Mastering Cryptol: A Comprehensive Guide
- A Guide for C Programmers -

Ji Yong-Hyeon

Department of Information Security, Cryptology, and Mathematics
College of Science and Technology
Kookmin University

March 8, 2024

Data

Table 1: Hierarchy of Physical and Logical Data Units

Physical Units	Logical Units
Bit (b)	Field (Set of bytes or words)
Nibble (4 bits)	Record (Collection of fields)
Byte (B)	Block (Groups of bytes or words for storage)
Half-word (2 Bytes)	File (Collection of records)
Word (4 Bytes)	Database (Organized collection of files)
Double-word (8 Bytes)	Data Warehouse (Aggregated collection of databases)

Cryptol vs EasyCrypt

- **Cryptol**: Cryptol is a domain-specific language designed specifically for specifying cryptographic algorithms. A creation by Galois, Inc., it's a tool used primarily for creating high-assurance cryptographic software. Cryptol allows developers to write cryptographic algorithms in a way that directly reflects the mathematical specifications, which makes it easier to analyze and verify for correctness and security.

- **EasyCrypt**: On the other hand, EasyCrypt is a toolset designed for the formal verification of cryptographic proofs. It provides a framework for developing and verifying mathematical proofs of the security of cryptographic constructions, such as encryption schemes, signature schemes, and hash functions. EasyCrypt operates at a higher level of abstraction compared to Cryptol and is used for proving the security properties of cryptographic protocols mathematically.

If you're comparing them from a user perspective, Cryptol is more about the implementation and specification of cryptographic algorithms, making sure they are implemented correctly according to their mathematical definitions. EasyCrypt is more about proving the theoretical security properties of cryptographic protocols and systems.

Contents

Chapter 1

A Crash Course in Cryptol

1.1 Basic Data Types

1.1.1 Bit: Booleans

"The most basic data type in Cryptol, representing a single binary digit (0 or 1)."

Basic Bit Operations

> **Data Type: `Bit`**
>
> In Cryptol, a '`Bit`' can either be
>
> '`True`' (equivalent to 1 in C)   or   '`False`' (equivalent to 0 in C).

```
1 a : Bit
2 a = True
3
4 b : Bit
5 b = False
```

Similar to C, Cryptol supports logical operations like **AND**, **OR**, and **NOT**.

```
1  // AND operation (similar to a & b in C)
2  result_and : Bit
3  result_and = a && b
4
5  // OR operation (similar to a | b in C)
6  result_or : Bit
7  result_or = a || b
8
9  // NOT operation (similar to !a in C)
10 result_not : Bit
11 result_not = ~a
```

```
$ Main> result_and
False
$ Main> result_or
True
$ Main> result_not
False
```

Intermediate Bit Operations

Conditional Expressions: Cryptol uses the '**if...then...else**' syntax, which is similar to the ternary operator '**?:**' in C.

```
1  // Conditional expression (similar to a ? b : c in C)
2  result_cond : Bit
3  result_cond = if a then b else ~b
```

XOR Operation: This is commonly used in cryptography for things like one-time pads.

```
1  // XOR operation
2  c : Bit
3  c = True // Assuming some value
4
5  result_xor : Bit
6  result_xor = a ^ c // Similar to a ^ c in C
```

Advanced Cryptographic Applications with Bit

Simple Cryptographic Operation: Here's a very basic example of a cryptographic operation in Cryptol, a one-time pad encryption of a single bit. In practical applications, you'll work with sequences or arrays of bits.

```
1  // One-time pad encryption
2  plaintext_bit : Bit
3  plaintext_bit = True // Your plaintext
4
5  key_bit : Bit
6  key_bit = False // Your secret key
7
8  // XOR for encryption
9  encrypted_bit : Bit
10 encrypted_bit = plaintext_bit ^ key_bit
11
12 // XOR for decryption (should equal plaintext_bit)
13 decrypted_bit : Bit
14 decrypted_bit = encrypted_bit ^ key_bit
```

```
1  module XOR where
2
3  // Defines an XOR function for two bits
4  xor : Bit -> Bit -> Bit
5  xor a b = (a || b) && ~(a && b)
6
7  // Property: XOR is its own inverse
8  property xor_inverse a b = xor (xor a b) b == a
```

```
$ cryptol XOR.cry
version 2.8.0

Loading module Cryptol
Loading module XOR
$ XOR> :prove xor_inverse
Q.E.D.
(Total Elapsed Time: 0.008s, using Z3)
```

1.1.2   Sequences: Homogeneous Collections

"Fixed-length, ordered collections of elements, which can be bits or other types."

