introduction of TS-values.



What is the execution process

The execution process is the following. First, values of cells A and B are fetched. Then, values of the cells [A] and [B] (see Figure) are fetched. The address of cell [A] is the value of A and the address of cell [B] is the value of B. The operation O_1 of equation above is made and the result is stored to [B]. Next, the test Leq is performed on that value (value of [B]). If the test is positive, i.e. the value O_2([B]) less or equal to zero, then fetch value C and assign to IP, which is effectively a jump to the address of the cell where C points to. If the test is negative, set IP to the address of the cell one past C - same as - if IP was pointing to A, then increase IP by 3. After this repeat the whole operation again.

A special value, which by convention corresponds to unencrypted -1, is used for the following operations:

- input, if used as A operand;

- output, if used as B operand;
- halt, if used as C operand.

Note that this value is special only when used as an address (A, B, or C), but not as a value for arithmetic operations ([A] or [B]).

In Cryptoleq the operations O_1 and O_2 are defined as:

$$O_1(x,y) = (x)_{N^2}^{-1} y \quad \bmod N^2$$
$$O_2(x) = \left\lfloor \frac{x-1}{N} \right\rfloor$$

Here and further the notation $(\cdot)_X^{-1}$ means reciprocal (modulo inversion is an expensive operation. Later we will demonstrate how this operation can be avoided) in modulo *X*. Floor brackets $\lfloor \cdot \rfloor$ is taking integer part from the result of division in the equation above. The reason why this particular selection of operations *O* has been chosen will be explained in the next section.

O_2 translates an encrypted value into an integer, but we are yet to determine the range of negative values to make Leq operation be well-defined.

What is the difference with Subleq

Different operations *O* would result in different model behavior. For example, Subleq neatly falls into the description above with operations:

$$O_1(x,y) = y - x$$
$$O_2(x) = x$$

What are benefits of this model

The encryption scheme (next Section) shows that Cryptoleq is homomorphic to Subleq. A particular selection of encrypted value representation makes the Cryptoleq code backward compatible with Subleq. That means that programs written in Subleq can be run on Cryptoleq processor. The modular inversion and multiplication correspond to subtraction in Subleq. Operation Leq corresponds to Subleq test if the values are unencrypted (for encrypted values Leq returns random result which is a part of design). Cryptoleq processor is oblivious to whether the values are encrypted or they are a subset of values sufficient for Subleq. It allows mixing encrypted and unencrypted computations without having different modes for those two. As in Subleq, higher operations such as multiplication and division are implemented on a software level of the model. To be able to perform Fully Homomorphic calculations the model is required to do multiplication. This is a tricky business and it will be described later in detail.

3.2 Encryption Scheme

How to encrypt/decrypt

Let *N* be a cryptographic parameter equal to the product of two primes. The Paillier encryption scheme is defined as unique correspondence between value *x* from $\mathrm{Z}_{N^2}^*$ and values *m* and *r* from Z_N and Z_N^* :

$$x = r^N g^m \mod N^2$$

where g is a generator in $\mathbb{Z}^*_{N^2}$. We select $g = 1 + Nk$ with a random k prime to N. In our encryption model r serves as a random part of an encryption, and m is the value in question. The private key in this scheme is composed of values k and the Euler totient function φ. Hence the encryption is defined as:

$$x = \mathrm{Enc}(m) \equiv r^N (1 + Nkm) \mod N^2$$

and decryption

$$m = \mathrm{Dec}(x) = \frac{x^{\varphi (k\varphi)^{-1}_N} - 1}{N}$$
$$r = \left(xg^{-m}\right)^{N^{-1}_{\varphi}}$$

As before $(\cdot)^{-1}_X$ means reciprocal in X.

TS values

An interesting observation is that the encrypted value x is bijective to another pair of values t and s:

$$x = 1 + Nt + s$$

with t=[0,N-1], s=[0,N-2], and (s+1) being prime to N. If this equation is a reconstruction of x from t and s, then the opposite is:

$$t = \left\lfloor \frac{x-1}{N} \right\rfloor$$
$$s = ((x-1) \mod N)$$

Negative values, beta

To introduce the concept of a negative value let us break the range of possible numbers (0,N) into two parts:

- positive - a number greater than zero of which the position of the highest bit is less than the position of the highest bit of N; and
- negative - a number with its highest bit being of the same bit position as the highest bit of N

Since N is not a power of 2, the range of negative values is smaller than the range of positive. Let us define a valid positive number as a number which

- has corresponding negative counterpart; and
- does not become negative when doubled.

A natural parameter governed by this definition can be expressed as the largest power of two 2^{β} with

$$\beta = \lfloor \log_2 (N - 2^{\lfloor \log_2 N \rfloor}) \rfloor \qquad (1)$$

so that any number greater than zero and less that 2^β is a valid positive number in the program. Hence the whole range of integers in the program is split into the following parts:

- zero;
- valid positive numbers $< 2^\beta$;
- valid negative numbers; and
- other numbers testing positive but not expected to be used in the program.

The concept of negative values is important in arithmetic operations; and β parameter introduced above, as will be shown later, is used in non-branching algorithm of multiplication.

Homomorphism of TS

The operation O_1 is homomorphic to subtraction of *m*'s with recombination of random parts:

$$x^{-1}y = r_x^{-N} g^{-m_x} r_y^N g^{m_y} = (r_x^{-1} r_y)^N g^{m_y - m_x} \mod N^2$$

Multiplication of *y* by inverted *x* corresponds to subtraction of their unencrypted values.

Why use TS instead of directly encrypted values

The view of an encrypted value *x* as *t* and *s* (TS-value) has some benefits. First, a simple definition of negative values for O*2* is possible. Comparing O*2* in equation where is was defined and *t* in equation where it is calculated from *x*, we see that *t* can be considered to be negative when it is above the highest power of 2 fit in *N*; or in other words *t* is negative when its highest bit is of the same bit position as the highest bit of *N*.

That is handy because of another feature: when *s* is zero, *t* has the same homomorphic property as encrypted *m*:

$$x^{-1}y = (1 - Nt_x)(1 + Nt_y) = 1 + N(t_y - t_x) \mod N^2$$

but with subtraction of *t* values. This property allows Cryptoleq processor to execute instructions with such values with the same result as if the instructions were executed on Subleq processor.

Moreover, the value with *s*≠0 still can be used in addition and subtraction because a special unit for each particular *s* can be defined. Let $U = 1 + Nu$ be a unit incrementing value *t* by 1, so that the instruction:

$$U^{-1}(1 + Nt + s) = 1 + N(t + 1) + s \mod N^2$$

This equation solves into:

$$u = (1 + s)_N^{-1} \qquad (2)$$

which means that *U* can be used as a unit for any encrypted value to change part *t*.

Cryptoleq addresses
Since the addresses in Cryptoleq are of the same type as operand values of Cryptoleq instructions, the operator *U* makes encrypted values suitable as addresses of memory, because the encrypted values obtain concepts of sequence. Arithmetic in equations defining fetching and execution of the instruction becomes a simple multiplication by different powers of *U*, or in TS-value representation a mere change of *t* part.

Compatibility with Subleq
Having the definitions of operations *O*, concepts of negative values and organization of memory with sequential addressing; Cryptoleq processor is able to execute code oblivious to whether values are constructed by encryption or merely from open *t* and *s* values. This gives two advantages:

- Any Subleq program can be run on Cryptoleq processor (compatibility); and
- Both encrypted and unencrypted computation can be mixed in the same program.

3.3 Branching on encrypted data

Why the model is not complete
At this point according to the description, Cryptoleq processor is able to add, subtracts, and make conditional jump on non-encrypted values - the same basic operations as in Subleq. What is missing so far is multiplication (division can be built on top of multiplication) and comparison. Subleq solves this problem by software algorithm of multiplication expressed in Subleq instructions. The same can be done in Cryptoleq. However since the multiplication algorithm uses conditional jump and conditional jump in Cryptoleq is available only for non-encrypted values, the multiplication algorithm in Cryptoleq can be implemented as-is for non-encrypted values only. This is the consequence that encrypted values cannot be used for conditional jump (otherwise they leak secure key information). Or if they are used, the jump based on the arithmetic sign of *t*-value is unpredictable (but deterministic). To solve this problem we introduce a special software function *G* that can be used in a non-branching (without conditional jump) multiplication algorithm.

4. Multiplication over Encrypted Values

4.1 Function G

Definition
Function *G* is a software module which is assumed to be difficult to reverse engineer. The idea is to create the simplest mathematical function which is sufficient for construction of any other complex algorithm, such as multiplication or equality test.

Let *x* be encrypted *m*; let $\bar{x} = 1 + Nm$ be the corresponding unencrypted value; let $\tilde{y}$ be re-encrypted *y*, i.e. $y^{-1}\tilde{y} = r^N$ with some random *r*; and $\tilde{0}$ be encrypted zero; then function *G* is defined as:

$$G(x,y)=\begin{cases}\tilde{0}, & \text{if } \mathrm{Leq}(\bar{x})\\ \tilde{y}, & \text{otherwise}\end{cases}$$

Function *G* takes two encrypted arguments and outputs re-encrypted second argument if the unencrypted value of the first argument tests positive; otherwise it outputs encrypted zero.

It turns out that function *G* can be expressed in Cryptoleq instructions. Since Cryptoleq instruction is multiplication, taking a value to a power is trivial - it is a sequence of squares and multiplications based on bit expansion of the exponent. If an encrypted value is taken to the power $\varphi(k\varphi)_N^{-1}$, it actually decrypts into unencrypted Cryptoleq representation:

$$x^{\varphi(k\varphi)_N^{-1}}=\left(r^N(1+Nkm)\right)^{\varphi(k\varphi)_N^{-1}}=(1+Nkm)^{\varphi(k\varphi)_N^{-1}}=$$
$$=1+Nk\varphi(k\varphi)_N^{-1}m=1+Nm \mod N^2$$

Note, that $r^{N\varphi}=1 \mod N^2$. The value $\varphi(k\varphi)_N^{-1}$ does not have to be stored in the program since only its bit expansion governs the sequence of multiplications in calculation of function *G*.

Is G enough for multiplication

As it appears in the next section, function *G* is sufficient to implement a multiplication algorithm on encrypted values.

4.2 Multiplication algorithm

Multiplication algorithm (Do not get confused with Cryptoleq instruction multiplication: the instruction multiplication is homomorphic to addition in unencrypted domain whereas the multiplication algorithm is built on top of the addition operation) is a software function which produces an encrypted product given two encrypted factors. The requirements to the algorithm are:

- it should be based on addition and subtraction;
- it uses *G*; and
- it should not have conditional jump, no branching (More precisely conditional jump based on encrypted value; otherwise security of the encryption is compromised).

What are valid ranges, β

Non-branching requirement for multiplication algorithm makes algorithm iterations be deterministic and not depending on the arguments of the algorithm. Previously we have defined the ranges of valid positive and negative values for Cryptoleq program in such way that the parameter β can be used in iterations without the risk of overflowing positive values.

What is multiplication algorithm? Top-down

First, let us see that it is sufficient to construct multiplication only for positive values because multiplication of arbitrary values can be reduced. Consider:

$$\begin{aligned}
xy &= z_1 z_3 - z_2 z_3 \\
z_1 &= G(x,\tilde{1})G(y,\tilde{1}) + G(-x,\tilde{1})G(-y,\tilde{1}) \\
z_2 &= G(x,\tilde{1})G(-y,\tilde{1}) + G(-x,\tilde{1})G(y,\tilde{1}) \\
z_3 &= |x| \cdot |y|
\end{aligned}$$

with

$$|x| = G(x,x) + G(-x,-x)$$

This equation expresses multiplication of arbitrary two numbers *xy* through a multiplication of their absolute values and multiplications of *G* functions which are, in this case, either $\tilde{0}$ or $\tilde{1}$. Negation is done as usual in modular arithmetic.

