Cryptol: A Comprehensive Guide
- Mastering the Art of Cryptol Programming -

Ji Yong-Hyeon

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

November 3, 2024

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

Research (HIGHT)

```
#define BYTE    unsigned char       //  1-byte data type
#define WORD    unsigned short int  //  2-byte data type
#define DWORD   unsigned int        //  4-byte data type

void HIGHT_KeySched(
BYTE    *UserKey,
DWORD   UserKeyLen,
BYTE    *RoundKey);

void HIGHT_Encrypt(
BYTE    *RoundKey,
BYTE    *Data);

void HIGHT_Decrypt(
BYTE    *RoundKey,
BYTE    *Data);
```

```
rotateLeft8 : [8][8] -> [8][8]
rotateLeft8 block = drop‘{1} block # take‘{1} block

rotateRight8 : [8][8] -> [8][8]
rotateRight8 block = drop‘{7} block # take‘{7} block

right_update_MK : [16][8] -> [16][8]
right_update_MK mk = rotatedMK
where
(firstHalf, secondHalf) = splitAt‘{8} mk
rotatedFirstHalf = rotateRight8 firstHalf
rotatedSecondHalf = rotateRight8 secondHalf
rotatedMK = rotatedFirstHalf # rotatedSecondHalf

left_update_MK : [16][8] -> [16][8]
left_update_MK mk = rotatedMK
where
(firstHalf, secondHalf) = splitAt‘{8} mk
rotatedFirstHalf = rotateLeft8 firstHalf
rotatedSecondHalf = rotateLeft8 secondHalf
rotatedMK = rotatedFirstHalf # rotatedSecondHalf

HIGHT_KeySched_Cryptol : [16][8] -> [136][8]
HIGHT_KeySched_Cryptol MK = RK
where
RK = [ subkey i | i:[8] <- [0..135] ]
where
subkey i =
if i < 4
then MK @ (12 + i)
else if i < 8
then MK @ (i - 4)
else if i < 24
then MK @ (i - 8) + delta_table @ (i - 8)
else if i < 40
then (right_update_MK MK) @ (i - 24) + delta_table @ (i - 8)
else if i < 56
then (right_update_MK (right_update_MK MK)) @ ((i - 40)) + delta_table @ (i - 8)
else if i < 72
then (right_update_MK (right_update_MK (right_update_MK MK))) @ (i - 56)
+ delta_table @ (i - 8)
else if i < 88
then (right_update_MK (right_update_MK (right_update_MK (right_update_MK MK)))) @ (i - 72)
+ delta_table @ (i - 8)
else if i < 104
then (left_update_MK (left_update_MK (left_update_MK MK))) @ (i - 88)
+ delta_table @ (i - 8)
else if i < 120
then (left_update_MK (left_update_MK MK)) @ (i - 104) + delta_table @ (i - 8)
else
(left_update_MK MK) @ (i - 120) + delta_table @ (i - 8)
```

Chapter 2

CH1

2.1 Basic Syntax

Identifiers

Examples of Identifier

name	name1	name’	longer_name
Name	Name2	Name”	longerName

Keywords and Built-in Operators

Keywords

as	extern	include	interface	parameter	property	where
by	hiding	infix	let	pragma	submodule	else
constraint	if	infixl	module	primitive	then	
down	import	infixr	newtype	private	type	

Built-in Type-level Operators

Keywords

Operator	Meaning
+	Addition
−	Subtraction
*	Multiplication
/	Division
/^	Ceiling Division (/ rounded up)
%	Modulus
%^	Ceiling Modulus (Computing Padding)
^^	Exponentiation
`lg2`	Ceiling logarithm (base 2)
`width`	Bit Width (equal to `lg2(n+1)`)
`max`	Maximum
`min`	Minimum

Numeric Literals

Examples of Literals

```
254         // Decimal literal
0254        // Decimal literal
0b11111110  // Binary literal
0xFE        // Hexadecimal literal
0xfe        // Hexadecimal literal
```

Polynomial Literals

```
<| x^^6 + x^^4 + x^^2 + x^^1 + 1 |>  // :  [7], equal to 0b1010111
<| x^^4 + x^^3 + x |>                // :  [5], equal to 0b11010
```

Fractional Literals

```
10.2
10.2e3    // 10.2 * 10^3
0x30.1    // 3 * 64 + 1/16
0x30.1p4  // (3 * 64 + 1/16) * 2^4
```

Using _

```
0b_0000_0010
0x_FFFF_FFEA
```

2.2 Expressions

Calling Functions

```
f 2      // call ‘f’ with parameter ‘2’
g x y    // call ‘g’ with two parameters:  ‘x’ and ‘y’
h (x,y)  // call ‘h’ with one parameter, the pair ‘(x,y)’
```

Chapter 3

AES on Cryptol

3.1 Add Round Key

```
type AES128 = 4
type AES192 = 6
type AES256 = 8

type Nk = AES128

// For Cryptol 2.x | x > 0
// NkValid: ‘Nk -> Bit
// property NkValid k = (k == ‘AES128) || (k == ‘AES192) || (k == ‘AES256)

// Number of blocks and Number of rounds
type Nb = 4
type Nr = 6 + Nk

type AESKeySize  = (Nk*32)

// Helper type definitions
type GF28        = [8]
type State       = [4][Nb]GF28
type RoundKey    = State
type KeySchedule = (RoundKey, [Nr-1]RoundKey, RoundKey)
```