Basic Sequence Operations

> **Sequence**
>
> In Cryptol, a **sequence** is defined with square brackets.
>
> $$[, , ,]$$

Remark 1.1. Unlike C, where you typically define the array size and type, Cryptol infers these from the context.

```
1 seq : [4]Bit // A sequence of 4 bits
2 seq = [True, False, True, False] // Equivalent to {1, 0, 1, 0} in C
3 // seq = 0b1010
```

```
$ Main> seq
0xa
```

Accessing Elements: Accessing sequence elements is similar to accessing array elements in C.

```
1 element1 : Bit
2 element1 = seq @ 0 // Accessing the first element, similar to seq[0] in C
3
4 element3 : Bit
5 element3 = seq @ 3
```

```
$ Main> element1
True
$ Main> element3
False
```

Intermediate Sequence Operations

Sequence Concatenation: You can concatenate sequences to form longer sequences.

```
1 seq1 : [3]Bit
2 seq1 = [True, True, False] // Similar to {1, 1, 0} in C
3
4 seq2 : [5]Bit
5 seq2 = [False, True, True, False, True] // Similar to {0, 1, 1, 0, 1} in C
6
7 // Concatenating two sequences 0b 110 01101 = 0b 1100 1101
8 combined_seq : [8]Bit
9 combined_seq = seq1 # seq2
```

```
$ Main> seq1
0x6
$ Main> seq2
0x0d
$ Main> combined_seq
0xcd
```

Advanced Cryptographic Applications with Sequence

Block Operations: In cryptography, you often work with blocks of data. In Cryptol, you can easily split sequences into blocks or combine blocks into a single sequence.

```
1  // Splitting a sequence into blocks
2  long_seq : [16]Bit // Similar to a block of data in cryptography
3  long_seq = 0xAAAA
4
5  blocks : [2][8]Bit // Splitting into two blocks of 8 bits
6  blocks = split long_seq // There's no direct C equivalent
7
8  joined_seq : [16]Bit
9  joined_seq = join blocks
```

```
$ Main> long_seq
0xaaaa
$ Main> blocks
[0xaa, 0xaa]
$ Main> joined_seq
0xaaaa
```

Applying Functions to Sequences: In Cryptol, you can apply functions to entire sequences, which is useful for cryptographic transformations.

```
1  // A simple bitwise NOT operation applied to a sequence
2  negate_seq : [8]Bit -> [8]Bit
3  negate_seq input = ~input // Bitwise NOT each element
4
5  input_seq : [8]Bit
6  input_seq = 0xCD // ~ 0b11001101 = 0b00110010 = 0x32
7
8  negated_seq : [8]Bit
9  negated_seq = negate_seq input_seq
```

```
$ Main> negate_seq
<function>
$ Main> input_seq
0xcd
$ Main> negated_seq
0x32
```

Advanced Cryptographic Patterns: Sequences are used in various cryptographic algorithms, such as block ciphers and hash functions, to handle data in structured formats.

```
1  // Example of XORing two sequences (common in many cryptographic algorithms)
2  xor_sequences : [8]Bit -> [8]Bit -> [8]Bit
3  xor_sequences seq1 seq2 = zipWith (^) seq1 seq2 // Element-wise XOR
4
5  seqA : [8]Bit
6  seqA = 0xAA
7
8  seqB : [8]Bit
9  seqB = 0x55
10
11 //   0b 1010 1010
12 // ^ 0b 0101 0101
13 //   0b 1111 1111
14 result_seq : [8]Bit
15 result_seq = xor_sequences seqA seqB // Result of XORing seqA and seqB
```

```
$ Main> seqA
0xaa
$ Main> seqB
0x55
$ Main> result_seq
0xff
```

Note ('zipWith' in Haskell).

```
1  // [f(x1, y1), f(x2, y2), ..., f(x_n, y_n)]
2  zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
3  zipWith f list1 list2
```

1.1.3 Tuples: Heterogeneous Collections

> "Collections of possibly different types grouped together."