Algorithm 1 shows a simplistic non-branching multiplication procedure.

Algorithm 1 Multiplication: top level

```
1:  procedure MULTIPLY($x,y$)
2:      $sum \leftarrow \tilde{0}$
3:      for $(\beta+1)$ times do
4:          $z \leftarrow$ Div2($x$)          ▷ Div2 to be defined in alg. 2
5:          $bit \leftarrow x-(z+z)$
6:          $sum \leftarrow sum + G(bit, y)$
7:          $y \leftarrow y+y$
8:          $x \leftarrow z$
9:      end for
10:     return $sum$
11: end procedure
```

This algorithm uses a function Div2() returning integer division by 2. This function in turn has to be expressed as a non-branching Algorithm 2.

Algorithm 2 Division by 2

```
1:  procedure DIV2($x$)
2:      $sum \leftarrow \tilde{0}$
3:      $p_2 \leftarrow 2^\beta$
4:      for $\beta$ times do
5:          $p_2 \leftarrow$ Half2($p2$)     ▷ Half2 to be defined in alg. 3
6:          $y \leftarrow sum + p_2$
7:          $y \leftarrow y+y-x$
8:          $sum \leftarrow sum + G(\tilde{1}-y, p_2)$
9:      end for
10:     return $sum$
11: end procedure
```

Function Half2() returns division by 2, but the argument can only be a power of 2. This function can be expressed as shown in algorithm 3. However, in practice a lookup table with precalculated values is much faster and is used in our prototype. Algorithm 3 is simple and presented here for theoretical completeness.

Algorithm 3 Division by 2 of power of 2

```
1: procedure HALF2($x$)
2:     $sum \leftarrow \widetilde{0}$
3:     $p_2 \leftarrow \widetilde{1}$
4:     for $\beta$ times do
5:         $y \leftarrow p_2 + p_2 - x$
6:         $y \leftarrow G(|y|, \widetilde{1})$
7:         $y \leftarrow G(\widetilde{1} - y, p_2)$
8:         $sum \leftarrow sum + y$
9:         $p_2 \leftarrow p_2 + p_2$
10:    end for
11:    return $sum$
12: end procedure
```

The absolute value used in this algorithm (line 6) is defined again via function *G*.

4.3 Equal function

How to calculate comparison

Sometimes other arithmetic operations over encrypted values are required in programs. For instance, PIR example requires comparison of equivalence of two encrypted values. Let us define this operation as:

$$\mathrm{Equal}(x, y) = \begin{cases} \widetilde{1}, & \text{if } \mathrm{Dec}(x) = \mathrm{Dec}(y) \\ \widetilde{0}, & \text{otherwise} \end{cases}$$

This function can easily be expressed via function *G*:

$$\mathrm{Equal}(x, y) = \widetilde{1} - G(x - y, \widetilde{1}) - G(y - x, \widetilde{1}) \qquad (3)$$

Again, tilde in the equations above denotes an encrypted value.

5. Quick introduction into CEAL

This section is a quick overview of some complex CEAL features without deep explanations. It is recommended to return back to this section after reading Section 6.

5.1 General notes

Enhanced Assembly Language

Cryptoleq Enhanced Assembly Language (CEAL) is designed to facilitate writing extended programs such as PIR example. The software library including the multiplication algorithm has to be written in Cryptoleq itself, and its complexity requires more expressive language than mere sequence of Cryptoleq instructions. CEAL allows writing symbolic notation avoiding explicit memory address manipulations. A CEAL compiler arranges memory, resolves the addresses, evaluates expressions, substitutes macro definitions, and does some other things. It can also generate function *G*.

Current implementation of CEAL compiler and emulator can work with two representations: X-values and TS-values. X-value is a number in the range $[0,N^2)$ representing the left hand side of equation $x = 1 + Nt + s$. TS-value are two numbers *t* and *s* in the range $[0,N)$ representing the right hand side of this equation. TS-values are written down as two numbers separated by a dot: **`T.S`**. This is shown in the following example.

How addresses are organized

Previously we described how addresses are organized in Cryptoleq memory model. The following simple example

```
.pragma r=2 PQ=5.5        #pragma N=25
b=$random()               4.23:4.23 24 7.23
b:b (-1)                  4.23 4.23 24
b b (-1)
```

shows an input text and the compiled Cryproleq code ready for execution. The input specifies some encryption parameters: a random seed and two primes. Address **`b`** is defined as an expression generated by a built-in function. In this example this value **`4.23`** becomes the address of the first memory cell and its value at the same time. Semicolon defines a label or an address: **`b:`** would be a label if **`b`** is not defined, or **`b:`** is an address (since it has assigned value). The same in the output: **`4.23`** is initial address. The first instruction has 2 explicit operands, the third one (implicit) is the address of the next memory cell which is **`7.23`** because the addresses of the operands of the first instruction are **`4.23`**, **`5.23`**, and **`6.23`**. The first instruction outputs the value of **`b`** and the second instruction sets the value of **`b`** to zero and halts the program. Both instructions use the special address (-1) which becomes **`24`** because N=25. The compiled code is presented in TS form: two numbers $t.s$ separated by a dot. S part $.s$ is optional if s=0. The Cryptoleq code (the right column) specifies N and declares 6 memory cells with starting address **`4.23`**. In this simple example the addresses are sequential and share the same S part, but in general this is not the case - a program can have many memory sections with different s.

How encryption is handled

CEAL provides syntax for both specifying encrypted values directly and encrypting during compilation. The compiler can also generate random encrypted values to be stored in memory or values to be used as memory addresses. So the program data can be easily initialized with encrypted values as well as placed in randomly selected memory addresses.

5.2 Arrays

How arrays are statically processed and dynamically accessed

As any other high level programming languages, CEAL can allocate memory arrays and work with access to memory cells by either indexes or pointers. This is supported by Cryptoleq instructions (in a similar way how it is done in Subleq) via self-modifying code. To access memory indirectly, via a pointer, for reading or writing, one instruction writes an address into a corresponding operand of the target instruction. How this is done exactly requires some knowledge of programming in Subleq, and is discussed in more detail in "Hello, World!" example. The CEAL compiler allocates required amount of memory initializes it and assigns the addresses. An array can be placed in a continuous block of memory with all elements sharing s part of their addresses. The syntax allows to choose either the default block of memory (with s=0), or starting from a particular address explicitly specified, or at random position. In the last two cases the addresses of the array elements look like encrypted values. The access to an array element involves pointer arithmetic which is as simple as modification of *t* part: the program should use the unit value to navigate through the array elements.

5.3 Library, Multiplication and G Function

Macros and Function calls

Macros are introduced into CEAL to simplify repetition of instruction sequences, to make a local scope for names, and to build more complex construction as functions. An example of the simplest macro

```
.def clear x
     x
.end
```

defines a macro command **`clear`**, which serves more aesthetic purposes rather than practical: an instruction '**`x`**' on line 2 is a synonym of '**`x x ?`**' where the question mark denotes the address of the next memory cell, i.e. the address of the next instruction. Due to the nature of Cryptoleq operation O_1 the value of the cell pointed by **`x`** becomes 1, which is equivalent to unencrypted zero. Any macro defines a sequence of instruction which are inserted at a place of macro usage directive. Before such substitution the CEAL compiler resolves local names: labels and variables. More complex examples of macros can be found in Section 6.

To reuse useful general macros CEAL uses a library - a Cryptoleq code which can be attached to a program by an include directive. This is similar, for example, to C include directive and C standard library linking. The CEAL library is written in CEAL and implements many macros and algorithms such as multiplication of encrypted values.

Since macros are recursive, macro definitions can use other macros, usage of macros becomes prodigally when the macro is big. For example, a usage of multiplication macro is wasteful as every multiplication would require a substitution with too many instructions. In this case the code is organized as a static instantiation of a macro with an entry and exit addresses and a list of input and output arguments. Such mechanism is called a function. This is done in the current CEAL implementation for multiplications of encrypted and unencrypted values, *G* function, and equal operation on encrypted values. The idea is that a short macro substitutes a directive with a several instructions cooking up the arguments and the return address, and then passing the control to the entry point of the function. When the function is done it returns to the point of the call with the result stored at the output arguments.

How multiplication is introduced into CEAL

The CEAL library (current implementation) defines two multiplication algorithms: one for encrypted and the other for unencrypted values. The reason is that the multiplication of unencrypted values *can* use conditional jump hence can be implemented in more optimal way. The encrypted multiplication in Cryptoleq is implemented in the library by a code written in CEAL implementing all three algorithms (The 3rd algorithm is implemented as a look up table for efficiency) from multiplication section.

How G is generated

The current implementation of CEAL compiler and its library uses a built-in directive "autobits" to generate function *G*. The directive takes three arguments: the first is a constant expression, and the other two are names of macros. The compiler replaces "autobits" macro usage with a list of macro directives constructed by binary expansion of the value of its first argument: one for each argument macro names for 0 or 1. The decryption function can be generated by giving the value $\varphi(k\varphi)_N^{-1}$ as the first argument and macro names of two operations corresponding to bit 0 and bit 1: 0 - squares the argument *x*; and 1 - multiplies the accumulator by the argument *x* and then squares the argument *x*. This simple procedure produces the unencrypted value of *x*. The next step in function *G* is to test this value with Leq, then to generate random $\tilde{0}$, and depending on the test return this random $\tilde{0}$ or return random $\tilde{0}$ multiplied by the second argument to function *G*.

5.4 Examples

In this section we would like to show a few simple example programs written in CEAL.

The following program outputs first eleven encrypted **Fibonacci** numbers.

```
.pragma r=2 PQ=7.11                          1
start:                                       2
.out    a                                    3
.mov    b c                                  4
.add    a c                                  5
.mov    b a                                  6
.mov    c b                                  7
m1 counter start                             8
.halt                                        9
. a:~1 b:~1 c:0                              10
. counter:-10 m1:-1                          11
.include "general.lib"                       12
                                             13
13.15 9.24 12.14 54.66 10.3                  14
42.36 0.70 13.8 7.22 56.52 57.63             15
                                             16
1 1 2 3 5 8 13 21 34 55 12                   17
```

The program is lines 1 to 12 in the listing above. Lines 14-15 is the output of the program, and line 17 is the output after decryption is applied. The first line of the program is one of three built-in directives "pragma". The other two are "autobits" and "include". Pragma directive defines the encryption parameters: random seed and two primes. In this program *N*=77 which means that *β*=3 according to equation 1. The range of valid arithmetic values to be used in a program with such *N* is [-8,8]. This range is guaranteed to work correctly for secure multiplication algorithm hence for any arithmetical computation. Since this program does not use multiplication, we see that the values can safely go up to 77. Line 2 is a label which the compiler will replace at the instruction, line 8. Lines 3 to 7 and 9 are macro directives defined in the library "general.lib" included at line 12. Their meaning is obvious: output memory **a**, then assign **c** value stored in **b**, add value from **a**, and then move values **b** to **a** and **c** to **b**. Macro "out" is defined in the library as:

```
.def out x
     x (-1)
.end
```

only one instruction where the third operand is implicit '`?`', i.e. '`x (-1) ?`'. Macros "add" and "mov" are:

```
.def add x y : Z
     x Z; Z y; Z
.end
.def mov x y
.clear  y
.add    x y
.end
```

They differ by only clearing the argument. Macro `add` uses a global variable `Z` defined in the library as "`. Z:0`". This variable is used as a temporary. The list of arguments in macro definition separated by a colon specifies global variables - names which should not be resolved within the macro definition. Macro `add` has three instructions: move `x` inverted value to `Z` (because `Z` is 0), add inverted again value to `y`, then clear `Z`. Note that each instruction has 3 operands, which are implicit, and which are added at the compilation.