Basic Tuple Operations

> **Tuple**
>
> A **tuple** can hold a fixed number of elements, which can be of different types.

Remark 1.2. Unlike C, where you define a struct with named fields, Cryptol tuples are indexed by position.

```
1  // A tuple containing a Bit, a 3-Bit sequence, and a 4-Bit sequence
2  myTuple : (Bit, [3]Bit, [4]Bit)
3  myTuple = (True, [False, True, False], [True, True, False, False])
```

```
$ Main> myTuple
(True, 0x2, 0xc)
```

Accessing Tuple Elements: Elements in a tuple are accessed by their positions, starting from zero.

```
1  first_element : Bit
2  first_element = myTuple.0 // Accessing the first element of the tuple
3
4  second_element : [3]Bit
5  second_element = myTuple.1 // Accessing the second element of the tuple
```

```
$ Main> first_element
True
$ Main> second_element
0x2
```

Updating Tuple Elements: Unlike in C, where you might update an element directly, in Cryptol, you usually create a new tuple with the updated value due to its immutable nature.

```
1  // Updating the second element
2  updatedTuple : (Bit, [3]Bit, [4]Bit)
3  updatedTuple = (myTuple.0, [True, True, True], myTuple.2)
```

```
$ Main> updatedTuple
(True, 0x7, 0xc)
```

Intermediate Tuple Operations

Tuple Decomposition: You can decompose a tuple into its components, similar to unpacking a struct in C.

```
1  // Decomposing the tuple into variables
2  (bitVal, threeBitSeq, fourBitSeq) = myTuple
```

```
$ Main> bitVal
True
$ Main> threeBitSeq
0x2
$ Main> fourBitSeq
0xc
```

Nested Tuples: Tuples can contain other tuples, providing a way to structure complex data.

```
1  // A tuple where the first element is also a tuple
2  nestedTuple : ((Bit, Bit), [4]Bit)
3  nestedTuple = ((True, False), [True, True, False, False])
```

```
$ Main> nestedTuple
((True, False), 0xc)
```

Advanced Cryptographic Applications with Tuple

Using Tuples in Cryptographic Algorithms: Tuples can represent structured data in cryptographic algorithms, like keys and states.

```
1  // An example with a simple encryption scheme
2  type Key = [8]Bit
3  type State = [8]Bit
4  type CipherText = [8]Bit
5
6  encrypt : (Key, State) -> CipherText
7  encrypt (key, state) = zipWith (^) key state // A simple XOR-based encryption
8
9  property EncryptDecrypt =
10   \(key : Key) (state : State) ->
11     (encrypt(key, encrypt(key, state))) == state
```

```
$ Main> :prove
:prove EncryptDecrypt
         Q.E.D.
(Total Elapsed Time: 0.012s, using Z3)
```

Advanced Data Structuring: In more complex cryptographic algorithms, you might use tuples to represent different parts of the algorithm's state, input, or output.

```
1  // An example using tuples for algorithm state management
2  type RoundKey = [8]Bit;
3  type RoundState = ([8]Bit, [8]Bit); // Tuple representing state for two rounds
4
5  // Update both states
6  updateState : (RoundKey, RoundState) -> RoundState;
7  updateState (key, (state1, state2)) =
8    (zipWith (^) key state1, zipWith (^) key state2);
9
10 property DoubleUpdate =
11   \(key : RoundKey) (initialState : RoundState) ->
12     updateState(key, updateState(key, initialState)) == initialState
```

```
$ Main> :prove
:prove DoubleUpdate
         Q.E.D.
(Total Elapsed Time: 0.013s, using Z3)
```

1.1.4   Records: Named Collections

"Similar to tuples, but each element is identified by a name."

Basic Record Operations

> **Record**
>
> In Cryptol, you define a record with braces '{}' and specify the names and types of its fields.

```
1  type MyRecord = {
2    field1 : Bit,
3    field2 : [3]Bit,
4    field3 : [4]Bit
5  }
6
7  myRecord : MyRecord
8  myRecord = {
9    field1 = True,
10   field2 = [False, True, False],
11   field3 = [True, True, False, False]
12 }
13
14 first_field : Bit
15 first_field = myRecord.field1 // Accessing the first field of the record
16
17 second_field : [3]Bit
18 second_field = myRecord.field2 // Accessing the second field of the record
```

```
$ Main> myRecord
{field1 = True, field2 = 0x2, field3 = 0xc}
$ Main> first_field
True
$ Main> second_field
0x2
```

Intermediate Record Operations

Nested Records: Records can contain other records, which allows you to structure complex data hierarchically.

```
1  type NestedRecord = {nested : MyRecord, anotherField : [5]Bit}
2  nestedRecord : NestedRecord
3  nestedRecord = {
4    nested = {
5      field1 = False,
6      field2 = [True, False, True],
7      field3 = [False, False, False, True]
8    },
9    anotherField = [True, True, True, False, False]
10 }
```

```
$ Main> nestedRecord
{nested = {field1 = False, field2 = 0x5, field3 = 0x1},
       anotherField = 0x1c}
```

Advanced Cryptographic Applications with Record

Cryptographic Keys and Configurations: Records are excellent for representing complex keys and configuration options in cryptographic algorithms.

```
1  // A record type for encryption configuration
2  type CryptoConfig = {key : [128], iv : [128]}
3
4  // An example configuration using all zeros (False)
5  defaultConfig : CryptoConfig
6  defaultConfig = {key = zero, iv = ~zero}
```

```
$ Main> defaultConfig
{key = 0x00000000000000000000000000000000,
 iv = 0xffffffffffffffffffffffffffffffff}
```

State Management in Cryptographic Algorithms: Use records to manage the state of an algorithm, making it easier to handle multiple pieces of related data.

```
1  type AlgorithmState = {counter : [32], buffer : [64], flags : [8]} // A record
       type for the algorithm state
2
3  initialState : AlgorithmState
4  initialState = {counter = zero, buffer = zero, flags = zero} // Initializes all to
       zero
5
6  updateState : AlgorithmState -> AlgorithmState
7  updateState state =  { state |
8    counter = state.counter + 1,
9    buffer = ~state.buffer,
10   flags = zipWith (^) state.flags 0xf0
11 }
```

```
1  property CounterIncrement state =
2    (updateState state).counter == state.counter + 1
3
4  property BufferToggle state =
5    (updateState (updateState state)).buffer == state.buffer
6
7  property FlagsXOR state =
8    (updateState state).flags == zipWith (^) state.flags 0xf0
```

```
$ Main> :prove
:prove CounterIncrement
Q.E.D.
(Total Elapsed Time: 0.013s, using Z3)
:prove BufferToggle
Q.E.D.
(Total Elapsed Time: 0.008s, using Z3)
:prove FlagsXOR
Q.E.D.
(Total Elapsed Time: 0.009s, using Z3)
```

1.1.5   Function

"Types representing functions from inputs to outputs."

Basic Function Usage

Simple Function: Let's start with a basic function that adds two numbers. In Cryptol, the syntax for function definition is concise.

```
1 add : Integer -> Integer -> Integer;
2 add x y = x + y;
```

```
$ Main> add 4294967296 4294967296
8589934592
```

Function with Bitwise Operation: Functions in Cryptol commonly operate on bits and sequences of bits. Here's a function that performs a bitwise AND on two 8-bit sequences.

```
1 andBits : [8] -> [8] -> [8];
2 andBits a b = zipWith (&&) a b;
```

```
$ Main> andBits 0x0f 0xf0
0x00
```