The macro "halt" is

```
.def halt : Z
     Z Z (-1)
.end
```

Line 10 starts with a standalone dot. This tells the compiler that the following data is not a 3-operand instruction, but an arbitrary data. This line defines three memory cells and initializes them with $\tilde{1}$, $\tilde{1}$, and 0. The tilde in front of 1s tells the compiler to encrypt, so `a` and `b` are initialized with randomly encrypted 1's. Line 11 similarly defines two other memory cells. Their purpose is to stop the program after 11 iterations. Line 8 adds `m1` to `counter` and jumps to `start` while counter is not positive. The final comment on this program is the very last output is 12 which is wrapped 89 because *N*=77.

Another example is a **Hello, World!** program showing pointer iteration over an array. The following version of the program does not use the library.

```
.pragma io=ascii r=4 PQ=7.11                    1
a; p Z; Z a; Z                                  2
a:0 (-1); m1 p                                  3
a; E Z; Z a; Z                                  4
p a t; Z Z 0                                    5
t:a; p Z; Z a; Z                                6
E a (-1); Z Z 0                                 7
. p:H Z:0 m1: -$unit(H)                         8
{ H: "Hello, World!\n" E:E }                    9
```

The first line is similar to one in Fibonacci program with an exception of parameter `io` which tells to use character input/output rather than numbers. Line 2 has four instructions which move value from `p` defined on line 8 to `a` defined on line 3 '`a:0`'. Memory cell `p` is initialized with the address of the first element of the array holding

"Hello, World!" string. Line 3 outputs the element of the array, and then increments pointer **`p`** by a unit **`m1`**. The value **`m1`** is initialized by a built-in function calculating a unit increment in *t* as in equation 2. This is necessary because line 9 has curly braces **`{}`** which direct the compiler to place the array in a random memory position by generating a random address. This built-in unit function takes this address **`H`** as an argument and uses its *s* part. After the instruction '**`m1 p`**' pointer **`p`** points to the next element of the array. Line 4 moves value **`E`** to **`a`**. Memory cell **`E`** holds its own address and serves as the termination sign of the array. Line 5 has two instructions: first tests **p≤E** (**a←E**), and if yes - jump to **`t`**; if no - the second instruction is executed making jump to the beginning of the program, i.e. new iteration. Obviously, since **`p`** and **`E`** hold encrypted values, comparison greater or less does not make sense - the result is random. This test was used to detect that **p≠E**. If the execution is passed to line 6, this is either because **`p=E`** or **`p<E`**. The same trick is done here: move **`p`** to **`a`** and then test **E≤p** (**a←p**). Now if test gives positive result it is only when **`p=E`** and the program halts; otherwise the execution passed back to the beginning.

Compare the above version with the version using the library.

```
.pragma io=a
.mov    B p
start:
.mov21  p a
.out    a
.inc    p
.ifneq  p E start
.halt
. p:0 y:0 B:H a:0
. H: "Hello, World!\n" E:E
.include "general.lib"
```

Macro **`mov21`** is a dereferencing a pointer:

```
.def mov21 x y : Z
.clear  y
.clear  z
.mov    x z
        z:0 Z; Z y; Z
.end
```

And **`inc`** is a simple increment

```
.def inc x : om1
     om1 x
.end
```

with a constant defined in the library '**`. om1:-1`**'. And **`ifneq`** is a conditional jump "if not equivalent". It uses another simple macro **`goto`**

```
.def goto x : Z
     Z Z x
.end
```

and is defined as

```
.def ifneq A B C
.mov    A a
```

```
        B a t
.goto   C
        t:a
.add    B a
        A a end
.goto   C
. a:0
end:
.end
```

This macro is used in the following even simpler looking program which prints a few first encrypted **Factorials**

```
.pragma r=2 PQ=101.113                              1
start:                                              2
.smul   x i x                                       3
.out    x                                           4
.inc    stp                                         5
        n i                                         6
.ifneq  stp max start                               7
.halt                                               8
. x:~1 i:~1 n:~-1                                   9
. stp:1 max:8                                       10
.include "secure.lib"                               11
                                                    12
5241.10388 3208.5876 10428.4625 10625.8131          13
9916.4208 2805.10730 986.2275                       14
                                                    15
1 2 6 24 120 720 5040                               16
```

Variable `i` is an encrypted index running from 1 to 7, `x` is a factorial accumulator; `stp` and `max` are loop variables. Macro `smul` is a header for encrypted multiplication function defined in the library "secure.lib". Lines 1-11 is the program itself, lines 13-14 is the output, and line 16 is decrypted output.

The following section investigates efficiency of Cryptoleq using another more complex example.

5.5 PIR Example

What is the PIR example

Private Information Retrieval (PIR) example is a classic example for FHE calculations. It is also sometimes called Oblivious Transfer since the database is not open to the processor.

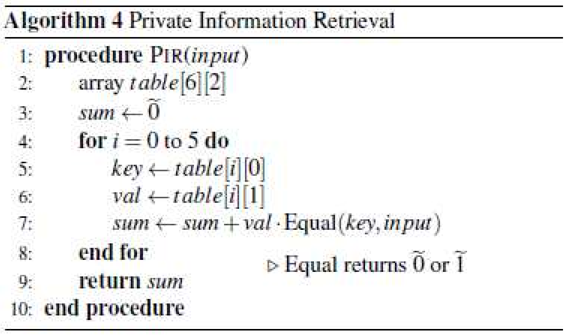
Consider a situation when a user keeps an encrypted database on some server. When required the user wants to make inquiry to the database and obtain a result. At the same time the user does not want the server or any party controlling the server to be able to reveal any bit of information about inquiry, data in the database and the return output. There are three options what the user can do: download encrypted database entries; run on the server decryption on the fly; or run FHE calculation on the server. We reject the first two options for obvious reasons. Cryptoleq with its implementation of secure multiplication provides full equivalent of FHE calculations.

As a simple case let database be a list of 12 numbers represented in two columns with each row being a table entry a key-value pair:

1	2
3	4
5	6
7	8
9	10
11	12

An inquiry to the database is a particular key, let’s say 7, and the expected output is a corresponding value: 8. Suppose the database and the input are encrypted. The user wants the program to go through all encrypted entries, secretly comparing keys with the input and to store the value when one of the keys matches.

The logic of the program can be expressed as Algorithm 4.



Algorithm 4 Private Information Retrieval

```
1: procedure PIR(input)
2:     array table[6][2]
3:     sum ← $\widetilde{0}$
4:     for i = 0 to 5 do
5:         key ← table[i][0]
6:         val ← table[i][1]
7:         sum ← sum + val · Equal(key, input)
8:     end for                    ▷ Equal returns $\widetilde{0}$ or $\widetilde{1}$
9:     return sum
10: end procedure
```

The flow of this algorithm ensures both: that no data is required to be decrypted; and that there is no branching based on sensitive data.

How PIR example is implemented

This algorithm is implemented in CEAL is straightforward. It iterates through the entries of the table accumulating the result of multiplication of the value of each entry and the result of equal function between a key and the input.

```
# PIR example
.pragma PQ=29.101 k=5 r=17 io=ts
.mov    szero   sum
start:
.mov21  p key
.inc    p
.mov21  p val
.inc    p
.dec    dbsz
.dec    dbsz
.seq    key input x
.smul   x val y
.add    y sum
.ifneq  dbsz ozero start
.out    sum
.halt
. sum:0 key:0 val:0 p:db_start x:0 y:0
db_start: # data
.include datax "db.sec"
db_end:
input:;.include datax "input.sec"
dbsz: (db_end - db_start)
. ozero:0 szero:~0 # constants
```

```
    .include "secure.lib"                              24
```

Macro **`dec`** is the same as previously described macro **`inc`** but decreases the value by 1 instead of increasing. Directives include on lines 19 and 21 have extra function **`datax`** telling the compiler to treat include data as a sequence of X-values. Macro **`seq`** (secure equal) compared two encrypted values and returns encrypted $\tilde{0}$ or $\tilde{1}$. Macro **`smul`** is secure multiplication. Pragma on line 2 specifies value *k* and format of input/output **`ts`**. Variable **`dbsz`** is initialized with the size of the database, and decremented by 2 on each iteration. The program terminates when it reaches 0.

C++ prototype

The PIR example has been implemented in C++. Class **`sec_int`** implements the encryption structure and algorithms described above. A snippet (the main function) is presented in the listing below.

```
int main() try
{
    sec_int tbl(file);
    sec_int sum = sec_int::zero;
    for ( int i = 0; i < tbl.size(); ++i )
    {
        sec_int key = tbl[i+i];
        sec_int val = tbl[i+i+1];
        sum += equal(key,input) * val;
    }
    save(sum);
}
```

6. Gentle introduction into CEAL grammar

This big section is not structured by elements of CEAL, rather it introduces syntax or semantic elements by gradually increasing complexity.

6.1 A Note on Continuous Enhancement

In this section we show that the output of the compiler is often a valid CEAL program. Exceptions to this are (-1) when *N*=0 and data instructions which size is not multiple of 3. This compatibility arises from the idea that CEAL as a language grows from Cryptoleq code as sequential steps of improvements rather than a fresh new language. See more in section 10.1.

6.2 Syntax: CEAL vs Cryptoleq

Consider the following CEAL program

```
1 2 3
4 5 6
```

which compiles into the following code

```
#pragma N=0
1 2 3
4 5 6
```

If the same is given back to the compiler it will produce exactly the same output. This is because: (1) numbers are treated as open TS values and the result is produced in TS values (by default, otherwise pragma can be used to specify the type); and (2) '`#`' is a symbol of comments – everything to the end of line is ignored.

When this code is executed it produces the error: "Memory access violation". This program is ill formed because: (1) it modifies the memory cells of the instruction being executed; (2) it accesses the memory 6 which was not defined; and (3) it does not have proper termination. Let us fix these problems. This is the same code with addresses shown explicitly:

```
0:1 1:2 2:3
3:4 4:5 5:6
```

A number or label standing in front of the cell definition (each standalone number is a cell definition) followed by colon specifies cell's address. If it is a number, then the address is specified explicitly; if it is a label, then the label gets the value of the cell's address. Multiple labels are allowed, so, for example: **`1:a:b:2`** defines a cell with value 2 and address 1 and also introduces two labels **`a`** and **`b`** which become equal to 1.

The first instruction '**`0:1 1:2 2:3`**' subtracts 2 (the value of B operand) from 3 (the value of C operand) and stores the result into C operand of the first instruction (**`2:3 -> 2:1`**). Actually Cryptoleq does multiplication, but since these values are *open* values (*t* of TS with *s*=0) we can think in subtraction and addition terms.

First, operand B of the instruction should point outside of this instruction to avoid race conditions (rather implementation specific behaviour), let's subtract the second to the last cell from the last cell: '**`4 5 3`**'. If this would result into (-1), then it can serve as halt instruction. The only condition is that the second instruction should result in zero or negative value. This can also be done by subtracting, for example, the first cell (address 0) from the third one (address 2): '**`0 2 1`**'. Hence the code becomes:

```
4 5 3
0 2 1
```

This works as follows:

1. Memory={**`4 5 3 0 2 1`**}, IP=**`0`**, [IP]=**`4`**
2. **`(1-2)`**→**`(-1)`**, Leq→*True*
3. Memory={**`4 5 3 0 2 -1`**}, IP=**`3`**, [IP]=**`0`**
4. **`(3-4)`**→**`(-1)`**, Leq→*True*
5. Memory={**`4 5 -1 0 2 -1`**}, IP=**`-1`**, [IP]=halt

The convention is that when IP becomes -1, program halts.