Intermediate Function Usage

Functions with Conditional Logic: Cryptol functions can implement more complex logic, including conditionals.

```
1 max : Integer -> Integer -> Integer
2 max x y = if x > y then x else y
```

```
$ Main> max 51 94
94
```

```
1 property max_ge_x (x : Integer, y : Integer) = max x y >= x
2
3 property max_ge_y (x : Integer, y : Integer) = max x y >= y
4
5 property max_is_either (x : Integer, y : Integer) =
6   (max x y == x) || (max x y == y)
```

Recursive Functions: Cryptol supports recursive functions, which can be powerful for certain types of algorithms.

```
1 factorial : Integer -> Integer;
2 factorial n = if n == 0 then 1 else n * factorial (n - 1);
```

```
Main> factorial 10
3628800
```

Advanced Cryptographic Function Usage

Encrypting with XOR (One-Time Pad): Here's how you might define a simple encryption function using XOR, which is a foundational operation in many cryptographic systems..

```
1 xorEncrypt : [8] -> [8] -> [8];
2 xorEncrypt key plaintext = zipWith (^) plaintext key;
3
4 property Reversibility (key : [8], plaintext : [8]) =
5   xorEncrypt key (xorEncrypt key plaintext) == plaintext
```

```
$ Main> :prove
:prove Reversibility
          Q.E.D.
(Total Elapsed Time: 0.012s, using Z3)
```

1.1.6 Type Variables

Basic Usage of Type Variables

Generic Identity Function: A simple example of a type variable in use is the identity function, which returns whatever value is passed to it, regardless of type.

```
1 id : {a} a -> a;
2 id x = x;
```

```
$ Main> id 0xff
0xff
$ Main> id 0xffff
0xffff
```

Generic Swap Function: This function swaps the elements of a 2-tuple, and works for any types contained in the tuple.

```
1 swap : {a, b} (a, b) -> (b, a)
2 swap (x, y) = (y, x)
3
4 mixedPair : (Integer, [8])
5 mixedPair = (42, 0xaa)
6
7 swappedMixedPair : ([8], Integer)
8 swappedMixedPair = swap mixedPair
```

```
$ Main> mixedPair
(42, 0xaa)
$ Main> swappedMixedPairnewp
(0xaa, 42)
```

Intermediate Usage of Type Variables

Generic Arrays: You can define functions that operate on arrays of any type. This is useful for functions that perform operations like reversing an array.

```
id : {a} a -> a
id x = x
```

```
$ Main> id 0xff
0xff
$ Main> id 0xffff
0xffff
```

Chapter 2

Functional Cryptography and Software Verification with Cryptol

2.1 Introduction to Cryptol

Cryptol is a purely functional programming language for cryptographic specifications. It's got a large amount of Haskell influence.

- Very syntactically similar to Haskell

- And a similar type system (with some additions, and reductions)

- Purely functional

- Clearly the second best programming language by these metrics alone.

Some additions.

- Type level arithmetic, and a solver for it

- Verification technology

- A smattering of other small scoping

Some reductions.

- It has type classes, but you can't write them.

- No built in IO or any other operations.

- No user-defined data type or "pattern matching" beyond the built-in sequence type.

C:

```
1 uint32_t f(uint32_t x, uint32_t y) {
2     return x + y;
3 }
```

Haskell:

```
1 f :: Word32 -> Word32 -> Word32
2 f x y = x + y
```

Cryptol:

```
1 f : [32] -> [32] -> [32]
2 f x y = x + y
```

Large or "weird" numbers.

```
1 f : [17] -> [17]
2 f x = -x
3
4 g : [23424] -> [548683776]
5 g x = x * x
```