Now let's pay attention to the line "`#pragma N=0`" in the output of the compiler. Cryptoleq processor can execute code presented as a sequence of numbers. Each instruction is executed as multiplication of A and B operands. The multiplication is modular and always requires modulo argument *N*. This is achieved by convention that any Cryptoleq code starts with a line beginning with '`#pragma`' following a list of additional parameters where one of them is *N*. CEAL has a corresponding directive '**`.pragma`**' that defines some meta arguments for the compiler as well as the parameters to be passed to the Cryptoleq code – the output. When *N* is not defined, then its default value is 0. In this case Cryptoleq implementation can select the size of the register. The current implementation is 4096 bits. If *N*=0, (-1) is not replaced with *N*-1, but remains in the text as '**`(-1)`**' or '**`-1`**'. CEAL requires parentheses because its syntax has expressions and '**`-1`**' can be seen as a term to the previous expression. Cryptoleq code can use the latter case, although parentheses are also allowed for compatibility. If *N*≠0, (-1) will be outputted as *N*-1.

Cryptoleq code can also contain address definitions using colon syntax. For example, the previous code can be rewritten as

```
100:104 105 103
    100 102 101
```

which is both valid CEAL and Cryptoleq code. The addresses of the cells without explicit address definition get sequential values. The address of the last cell in this example is 105.

6.3 Processor and Compiler parameters

The simplest trivial CEAL program is

```
0 0 (-1)
```

The command line option showing compiler parameters (see section 2.4) produces:

```
Processor: N=0 N2=0 M=0 A2=0 B2=0 beta=0 high_bit_posN=4095
Compiler: p=0 q=0 phi=0 k=1 g=0 Nm1=0 p1Nk1N=0 rnd=0 rndN=0
```

All internal parameters are 0’s except **`high_bit_posN`**, which is required for correct Leq test. Its value 4095 says that the default register size used is 4096 bits. The compiler has an object Compiler (which is not the best selection of the name). This object has a sub-object Processor. The difference is that Processor knows all parameters necessary to run Cryptoleq code and does not know any parameters revealing cryptographic information. Compiler object has all the parameters necessary for encryption and decryption.

Adding *N*

```
.pragma N=77
0 0 (-1)
```

gives

```
Processor: N=77 N2=5929 M=13 A2=64 B2=8 beta=3 high_bit_posN=6
Compiler: p=0 q=0 phi=0 k=1 g=0 Nm1=0 p1Nk1N=0 rnd=0 rndN=0
```

and

```
.pragma PQ=7.11
0 0 (-1)
```

gives

```
Processor: N=77 N2=5929 M=13 A2=64 B2=8 beta=3 high_bit_posN=6
Compiler: p=7 q=11 phi=60 k=1 g=78 Nm1=53 p1Nk1N=9 rnd=0 rndN=0
```

The values are explained in the table below

`N2=N`^{2}	**`beta=`**$\log_2$(**`B2`**)	**`A2`** is the highest power of 2 **`<N`**
`M=N-A2`	**`p`** and **`q`** are factors of **`N`**	**`B2`** is the highest power of 2 **`<M`**
`g=1+Nk`	**`phi=`**(**`p-1`**) (**`q-1`**)= φ	**`high_bit_posN=`**$\log_2$(**`A2`**)
`rndN=rnd`^{N}	**`Nm1=N`**$^{-1}{}_{\mathtt{phi}}$, i.e. N_φ^{-1}	**`p1Nk1N=phi`**$^{-1}{}_N$**`k`**$^{-1}{}_N$, i.e. $(k\varphi)_N^{-1}$

Parameter **`rnd`** is 0 as it has not been specified. This parameter is required to be specified (in pragma) if a random function ($peekrnd or $random) is used in the program or if encryption is invoked by the compiler – random memory addresses or direct encryption instruction such as '**~**'.

6.4 Input and Output

A special value (-1), whether with N=0 or not, is used as a halting address when used in place of operand C. The same value can be used also in place of operands A or B, or both. Here some conventional rules have to be defined. The following table suggest such agreements.

Instruction	Action
`A B (-1)`	[B]=[A]$^{-1}$[B], then halt if Leq([B])
`A (-1) C`	Output [A] according to pragma settings No Leq test – Jump to C unconditionally
`(-1) B C`	Input to [B] according to pragma setting No Leq test – Jump to C unconditionally
`(-1) (-1) C`	Implementation defined This is reserved for System calls with the code C

A program

```
.pragma N=91 io=a
2 (-1) (-1)
```

outputs '**`Z`**' because '**`io=a`**' (same as '**`io=ascii`**') outputs value 90, which is [2], the third cell and ($-1 = N-1 \bmod N$); and ASCII value of '**`Z`**' is 90. Changing pragma to '**`io=ts`**', makes output '**`90`**'; and changing to '**`io=x`**', makes '**`8191`**', which is $1+N(N-1)$.

6.5 Data fields

So far programs were defined via list of instructions and number of cell was multiple of three. Sometimes a program can use memory cells only as data. In this case there is no need to make them multiple of three. Consider a program

```
0
```

Yes, it is just one zero. It compiles into

```
0 0 3
```

Why, because one line is a complete Cryptoleq instruction which must have 3 operands. If operand C is missing it is added as the address of the next cell, so [C] points to the next instruction (in our case it is absent). If both B and C are missing then B gets the value of A and C as described above.

To prevent such compiler behaviour we put a period and space in front of the instruction. So

`(-1) (-2)`	compiles into	`-1 -2 3`
`. (-1) (-2)`	compiles into	`-1 -2`

It is important to put a space after the dot, because dot is used also for directives and macros. This program

```
.pragma io=ascii
6 (-1)
7 (-1) (-1)
. 72 105
```

compiles into 8 cells and prints '`Hi`'. And the program

```
.pragma io=ascii
15 (-1) 3
1 0 6
1 10 9
27 15 (-1)
27 27 0
. 104 101 108 108 111 44 32 119 111 114 108 100 0
```

prints '`hello, world`'. The last line in this program is just a sequence of ASCII codes. The same line could have simply been written as '`. "hello, world" 0`', as will be explained in the following sections. This program is terse, simple, and worthwhile to be understood how it works. When rewritten using named labels, it becomes:

```
.pragma io=ascii
A:H M:(-1)
M A
M B
E B:H (-1)
E E A
. H:"hello, world" E:0
```

The first instruction outputs one character. The following two instructions increment two pointers A and B. The forth instruction subtracts 0 from the current character pointed by B. The trick here is that this subtraction results in 0, and halts the program, when B points to E, i.e. one past the last character of the string. The next instruction is a mere jump to the beginning to repeat the whole iteration until the end condition is not met.

6.6 Using TS

Cryptoleq works with encrypted values and CEAL uses TS notation. In the examples above values were presented as single numbers. That corresponds to open values:

$$x = 1 + Nt$$

when *N*≠0, and to some implementation dependent form when *N*=0. The CEAL syntax allows specifying both *t* and *s* parts of a number using a dot between them. If *s*=0 it can be left out. And the opposite is true: if a number is not presented in the form *t.s*, then *t* get the value and *s*=0. The program

```
0.1 (-1) (-1); . 0.1:65
```

prints **`65`**. Here the address of the cell holding 65 is defined as **`0.1`**, i.e. *t*=0, *s*=1. Note that the data is separated from the only instruction by semicolon – CEAL treats semicolon and end-of-line (EOL) symbol as synonyms. This allows sometimes writing several instructions in one line: for example, '**`a;b;c`**' are three instructions clearing cells named **`a`**, **`b`**, and **`c`**.

If the data in the program above would define more than one cell '**`. 0.1:65 66`**', then the second cell gets the sequential in *t* address '**`. 0.1:65 1.1:66`**'.
The following program

```
.pragma N=91 io=a
0.1 (-1) 3
0.2 0.1 6
0.1 (-1) (-1)
. 0.1:65 0.2:90
```

prints '**`AB`**'. Why? Values 3 and 6 in this program are redundant.

6.7 Definitions and Expressions

In contrast to Cryptoleq code, the value (-1) cannot be used without parentheses because CEAL allows expressions. Consider the following program which prints ‘2’:

```
.pragma io=a                  |
z=-1                          |
A z z                         |           3 -1 -1
. A:'1'-?+(-A)+z+9            |           50
```

The right side shows the compiled code. First, the line '**`z=-1`**' introduces a constant into a program. The right-hand-side is an expression, here, consisting of unary minus and 1. The first instruction (and it is the last one) outputs a character stored in cell A. The value of this cell defined at the last line is an expression. Let’s see

Expression	Value	Comment
`'1'`	49	ASCII value of ‘1’
`'1'-?`	49-4=45	**`?`** is the address of the next cell; address of this cell is 3, so **`?`**=4
`'1'-?+(-A)`	45-3=42	**`A`** is the address of this cell
`'1'-?+(-A)+z`	42-1=41	**`z`** is defined as -1
`'1'-?+(-A)+z+9`	41+9=50	This is ASCII value of ‘2’

Apart from numerical values, characters and string literals are supported with escape sequences. The following shows data and its compiled equivalent:

```
. "ab\\\n" 'c' '\x65'         |           97 98 92 10 99 101
```

For the list of escapes see 9.1.

Label also can be defined in definitions. If more than one label is used, they all get the same value – the address of the current cell. Check if you understand the following example:

```
x=0.1;a=0.2                    |
c:a:b:a                        |           0.2:0.2 0.2 3.2
e:x:f:e                        |           0.1:0.1 0.1 3.1
```

If you do, you understand a lot!

6.8 Encryption of data

If a value of a cell is prepended with tilde '**~**', its content is encrypted by the compiler. Obviously to make an encryption, the compiler must have $N \neq 0$ and random seed to be specified (even if it is 1, i.e. no random part). Consider the following:

```
.pragma PQ=7.11 r=1            |
. "ab" ~"xy"                   |           97 98 43 44
```

Numbers 97 and 98 come from ASCII values of ‘a’ and ‘b’. The string **"xy"** is first broken into two characters '**x**' and '**y**' (values 120 and 121), then encryption is applied to each. For **x**, 120 it is:

$$x = \left(r \cdot (1 + N \cdot 120) \bmod N^2\right)$$

$$\frac{x-1}{N} = 43 \qquad (\, N = 77, \quad r = 1 \,)$$

The first line gets X value, and the second extracts T value. Since *r*=1, *s*=0. The same is done for **y**, yielding 44.

Tilde can be used in front of the dot which starts data. In such case encryption is applied to all elements of data. But if tilde appears at the front of the dot and inside data, then the other tilde negates. As in the following example:

```
.pragma N=7.11 r=2             |
 .  1  2  3                    |           1 2 3
 . ~1  2  3                    |           13.15 2 3
 .  1 ~2 ~3                    |           1 34.24 5.8
~.  1  2  3                    |           71.3 29.15 59.24
~. ~1  2 ~3                    |           1 73.8 3
```

Encrypted values are seen in TS form with $s \neq 0$.

6.9 Arrays

One example of an array is a string literal. Another way to specify an array of zero initialised elements is to give a constant in square brackets:

```
. "ab" [2] a:a                 |           97 98 0 0 4
```

Here '**[2]**' is replaced with '**0 0**'. Arrays also can be initialised with encrypted zeros:

```
.pragma N=7.11 r=2             |
. ~[2] [2]                     |           74.15 61.24 0 0
~. ~[2] [2]                    |           0 0 55.8 67.3
```

In theory Subleq assumes to be working on infinite array of memory and emulators usually support growing memory. For example, Subleq programs can use past program memory as a stack. Cryptoleq is different is this respect. It requires all cells to be defined. [Although it is possible to create cells on the fly, but this depends on the implementation and the organization of memory.] If a Subleq program, which works with extended memory, needs to be run on Cryptoleq processor, it requires to be appended with extra allocation, e.g. '`. [1000]`'.

Square brackets specify the size of elements to be added. If the square brackets contain a name of variable, then this gets the size of the instruction or data field. Consider:

```
. [a] a c                      |           2 5
[b] b                          |           3 3 5
. "hello" [c]                  |           104 101 108 108 111
```

This artificial program (as many above) is only to show the translation rules, and is not to be run.

6.10 Entry point

A Cryptoleq program can occupy fragmented areas of memory. By default the execution start with the first memory cell. The first memory cell is defined as the one which has the smallest *s*, and among equal *s* it has the smallest *t*. The following program

```
.pragma PQ=19.23 entry=10.30   |     #pragma N=437 entry=10.30
.     0: 0.1 0.2 0.3           |     0.1 0.2 0.3
.   100: 1.1 1.2 1.3           |     100:1.1 1.2 1.3
. 400.1: 4.1 4.2 4.3           |     400.1:4.1 4.2 4.3
. 300.1: 3.1 3.2 3.3           |     300.1:3.1 3.2 3.3
10.30: 402.1 (-1) (-1)         |     10.30:402.1 436 436
```

outputs '`4.3`'. It specifies addresses for some cells. Subsequent cells get incremented addresses. For example, the cell with value '`3.2`' gets an address '`301.1`'. This is true for both CEAL program and Cryptoleq code – the loader of the emulator should be smart enough to assign the proper addresses to all cells. Furthermore, it should be smart enough to detect memory address overlapping because sequential increment of addresses may generate an address which is defined somewhere else.

The default entry point for this program is '`0`'. Pragma parameter '`entry=10.30`' redefines it. This information propagates to the “pragma” of Cryptoleq code, so the code after being loaded into memory starts execution from the address `10.30`'.

6.11 Random addresses

CEAL compiler can place cells into address randomly generated. Random seed in this case is necessary. We use curly braces for this. Curly braces work as a data field, i.e. no expansion to 3 cells.

```
.pragma PQ=7.13 r=2                            |
{ 0 }                                          |           61.3:0
```

Remember that random seed can be specified as a number, or as “time”. In the latter case the output is not deterministic – each compilation will very likely produce different output.

In the following example

```
.pragma PQ=17.11 r=2     |
. a b c d e f            |     17 16 15 14 13 12
. A B C D E F            |     12 61.114 29 44 75.134 151.85
 . [a] A: "Opn addr, opn val"      |     79 112 110 32 97 ...
 { [b] B: "Sec addr, opn va" }     |     61.114:83 101 99 ...
~. [c] C: "Opn addr, sec v"        |     29:114.102 34.136 ...
 . [d] D:~"Opn addr, sec "         |     26.85 147.102 ...
~{ [e] E: "Sec addr, sec" }        |     75.134:175.136 ...
 { [f] F:~"Sec addr, se" }         |     151.85:61.68 ...
```

small letter variables show the sizes if data fields, capital letter names are labels – addresses. The arrays are made of different sizes intentionally to show that the values appearing on the first line are different. The first encrypted address labelled B is generated here as '**61.114**'. Note, that address C continues the sequence from the field with label A: C(29)=A(12)+a(17). This means that '**{...}**' can appear in the text in different places without affecting correctness of the program; although the generated random addresses and data encrypted values may be different.

Random addresses can be used as data, but they cannot be instructed to get encrypted.

```
.pragma r=2 PQ=7.11                            |
. A:A                                          |           0
~{ E:~E }                                      |           74.15:74.15
. B:~B                                         |           1:9.24
```

First, note that cell B gets the address one after A. Second, cell B gets the value of its address which is instructed to be encrypted. So the compiler encrypts 1, which gives 9.24=718=$16^{77}(1+77) \bmod 77$. Third, address E is already encrypted (because of '**{...}**'). Two tildes cancel each other, so '**~{E:~E}**' is the same as '**{E:E}**'. If you remove one of the tildes, the compiler gives an error saying that it cannot encrypt an already encrypted value.

6.12 Arithmetic on TS values

In 6.7 we introduced expressions. But Cryptoleq works with encrypted values, not just plain ones. What is the result of expression if an encrypted value is used?

```
.pragma PQ=7.11 r=2                            |
A+1                                            |           13.15 13.15 3
{ A:0 }                                        |           74.15:0
```

Address A is generated as '**74.15**'. The expression '**A+1**' actually means $(xy \bmod N^2)$, where $x = 1 + N \cdot 74 + 15$ and $y = 1 + N \cdot 1$. It is equal to $1017 = 1 + N \cdot 13 + 15$, which appears as values of the two operands in the first instruction.

6.13 X Code

So far examples have been showing compilation to TS values. The CEAL compiler (as well as the emulator) also supports X values. The following program

```
.pragma N=77 cqtype=x io=a    |     #pragma N=77 io=a cqtype=x
a (-1) (-1)                   |     235 5853 5853
. 1.1 2.1 a: 3.3: 'A'         |     79 156 235:5006
```

prints ‘A’. The first memory cell gets value '**`235`**' which is '**`3.3`**' because label '**`a`**' gets this value from the address of the last cell with **`'A'`**. Pragma option '**`cqtype=x`**' tells the compiler to output X values instead of TS. This option propagates to the Cryptoleq code because the loader must know the format of the values. 5853=1+*N*·76(which is -1); 5006=1+*N*·65(which is ASCII of ‘A’).

6.14 Label accumulation

Labels accumulate until they are assigned. CEAL allows empty instructions. Labels are allowed in empty instructions or at the end of instruction (this is a valuable feature for macros). In that case all labels are transferred to the next defined cell.

```
. 7 a:; b:                         |           7
. 8; c:; . 9 a b c                 |           8 9 1 1 2
```

Semicolon and end of line (EOL) are synonyms in CEAL. They end an instruction. In the above example '**`a`**' is declared at the end if instruction (data field), '**`b`**' is declared at an empty instruction. They both obtain value 1 which is the address of the cell holding 8. '**`c`**' is also declared at an empty instruction; so it is assigned the address of the cell with 9.

6.15 Include directive

“Pragma” is one built-in directive. Another one is “include”. Include directive includes the content of another file as if the content is present at the place where the directive is used. This works similar to ‘#include’ in C language. The syntax is

```
.include [function] "file"
```

The function is an optional parameter which can be either “asis” or “datax”. The default, “asis” includes the content of the file the content should be valid CEAL code. “Datax” includes file as if it is a data field with values specified in X form. Let us analyse an example of three files:

```
The program:      .pragma N=77 io=a
                  .include "a.inc"

file a.inc:       a (-1); b (-1) (-1)
                  a:; .include datax "b.inc"
                  b:'B'+(b-a)

file b.inc:       5006
```

This program prints “AC”. The program has a simple include on the second line, so we can consider it to be the body of the program. It has 2 output instructions with halt on the second. Label declaration '**`a:`**' is followed by the semicolon. This is necessary because a label declaration starts a new instruction, but include directive cannot be part of instruction – it has to start at the beginning or after an instruction ends. This semicolon ends the current instruction (EOL would also work). Because of label accumulation described in 6.14, label '**a**' get the address of the next cell, which happens to be included from file "b.inc". This include uses include function "datax". That is the value 5006 inserted into a program as a data field and as TS value 65.0 which corresponds to open value of ASCII code of 'A'. In the current program, this declaration of '**a**' and this include are equivalent to '**a:65**'. Label '**b**' gets value 'C' because '**(b-a)**' is 1. If "b.inc" would contain 5 values, the output were "AG".

The "include" directive complicates a bit generation of error messages with references to problematic places. CEAL compiler properly handles this and if more than one file is involved in compilation, in case of error, it produces a reference to the faulty position and to the corresponding file.

6.16 Simple macros

CEAL has macros. A macro is a piece of code which has a unique name. When this name is used the code is inserted at the place. A macro can have local variables, labels and arguments. It also may or may not see global names. A simple macro is defined in the following example

```
.abc                                    |           0 0 -1
.def abc                                |
a:a 0 (-1)                              |
.end                                    |
```

The first line is a usage of macro **abc**. A usage is a dot immediately followed by a name. A name must not be one of the built-in directives: "pragma", "include", "def", or "end". A macro definition starts with '**.def**', ends with '**.end**', and has the body between those two. In this example it contains only one instruction. The label **a** is used just to show that labels can be used in the same way as in the body of the program. However, they remain local to a macro instantiation. If a macro is instantiated several times, all labels are unique for each instantiation, and it is not possible to use those names from outside of the macro.

Another more advanced example:

```
. Z:0 0 b                               |           0 0 3
b: .abc (-1)                            |           0 0 -1
.def abc x : Z                          |
Z Z x                                   |
.end                                    |
```

The first line is defined as a data field, but when compiled it has three valid operands. Cryptoleq processor does not know about data fields. The first line written as '**. Z:0 0 b**' is equivalent to '**Z:0**'. The label **b** stands in the same instruction with usage of macro **abc**. This is possible (in contrast to include directive) because macros are resolved at evaluation stage of the compiler, and include directive is processed at the tokenization stage. This macro has two extra things: the parameter **x** and the global **Z**. The body of macro definitions uses those two names. The compiler binds the argument '**(-1)**' with the parameter **x**. Parameter **Z** declared as global – this is why colon is between them – is left alone in the body during instantiation. It means that **Z** is resolved as other program's variables.

Bodies of macros can use other macros and macros can be empty:

```
.def aaa                            |
a: a 0 (-1)                         |
.bbb a                              |
.end                                |
.aaa                                |          0 0 -1
.def bbb x                          |
.end                                |
```

Also macros can be used defined in other files which included into the program by “include”. Sometimes it is necessary to jump to the end of the macro body, so macros allow having labels declared at their end. This program

```
.pragma io=a                  |          #pragma N=0 io=a
.aaa;.aaa;.aaa                |          3 -1 4
0 0 (-1)                      |          97
.def aaa                      |          7 -1 8
a (-1) e                      |          97
. a:'a'                       |          11 -1 12
e:                            |          97
.end                          |          0 0 -1
```

prints “aaa”. Its compiled code shown does not have line correspondence with the source. The cell **a** defined in macro is instantiated three times. Remember that by convention the jump to operand C address in Input/Output instruction is done undonditionally.

Definitions as described in section 6.7 can also be used in macros.

6.17 Autobits

Autobits is a built-in directive which generates code based on binary expansion of a number. The best explanation of how it works is demonstrated by the following example.

```
.pragma io=a                        |
._autobits '5' bit0 bit1            |          23 -1 3
Z Z (-1)                            |          22 -1 6
. Z:0 a:'0' b:'1'                   |          23 -1 9
.def bit0: a                        |          22 -1 12
a (-1)                              |          23 -1 15
.end                                |          23 -1 18
.def bit1: b                        |          21 21 -1
b (-1)                              |          0 48 49
.end                                |
```

Directive '**._autobits** ' takes three arguments: the first argument is a number in question; the second argument is a name of a macro which will be placed in code for a bit zero; the thirst argument – for a bit one. In this case the number is 53 (ASCII code for ‘5’). It binary expansion is 110101. The compiler replaces the autobits directive internally with the following code:

```
.bit1
.bit0
.bit1
.bit0
.bit1
.bit1
```

which in turn is substituted by macro instantiations. This program prints ‘101011’, which are the bit in the order lower to higher.