Sequence are fundamental

```
1 [0 .. 3] = [0, 1, 2, 3]
2 f : { x } x -> [4]x
3 f x = [x, x, x, x]
4 // Define a sequence of 5 integers
5 sequenceOfInts : [5]Integer
6 sequenceOfInts = [1, 2, 3, 4, 5]
7
8 // Define a sequence of bits (binary sequence)
9 sequenceOfBits : [8]Bit
10 sequenceOfBits = [1, 0, 1, 1, 0, 0, 1, 0]
11
12 // Define a nested sequence (matrix of bits)
13 matrixOfBits : [3][4]Bit
14 matrixOfBits = [[1, 0, 1, 0], [0, 1, 0, 1], [1, 1, 1, 1]]
15
16 // Accessing elements and slices
17 firstElement = sequenceOfInts @ 0      // Access the first element (index starts
      from 0)
18 firstTwoBits = sequenceOfBits @ [0..1] // Access the first two bits
19 secondRow = matrixOfBits @ 1           // Access the second row of the matrix
20
21 // Sequence operations
22 reversedSequence = reverse sequenceOfInts    // Reverse the sequence
23 concatenatedSequence = sequenceOfInts # [6] // Concatenate [6] to the end of the
      sequence
```

```
1  // Define a sequence of 16 bits
2  myBits : [16]Bit
3  myBits = [1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1]
4
5  // Split the sequence into chunks of 4 bits
6  splitChunks : [4][4]Bit
7  splitChunks = split(myBits)
8
9  // Define a list of sequences
10 myChunks : [4][4]Bit
11 myChunks = [[1, 0, 1, 1], [0, 0, 1, 0], [1, 1, 1, 0], [1, 0, 0, 1]]
12
13 // Join the chunks into a single sequence
14 joinedSequence : [16]Bit
15 joinedSequence = join(myChunks)
```

2.2 Split and Join

```
1  // Pseudo-C code representing what 'g' might look like in C
2  void g(type *z, type (*result)[a], int a) {
3      // Assuming 'z' is an array of length 2*a and 'type' is
           whatever 'b' represents
4      // 'result' is an array of two elements, each of which can
           hold 'a' items of 'type'
5      for (int i = 0; i < a; i++) {
6          result[0][i] = z[i];      // Copy the first half of 'z'
               into 'result[0]'
7          result[1][i] = z[i + a];  // Copy the second half of 'z'
               into 'result[1]'
8      }
9  }
```

In this pseudo-C version, g takes an array z and splits it into two halves, storing these in result[0] and result[1]. This is analogous to the Cryptol version where split takes z and returns two sequences, y and x, which get assigned in reverse order due to the pattern y, x in the Cryptol code.

```
// Split into four bytes (8-bit)
split 0xAABBCCDD : [4][8] = 0xAA, 0xBB, 0xCC, 0xDD
// Concatenation
join [0xAA, 0xBB, 0xCC, 0xDD] = 0xAABBCCDD
```

```
1  split : { parts, each, a}
2      (fin each
3      ) => [parts * each]a -> [parts][each]a
4
5  g: (a, b) (fin a) => [2 * a]b -> [2][a]b
6  g z = {x, y}
7    where {y, x} = split z
```

2.3 Verification

The Quarter Round - C99

```
static void
qround(uint32_t* x, // Pointer to an array
       uint32_t a, // Index of 'x', representing 'a'
       uint32_t b, // Second index
       uint32_t c, // ...
       uint32_t d, // ...
) {
    /* L32 is left 32-bit ROLL, not a shift */
    x[a] += x[b]; x[d] = L32(x[d] ^ x[a], 16);
    x[c] += x[d]; x[b] = L32(x[b] ^ x[c], 12);
    x[a] += x[b]; x[d] = L32(x[d] ^ x[a],  0);
    x[c] += x[d]; x[b] = L32(x[b] ^ x[c],  7);
}
```

Cryptol has a sister tool, called **SAW**, which can do this using deep black magic that we dare not speak of (yet).

SAW has its own typed language - called SAWScript - used for scripting proofs in an automated way, which we'll use.

SAW doesn't work on surface-level syntax. It works with intermediate representations, often ones your compiler splits out. In particular, SAW comes with tools for ingesting **LLVM bitcode, JVM Bytecode and Cryptol programs**.

Bibliography

[1] "Cryptography and Verification with Cryptol" YouTube, uploaded by Compose Conference, 03 May 2016, `https://www.youtube.com/watch?v=sC2_5WaavFc`