This directive is used in the library for raising a value (possibly encrypted) to a power of a constant.

6.18 Table (low level)

Table is another built-in directive which generates a map or an array of data. Its purpose will be revealed in the following section 6.19. The format of the directive is the following:

_table modifier $g\ p_1\ c_1\ p_2\ c_2$

For example

```
.pragma N=7.11 r=2 k=1              |     #pragma N=77
._table _array_ $X(1814) 1 0 1 0    |     23.42 23.42 0 0
._table _map_ $X(1814) 1 4 1 0      |     41.42:23.42 4:0
```

This directive takes 6 arguments. The first is type modifier. It can be only either `_array_` or `_map_` (underscores added to distinguish these modifiers from variables). Argument *g* is a base number. Its order in *N* defines the size of the table. It has to be selected in particular way to accommodate the requirements to the table size.

The table is generated by the following process.

1. Raise *g* to power *i*, where *i* starts at 1 and ends at *w*, when $g^w = 1 \bmod N$.
2. Get next element by $c_1(g^i)^{p_1}$
3. Get next element by $c_2(g^i)^{p_2}$

The array function generates a linear sequence of numbers and the map function pairs elements to address-value pairs. Values p_1 and p_2 must be defined as open values (no s-part) as their t-values are used to raise to power.

6.19 Confinement

Confinement is a mechanism which replaces function *G*. It based on the idea to restrict possible values of *r*. Directive table defined in general terms, but becomes necessary to automate the process of generation a look up table for all possible *r* values. The implementation of this mechanism is briefly described in section 8.4.

Theory

The implementation of function *G* requires the value $\varphi(k\varphi)_N^{-1}$ to be encoded into a program. This may not always be desirable because this value, if deciphered, unlocks the whole program encryption.

Function *G* is required for multiplication and equal function algorithms. It turns that these algorithms can be implemented without function *G* and without bringing totient function into the program. This is possible if the program knows the the random part of the value. Suppose that

$$x = r^N (1 + Nkm) \mod N^2$$

is raised to power *N*:

$$x^N = \left(r^N (1 + Nkm)\right)^N = r^{N^2} \mod N^2$$

If this value (which now does not depend on *m*) is among a limited number of expected values, then an index table can resolve it to:

$$\begin{aligned} & r^{N^2} \Rightarrow c \\ & c = r^{-N} a \\ & a = (N-1)(2k)_N^{-1} \mod N \end{aligned}$$

The number *a* is judiciously selected for the following reasons:

$$\begin{aligned} cx &= r^{-N} a r^N (1 + Nkm) \mod N^2 = \\ &= a - (2k)_N^{-1} Nkm \mod N^2 = \\ &= \langle m = 2n + b \rangle = \\ &= a - 2_N^{-1} Nb - Nn \mod N^2 = \\ &= a + N\left(\frac{N-1}{2} b - n\right) \mod N^2 \end{aligned}$$

Since $0 < a < N$ then *t* part of *cx* is

$$O_2(cx) = \frac{N-1}{2} b - n = \begin{cases} N - n, & \text{if } b = 0 \\ (N-1)/2 - n, & \text{if } b = 1 \end{cases}$$

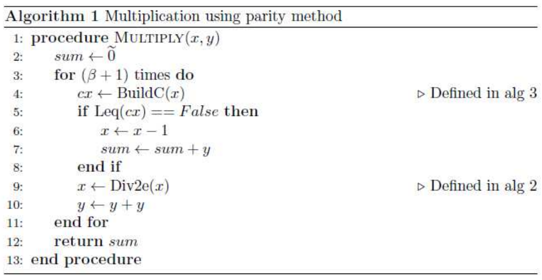
Hence

$$\mathrm{Leq}(cx) = \begin{cases} True, & \text{if } m \text{ is even} \\ False, & \text{if } m \text{ is odd} \end{cases}$$

This equation extracts the lowest bit of the encrypted value. This method can be used in the multiplication algorithm resulting in more efficient procedure. Also it can be used for Equal function. Obviously this method implemented in a Cryptoleq program is a weak spot in security. On the other hand it does not require totient function embedded into the program as function *G*.

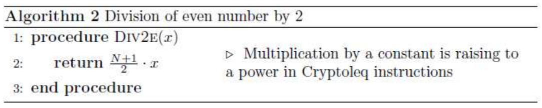
Multiplication
The multiplication algorithm using parity method is straightforward. One argument is divided by 2 while the other is multiplied by 2. Learning parity the division by 2 is possible. The algorithm is presented in Alg 1. It uses a function BuildC (Alg 3) which selects from a predefined table values a and r^{-N}. This, in turn, requires re-normalization of random parts r as described further.

Equal function is also simple: each number is sequentially tested for their bits and the bits are compared across each of the arguments. The algorithm is presented in Alg 4.



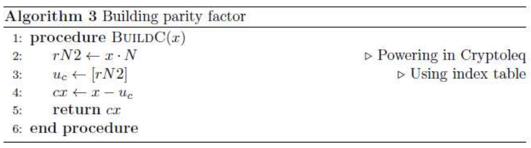
Algorithm 1 Multiplication using parity method

```
procedure MULTIPLY($x, y$)
    $sum \leftarrow \tilde{0}$
    for $(\beta + 1)$ times do
        $cx \leftarrow$ BuildC($x$)                ▷ Defined in alg 3
        if Leq($cx$) $== False$ then
            $x \leftarrow x - 1$
            $sum \leftarrow sum + y$
        end if
        $x \leftarrow$ Div2e($x$)                  ▷ Defined in alg 2
        $y \leftarrow y + y$
    end for
    return $sum$
end procedure
```



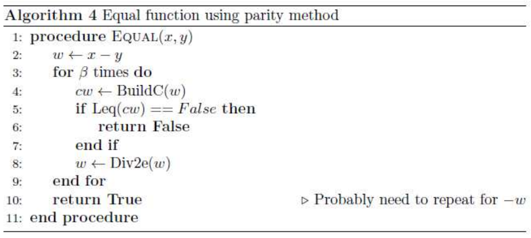
Algorithm 2 Division of even number by 2

```
procedure DIV2E($x$)
    return $\frac{N+1}{2} \cdot x$      ▷ Multiplication by a constant is raising to
                                        a power in Cryptoleq instructions
end procedure
```



Algorithm 3 Building parity factor

```
procedure BUILDC($x$)
    $rN2 \leftarrow x \cdot N$                 ▷ Powering in Cryptoleq
    $u_c \leftarrow [rN2]$                     ▷ Using index table
    $cx \leftarrow x - u_c$
    return $cx$
end procedure
```



Algorithm 4 Equal function using parity method

```
procedure EQUAL($x, y$)
    $w \leftarrow x - y$
    for $\beta$ times do
        $cw \leftarrow$ BuildC($w$)
        if Leq($cw$) $== False$ then
            return False
        end if
        $w \leftarrow$ Div2e($w$)
    end for
    return True                    ▷ Probably need to repeat for $-w$
end procedure
```

Confinement of *r*
The algorithm 3 is using an index table to find value c. Any practical implementation requires that the table is of a reasonable size to fit in memory. It means that its size should not depend on, or rather grow with increasing N. This can actually be achieved by reducing the order of r^N.

Let us assume that φ is factorized into a number of primes (not necessarily unique):

$$\varphi = p_1 p_2 p_3 \ldots p_n$$

A randomly selected number u has some order in N made of subset of the primes

$$\mathrm{ord}_N u = p_i p_j \ldots p_k$$

Any subset of primes taken from this subset, let us say, $w = p_q \ldots p_r$, would make a number

$$a = u^{w^{-1}\operatorname{ord}_N u}$$

of order *w*:

$$a^w = 1 \mod N$$

which means

$$a^{wN} = 1 \mod N^2$$

The random part of the encryption selected according to definition of *a* above:

$$r = a^i \mod N$$

where *i* is a random number, has always *w* distinct values.

Since *w* can be selected small regardless of how big *N* is, then the size of look up table for r^{N^2} is limited by *w*.

6.20 Table (high level)

The library defines macro **`table`** (see section **Error! Reference source not found.**). Macro **`table`** takes 2 arguments[2]. The first argument is one of the possible r^N values. The second is c_1. The following example multiplies encrypted 17 by 100 and outputs decrypted result (if sneak value is 1).

```
.pragma N=107.101 r=5137 k=5 io=ts cqtype=ts incdir=../lib
10:
.cmul   x y z
.outd   z
.halt
. x:~100 y:~17 z:0
.include "confy.lib"
.include "secure.lib"
.table $peekrnd 0
```

Build-in function (see section 6.21) **`$peekrnd`** returns the current r^N which is **`5137`**^{N}. The value 5137 has been selected to make the table of size 5. How?

Consider the following script of bignum calculator (see section 11.2, it can be reproduced when running ceal with '-m' option):

```
N = 107,101;
lam = lcm N[0]-1 N[1]-1;
modulus = N[0]*N[1];
w=lam/5;

prn;
```

[2] At this time of writing (Dec 2015) c_1 multiplication is not implemented and has to be 0, which means that a program using table must start at non-zero address.

```
"N = ". N[0] . "x" . N[1];           |     N = 107x101
"lam = ".lam;                        |     lam = 5300
factor lam/2;                        |     2
factor lam/4;                        |     5
factor lam/20;                       |     5
factor lam/100;                      |     53
"N = ". modulus;                     |     N = 10807
"w = ". w;                           |     w = 1060
"r = ".2^w;                          |     r = 5137
"ord r = " . ord 2^w;                |     ord r = 5
```

Variable **`lam`** holds the calculated Carmichael number. Using function **`factor`**, which finds the lowest factor, **`lam`** is tested for appropriate value. If the size of the table is desired to be around 5, then 5 is the appropriate number since it is a factor of **`lam`**. So *w* is set to **`lam/5`**. Next, the base is selected. In the above example 2 is used, although any number suffices. *r* becomes 2^w, and finally order of *r* is tested to be exactly as expected. If the order is less (which sometimes naturally happens), a different base must be selected, e.g. 3; and so on.

6.21 Built-in functions

CEAL has special built-in functions. The syntax is dollar sign, name, and optional parentheses or parentheses with arguments. The list of function is presented in section 2.3. Function '$beta()' returns value β of equation 12 or defined value if14 β has been redefined by pragma option. Function '$B2()' returns 2^{β}. In the program

```
.pragma PQ=7.11
a (-1)
b (-1) (-1)
. [c] [$B2()]
. [d] [1+$B2+$beta]
. a:c b:d
```

printing "8 12", **`c`** and **`d`** are defined with sizes of the data fields which are defined by the array declared. This example shows that the functions return values which are considered to be constant, so the result of expression using constants is also a constant. The usage of labels in expressions defining the array size is forbidden because values of labels resolved after expansion of the arrays.

Function "$unit()" returns the value calculated by equation 2:

```
.pragma r=2 PQ=7.11                  |
A+$unit(A) (-1)                      |     75.15 76 3
A (-1) (-1)                          |     74.15 76 76
{ A:$unit(A) $unit(0) }              |     74.15:53 1
```

This program prints "1 53": first the cell addressed '**`75.15`**', then the one before it with the value $53 = (1+15)_{77}^{-1}$.

Functions can also be used in definition expressions:

```
.pragma r=2 N=5.5                    |
b=$random()                          |
b:b (-1)                             |     4.23:4.23 24 7.23
b b (-1)                             |     4.23 4.23 24
```

There are a few things about this program printing “4.23”. Variable **`b`** is assigned a value via definition. This value then is used as a default initial address of the program (entry point), and the value of the first cell. *N* has a strange value 25. The first instruction prints value **`b`** and the second clears the cell **`b`** and halts. The jump address of the first instruction is 7.23 which is (4+3).23.

One limitation of current version of the compiler is that tilde sign '**`~`**' is an operator working on cell or data field definitions, and is not a part of the expression syntax. This means that it cannot be used inside expressions: for example, '**`r=~3`**' is an invalid; instead a function **`enc`** should be used: '**`r=$enc(3)`**'. This function returns an encryption of its argument. This limitation is likely to be fixed in the future.

6.22 Expressions in array definitions

Sometimes it is desirable to define an array with a predefined variable. The name cannot be used directly because it introduces the size of the data field. In this case the entity inside the square brackets should be an expression, not a name. This can be done by adding, for example '**`+0`**' or just putting the name inside the parentheses. This program

```
.abc 3                        |           6 -1 7
.abc 5                        |           0 0 0
0 0 (-1)                      |           3
.def abc x                    |           15 -1 16
p (-1) e                      |           0 0 0 0 0
. [z] [(x)]                   |           5
. p:z                         |           0 0 -1
e:;.end                       |
```

prints “3 5” that are sizes of the data fields initialized inside the macro instantiations. Note that the arguments passed to the macro are constant expressions, i.e. labels cannot be used there. Similar with definitions: compare

```
.a; .def a; y=1; . [(y)] 0; .end
```

compiles into '**`0 0`**', and

```
.a; .def a; y=1; . [y] 0; .end
```

compiles into '**`0`**', because in this case **`y`** does not define an array, but introduces a variable and gets the size of the data field which is 1.

6.23 Using the library

The library described in section 8 defines a number of useful macros. It also implements multiplication of open and encrypted values. This program

```
.pragma PQ=29.101 k=5 r=17
.smul   A B C
.outd   A
.outd   B
.outd   C
.halt
. A:~10 B:~25 C:0
.include "secure.lib"
```

using the library compiles into Cryptoleq code of 2214 cells. It multiplies two encrypted values using the command '**.smul**'. It prints “10 25 250” which are decrypted values of two multiplicands and their product.

A few other examples using the library are shown in section 5.4.

7. Macros

Macros are language constructions which represent a piece of code to be replaced at the point of macro usage. Macros have unique names. Definition syntax

```
.def name [ parameters [: globals ]]
body
.end
```

Usage syntax:

```
.name [ arguments ]
```

Parameters is a list of space separated names used inside the body of the macro. *Arguments* is a list of space separated expressions. The arguments and the parameters are bound during macro instantiation. If a parameter used as a constant inside macro, then the corresponding argument must be a constant expression. *Globals* is a list of space separated names which do not participate in local scope resolution during macro instantiation. The body can define variables which stay locally defined inside.

8. The Library FIXME confy

The library is a collection of files written in CEAL and defining functions and some useful macro commands. Several algorithms are implemented as functions: a macro usage employs the code to jump to a particular place in the program preliminarily preparing the address to jump back. Without a stack mechanism functions have to store their arguments and the return address in some static places and forbid recursion. The current library implementation consists of the following files:

Name	Description
general.lib	General macro utilities
o_const.lib	Constants related to open multiplication
o_d2.lib	Division by 2 for open values
o_d2p2.lib	Division by 2 of power of 2 for open values
o_mul.lib	Multiplication of open values
open.lib	Meta include for open modules
s_const.lib	Constants for encrypted multiplication
s_d2.lib	Division by 2 for encrypted values
s_d22t.lib	Division by 2 of power of 2 for encrypted values
s_eq.lib	Equal function for encrypted values
s_gfun.lib	Function *G*
s_mp1.lib	Decryption operation used in function *G*
s_mul.lib	Multiplication algorithm for encrypted values
s_rand.lib	Encrypted random generator
secure.lib	Meta include for encrypted and open modules
confy.lib	Meta include for confinement modules
c_build.lib	Building parity function BuildC

`c_div2e.lib`	Division of even number by 2
`c_eq.lib`	Equal function using parity method
`c_mul.lib`	Multiplication using parity method
`c_power.lib`	Raising to power
`c_table.lib`	High level table macro definition

The following description briefly touches some particulars of the library implementation: most interesting or most convoluted. For more comprehensive insight see the code of the library.

8.1 Module general.lib

The collection of utilities in the file **`general.lib`** is the most interesting. It is worthwhile to go through some of its definitions.

```
. Z:0 _T:0 _o1:1 _om1:-1
```

By convention cell **`Z`** is a special register used for immediate operations. It is assumed to be always zero before usage and cleaned up straight after the usage. This is the only library name used without underscore – all other library names use underscore prefix. Cell **`_T`** is for temporary usage, with no clean up assumption. The other two are just constants.

The basic operation is to move value from one cell to another:

```
.def mov x y
.clear  y
.add    x y
.end
```

It is defined via other two macros. Let's see.

```
.def clear x
        x
.end
```

is the simplest macro definition. And

```
.def add x y : Z
        x Z; Z y; Z
.end
```

uses **`Z`** as an intermediate cell to construct addition operation out of subtraction.

Goto, out, and halt are other simple but commonly used macros:

```
.def goto x : Z
        Z Z x
.end

.def out x
        x (-1)
.end

.def halt : Z
        Z Z (-1)
.end
```

Increment and decrement of open values use predefined constants:

```
.def inc x : _om1
        _om1 x
.end

.def dec x : _o1
        _o1 x
.end
```

In Cryptoleq dereferencing involves code self-modification. For example, macro **`mov`** (see above) moves the value pointed by the first argument to the cell pointed by the second argument. If one needs to dereference the first argument, i.e. move the value of the cell pointed by a value of the cell pointed by the first argument to the cell pointed by the second argument, the macro **`mov21`** can be used:

```
.def mov21 x y : Z
.clear  y
.clear  z
.mov    x z
        z:0 Z; Z y; Z
.end
```

Here, the value is first moved to the cell which is an operand of the next instruction, and then this value is used as an address when that instruction is executed.

A macro checking a particular condition can also be constructed. The macro '**`ifneq`**' takes three arguments: two values and an address. It compares the values and if they are not equal, execution is passed to that address:

```
.def ifneq A B C
.mov    A a
        B a t
.goto   C
        t:a
.add    B a
        A a end
.goto   C
. a:0
end:
.end
```

A more complex but a useful macro is a function call:

```
.def call start return : Z
        ?+8 Z; Z return; Z Z start; .?; return
.end
```

The first two instructions copy the value of the cell '**`.?`**' to where **`return`** point to. The next instruction cleans up **`Z`** and jumps to where **`start`** points to. The cell '**`.?`**' holds the address of the next cell which is the instruction cleaning up the cell where **`return`** point to. The last instruction '**`return`**' cleans up that cell making it ready for the next time use. In this example

```
.call F E
...
# somewhere
F:
# some code
Z Z E:0
```

the usage of this macro makes execution jump to `F`, execute “some code”, and then jump back and continue execution after '`.call`' line.

In reality a call to functions is a bit more complex because it usually involves passing arguments and returning the function return value, but not by much.

There are a few other macros defined in this module: `minswp` – sort two open values (swap if the second is greater); `outd` – output decrypted value (using decrypting function from secure module); and `abs` – take absolute value of a number.

8.2 Module open.lib

The content of `open.lib` is

```
.include "general.lib"
.include "o_d2p2.lib"
.include "o_const.lib"
.include "o_d2.lib"
.include "o_mul.lib"
```

Module `general.lib` is described in the previous section 8.1. The other files contain the implementation of the three-level multiplication algorithm for open values. The program

```
.pragma N=512
.omul R S U
.out R
.out S
.out U
.halt
. R:13 S:7 U:0
.include "open.lib"
```

uses `omul` macro to multiply two numbers. It prints “13 7 91”.

8.3 Module secure.lib

This module is also a list of includes:

```
.include "open.lib"
.include "s_mp1.lib"
.include "s_gfun.lib"
.include "s_rand.lib"
.include "s_d22t.lib"
.include "s_const.lib"
.include "s_d2.lib"
.include "s_mul.lib"
.include "s_eq.lib"
```

where each “s” file implements a particular algorithm. File '**`s_const.lib`**' defines encrypted constants required for multiplication algorithm:

```
~. _sB2:$B2() _s0:0 _s1:1 _sm1:-1
```

File '**`s_mul.lib`** ' is a top level function of the multiplication algorithm for encrypted values. Below is the text of this file to demonstrate a working example of a function implementation.

```
# z=x*y

# Macro to call function
.def smul x y z : _sm_ret _sm_start _sm_x _sm_y _sm_z
.mov    x _sm_x
.mov    y _sm_y
.call _sm_start _sm_ret
.mov    _sm_z z
.end

# Data members
. _sm_z:0 _sm_x:0 _sm_y:0

# Implementation
.__sm_f

# Definition
.def  __sm_f  :  _sm_z  _sm_x  _sm_y  Z  _sm_ret  _sm_start  _s0  _o1
_beta1
x=_sm_x
y=_sm_y
sum=_sm_z
. w:0 b:0 i:0
_sm_start:
.mov    _s0 sum
.mov    _beta1 i
.inc    i
loop:   _o1 i return
._sd2   x w
.mov    x b; w b; w b
._G     b y b
.add    b sum
.add    y y
.mov    w x
.goto   loop
return:
Z Z _sm_ret:0
.end
```

The first part is a definition of the call macro. It prepares arguments, makes the call and handles the result. Data members part declares globals used in the algorithm. Implementation defined the code – this is done to make the code contained in its own scope. The definition part is the body of the algorithm. It uses other macros: '`_sd2`' and '`_G`', which are defined in '`s_d2.lib`' and '`s_gfun.lib`'. Function '`_sd2`' uses another function '`_sd22t`' which is defined in '`s_d22t.lib`' as a lookup table of power of 2 numbers. The table is initialised on the first call, and later is used to extract values.

```
# # # # # # # # # # # # # # # # # #
# d22t function (table based)
# divides encrypted power of 2 by 2
# input index
# output Encrypted element of
# { 1, 2, ..., (2^beta) }[index]
# # # # # # # # # # # # # # # # # #

.def _sd22t idx res : Z _beta1 _o1 _s1
        Z tbl return
.mov    _beta1 i
.mov    _s1 p2
.mov    ptbl t
loop:
        _o1 i return
.mov    t u2
.mov    t u3
.mov    t u4
        u2:0 u3:0; p2 u4:0;
.inc    t
.add    p2 p2
.goto   loop
. [sz] tbl:1 [$beta-1]
. ptbl:tbl t:0 i:0 p2:0
return:
.mov    ptbl t
.add    idx t
.mov    t u1
        res; u1:0 res
.end
```

The flag `tbl` is used to differentiate the first and the subsequent calls – when the execution immediately jumps to the `return` section.

Function *G* defined in '`s_gfun.lib`' uses decryption function defined in '`s_mp1.lib`' and random generator defined in '`s_rand.lib`'.

Finally the equal function is defined in '`s_eq.lib`'. Its implementation is simple and it is worthwhile to present its text.

```
# smul z=(x==y) #

# Macro to call function
.def seq x y z : _q_ret _q_start _q_x _q_y _q_z
.mov    x _q_x
.mov    y _q_y
.call   _q_start _q_ret
.mov    _q_z z
.end

# Data members
. _q_z:0 _q_x:0 _q_y:0

# Implementation
.__q_f

# Definition
.def __q_f : _q_z _q_x _q_y Z _q_ret _q_start _s1
x=_q_x
y=_q_y
z=_q_z
. u:0 v:0 gu:0 gv:0
_q_start:
.mov    _s1 z
.mov    x u
        y u
        v; u v
._G     u z gu
._G     v z gv
        gu z
        gv z
Z Z _q_ret:0
.end
```

Similar to the multiplication code it consists of a macro definition, globals, an implementation declaration, and the body definition. It uses function *G* two times as described in equation 3.

8.4 Module c_table.lib

This very short module defines high level macro **`table`**.

```
.def table gN c1 : _map_
._table _map_ gN (-$phi) c1 $phi-1 $X($invN(-$k-$k))
.end
```

It takes a generator **`gN`** , already raised to *N*, and c_1 parameter. The generation of table stops for the smallest *i* when

$$\left(g^N\right)^i = g^N \quad \bmod N^2$$

The other arguments are selected so *c* from section 6.19 is:

$$c = r^{-N}(N-1)(2k)_N^{-1} \quad \bmod N^2$$

because

$$\left(g^{N}\right)^{p_1} = g^{N(-\varphi)_N} = g^{N(N-\varphi)} = g^{N^2-N\varphi} = g^{N^2} \quad \bmod N^2$$

and

$$\left(g^{N}\right)^{p_2} = g^{N(\varphi-1)} = g^{N\varphi-N} = g^{-N} \quad \bmod N^2$$

with $c_2 = (-2k)_N^{-1}$.

9. Some elements of Grammar

9.1 Escape sequences

Escape sequences are similar to standard C list:

Esc	Hex	Comment
\a	07	Beep
\b	08	Backspace
\f	0C	Formfeed
\n	0A	New line
\r	0D	Return
\t	09	Tab

Esc	Hex	Comment
\v	0B	Vert
\\\\	5C	\\
\\'	27	'
\\"	22	"
\\?	3F	?
\xhh	hh	Hex value

9.2 Empty instructions

Empty instructions are allowed. This text

```
;;;
.

{}
. 0
```

compiles into one cell '**0**'.

9.3 Syntax: Backus–Naur Forms

Backus–Naur Form (BNF) presented below is an older version of BNF. The syntax of the current CEAL compiler is slightly enhanced. For exact syntax definition see “parser.cpp”.

```
INPUT GRAMMAR
program := {statement}
statement := instruction | macrodef | macrouse
macrouse := [label:]+ .id {expression}
macrodef := .def id {id} [ : {id} ] eos {statement} .end eos
instruction := [ [~]. ] {l-items} eos
instruction := [~]{{ {l-items} }}
l-item  := ( [label:]+ item | id = expression )
item    := ( [~]expression | [[unumber]] | [[id]]
        | [[(const-term)]] | [~]"string" )
label   := id | tsnumber
expression := ( term | term+expression | term-expression )
term    := ( -term | (expression) | const | id | func )
const   := ( tsnumber | 'letter' | ? )
eos     := ';' | EOL
tsnumber  := t-number[.s-number] | ([r-number],m-number)
t,s,r,m-number are unumber

const-term := term which can be evaluated at parsing time
func := $id(args)
args := expression[,expression]+

TOKENS := . ~ ; [ ] ID NUMBER + - ( ) 'char' "STR" ? EOF EOL = { } : , $
```

```
COMMENTS := # <...> EOL

directives := .pragma .include
$functions := peekrnd, random, B2, beta, X, unit ...

OUTPUT GRAMMAR
instruction := [tsnumber(address):] ( LOLI(.) | LOLI({}) | LOLI(3) )
LOLI==list of l-items
l-item  := tsnumber(value)

String and char literal escapes: \a \b \f \n \r \t \v \\ \' \" \? \xhh
```

10. CEAL Source

10.1 Compiler and Current State

CEAL compiler source is written in C++. It is well structured but not documented. It can survive a fair amount of abuse before it becomes unmaintainable.

The compiler C++ code is developed focused on simplicity for feature development and expanding the language in exchange for efficiency during compilation. In other words this CEAL compiler is not fast (when compiling only, the quality of the output code produced is not affected), but its source is conceptually simple and allows easy expanding syntax of CEAL.

Current version (10 Sept 2015): v0.7.1460.

10.2 Bugs

Severity	Manifestation	List
Super Critical	Hangs or incorrect result on valid program	None known
Critical	Hangs on invalid program	None known
Severe	Produces unexpected result on invalid program	None known
	Internal error on invalid program	None known
Mild	Does not accept valid syntax	None known

10.3 Possibly future improvements

- Add negative tests to autobits
- Implement division and decimal printout

11. Additional topics

11.1 C++ prototype classes and PIR example: cpppir

C++ PIR prototype has been written before CEAL compiler as a proof of concept of working mathematics and algorithms. It implements three C++ classes:

- secure_int
- open_int
- secure_ptr

with C++ operators defined via Cryptoleq instruction. The subtraction operator '**`secure_int::operator-=(B,A)`**' is defined as '**`A.invert*B`**', where A and B are operands of Cryptoleq instruction represented as a C++ Cell class which in turn defines multiplication and inversion operations on its values. The algorithms of multiplication is generally follows the description of section 4.2. Function G is implemented directly with access to the value $\varphi(k\varphi)_N^{-1}$. The source code consists of a PIR program (**`prog_sec`**) and an encryption utility (**`crypter`**) to prepare the encrypted input and decrypt the output of the PIR program. Note, that the CEAL compiler embeds **crypter** functionality by the “crypter” command line option. The script running the PIR example is:

```
crypter enc db.opn db.sec
crypter enc input.opn input.sec
prog_sec
crypter dec output.sec output.opn
```

Files `db.opn` and `input.opn` are unencrypted database and the input.

11.2 Bignum calculator

The bignum calculator tool is an interactive calculator. It is embedded into CEAL compiler with a command line option '`-m`'. It can do calculations in modular arithmetic and factorize numbers. For example:

```
> print factors 1000000000000000000000000000000000001;
73,137,3169,98641,99990001,3199044596370769
```

The grammar is very simple: all values are presented as arrays with functions operating on them. The simplest way to learn how the calculator works is to go through an example:

Code	Comments	Output
`# comment`	This is a comment	
`define p print;`	Introduce a synonym: so 'p' is now the same as 'print'	

`N=91;`
`pq = factors N;`
	Function 'factors' returns a list of factors	

`assert size pq 2;`	Function size takes only 1 argument and returns the size of the array. Function assert takes 2 arguments and produces an error if they are different.	

`N2 = N^2;`
`phi = (pq[0]-1)*(pq[1]-1);`
	'^' is raising to a power and '[]' is accessing the elements of an array	

`p ("N^2 = ".N*N), (" phi=".phi);`	Print values. Comma ', ' is used to create a list, so print function takes an array of 2 elements. Dot '. ' is a string concatenation. Numbers are converted to strings if used in a string operation	`N^2 = 8281, phi=72`

`r1 = 2; k=3;`
`g = 1+k*N;`
`m = 4;`		

`p ("r1=".r1), (" k=".k),`
`(" g=".g), (" m=".m);`
		`r1=2, k=3, g=274, m=4`

`modulus=N2;`	'modulus' is a special variable. When it is set, the majority of operations is done in modular arithmetic	

`r1 = r1^N;`
`A=r1*(g^m);`
`p "A=".A;`
		
		`A=8045`

`p "A^phi==1+N*m*k*phi ".`
`assert A1=A^phi 1+N*m*k*phi;`
	Assert, as any other function, also returns a value (if arguments do not match, it aborts the execution)	`A^phi==1+N*m*k*phi 4096`

`p phim1 = 1/phi;`		`8166`

`p "A1^phim1==1+N*m*k " .`
`assert A2=A1^phim1 1+N*m*k;`
		`A1^phim1==1+N*m*k 1093`

`p km1 = 1/k;`
`p "A2^km1==1+N*m " .`
`assert A2^km1 1+N*m;`
`p (A^phi^phim1^km1-1)/N;`
		`5521`
		`A2^km1==1+N*m 365`
		
		`4`

`exit;`	Exit program; 'quit' or 'q' would work too	

Some useful syntax constructions are shown in another demo example

Code	Comments	Output

define p print;
define x;
x x x x x x x x;
	'p'is 'print'; 'x' is nothing	

define phello print "hello";
phello;
	'phello' is replaced with whatever
	It is defined with
		hello

p N=91[-0]+0;
p pq = factors N;
	'[]' index operator can be applied
	to a scalar
		91
		7,13

print "hello" . " ". "yello";	Concatenating strings	**hello yello**
print 1+3+"aas1"+(1);	Automatic conversion to a number, letters are ignored	**6**
print 1+2*3;	Mathematical order of arithmetic operations: multiplication first	**7**
print print 1234;	'print' is a printing function returning its argument	**1234** **1234**

aaa = 1+5;
aaa -= 2;
print "aaa=".(aaa+=1);
	C-style operations	**aaa=5**

aaa = aaa,aaa+2;
print aaa*5;
	'aaa' becomes a vector of 2:
	(5,7)
		25,35

print aaa[1,0];	Building a vector of out elements	**7,5**
print (aaa,aaa,aaa)+1;	Arithmetic operations can be done over vectors	**6,8,6,8,6,8**
delete aaa;	Removing the name and releasing its associated memory	

modulus=36;
print 38+8;
	Switch to modular arithmetic mode	**10**

print 3.2.1-4;	Concatenation numbers as strings; (-3) mod 36 is = 33	**3233**

a=20;
b=7;
print a/b , a%b;
	Normal division '/' is modular;
	modular division '%' is
	a remainder
		8,6

print (div a b) [1];	Function 'div' calculates both: quotient and remainder	**6**
modulus=0;	Turn off modular arithmetic	
print 1!,2!,3!,4!,5!;	Factorials	**1,2,6,24,120**
print 1/7, 1/3, 1/5;	Integer division, because modular arithmetic is off	**0,0,0**
print (factors 10007*10009) [1];		**10009**
p mod 10 0+mod 100 123*1;	'mod' is congruence: 123->23->3	**3**
p mod 20 16/3;	Operations under 'mod' are modular	**12**

a=10:100;
b=30,50;
p size a;
p index a b;
	Function ':' generates an array;
	function 'index' builds another
	array based on array of indices
		91
		20,40

a=(1,2):(3,4);
p a;
	Array generation by 1:3,1:4,2:3,2:4	**1,2,3,1,2,3,4,2,3,2,3,4**

exit;		

There are also functions '**gcd**' and '**lcm**' implemented taking two arguments each.

Function '**ord**' is brute force finding multiplicative order of its argument.

Function '**factors**' is using Pollard rho algorithm. Function '**factor**' (not in example) finds only one of the factors. This can be useful when full factorization is not required.

'**prn**' enters (and exits) printing mode when every statement's expression is printed.

11.3 Test suit and scop.sh

Shell script **scop.sh** is a script running automatic tests. [If you run it on Windows with ap14 far, **scop_inst.sh** can be used to add a command **scop**.] When running in a particular directory scop recursively finds all files **.sca**, **.sct** and **.sts** and runs them as tests cases. Then it compares the output with the expected results stored in

`.gold` files and then reports if they do not match. It is important to run all tests after any modification of the source. Also it is desirable to test more features of CEAL compiler and add more test cases as the current set of tests is not comprehensive.

References

1. "Cryptoleq: A Heterogeneous Abstract Machine for Encrypted and Unencrypted Computation", 2015