Chapter 4

HIGHT on Cryptol

SAWScript Helper Functions

1. `alloc_init`

Given a type ty and a value v of type ty, the function `alloc_init` allocates memory to store v and returns a pointer p to this memory.

$$\text{alloc\_init}(ty, v) = \begin{cases} p \leftarrow \text{crucible\_alloc}(ty); \\ \text{crucible\_points\_to}(p, v); \\ \text{return } p; \end{cases}$$

2. `alloc_init_readonly`

Given a type ty and a value v of type ty, the function `alloc_init_readonly` allocates read-only memory to store v and returns a pointer p to this memory.

$$\text{alloc\_init\_readonly}(ty, v) = \begin{cases} p \leftarrow \text{crucible\_alloc\_readonly}(ty); \\ \text{crucible\_points\_to}(p, v); \\ \text{return } p; \end{cases}$$

3. `ptr_to_fresh`

Given a name n and a type ty, the function `ptr_to_fresh` allocates a fresh variable x of type ty and returns a tuple (x, p) where p is a pointer to x.

$$\text{ptr\_to\_fresh}(n, ty) = \begin{cases} x \leftarrow \text{crucible\_fresh\_var}(n, ty); \\ p \leftarrow \text{alloc\_init}(ty, \text{crucible\_term}(x)); \\ \text{return } (x, p); \end{cases}$$

4. `ptr_to_fresh_readonly`

Given a name n and a type ty, the function `ptr_to_fresh_readonly` allocates a fresh variable x of type ty and returns a tuple (x, p) where p is a read-only pointer to x.

$$\text{ptr\_to\_fresh\_readonly}(n, ty) = \begin{cases} x \leftarrow \text{crucible\_fresh\_var}(n, ty); \\ p \leftarrow \text{alloc\_init\_readonly}(ty, \text{crucible\_term}(x)); \\ \text{return } (x, p); \end{cases}$$

5. `global_points_to`

Given a name n and a value v, the function `global_points_to` asserts that the global variable n has a value of v.

$$\text{global\_points\_to}(n, v) = \{\text{crucible\_points\_to}(\text{crucible\_global}(n), \text{crucible\_term}(v));$$

6. `global_alloc_init`

Given a name n and a value v, the function `global_alloc_init` declares that n is initialized and asserts that it has the value v.

$$\text{global\_alloc\_init}(n, v) = \begin{cases} \text{crucible\_alloc\_global}(n); \\ \text{global\_points\_to}(n, v); \end{cases}$$

LLVM Integer Type Aliases

$$\begin{aligned} i8 &= \text{llvm\_int}(8); \\ i16 &= \text{llvm\_int}(16); \\ i32 &= \text{llvm\_int}(32); \\ i64 &= \text{llvm\_int}(64); \\ i128 &= \text{llvm\_int}(128); \\ i384 &= \text{llvm\_int}(384); \\ i512 &= \text{llvm\_int}(512); \end{aligned}$$

4.1 Key Schedule

4.1.1 Whitening-Key

4.1.2 LFSR(Left Feedback Shift Register)

4.1.3 Sub-Key

4.1.4 Encryption and Decryption Key

4.2 Encryption

4.3 Decryption

$$\begin{tikzcd}
\text{Plaintext State} \arrow[r, "\mathbf{Enc}_{AES}"] \arrow[d, "\text{id}"'] & \text{Ciphertext State} \arrow[d, "\mathbf{Dec}_{AES}"] \\
\text{Plaintext State} \arrow[r, "\text{id}"'] & \text{Plaintext State}
\end{tikzcd}$$

Chapter 5

Research

In Cryptol '`foldl`' is a higher-order function that reduces a sequence (or list) to a single value by iteratively applying a binary function, starting from the left side of the sequence. It is similar to the fold operation found in many functional programming languages.

To understand '`foldl`', lets break it down mathematically. Given:

- A binary operation '`f`' of type '`(b, a) -> b`'
- An initial value '`z`' of type '`b`'
- A sequence '`xs`' of type '`[n]a`' (a sequence of '`n`' elements, each of type '`a`')

The '`foldl`' function can be defined as:

$$foldl\ f\ z\ [x_0, x_1, \ldots, x_{n-1}]$$

This can be described recursively as:

1. If the sequence is empty, the result is the initial value '`z`'.
2. Otherwise, apply the function '`f`' to the initial value '`z`' and the first element of the sequence '`x_0`', then recursively apply '`foldl`' to the result of this function application and the rest of the sequence.

Mathematically, this is:

$$\begin{aligned} foldl\ f\ z\ [] &= z \\ foldl\ f\ z\ (x : xs) &= foldl\ f\ (f\ z\ x)\ xs \end{aligned}$$

Example Let's take a concrete example to illustrate foldl. Suppose we want to compute the sum of a list of numbers using foldl.

Let:

- $f(a, b) = a + b$ (binary addition function)
- $z = 0$ (initial value)
- $xs = [1, 2, 3, 4]$ (sequence of numbers)

Using foldl to compute the sum:

$$foldl(+)0[1, 2, 3, 4]$$

Step-by-step:

1. Start with initial value $z = 0$.
2. Apply the function to the initial value and the first element: $f(0, 1) = 0 + 1 = 1$
3. Apply foldl to the result and the rest of the sequence: $foldl\ (+)\ 1\ [2, 3, 4]$
4. Repeat: $f(1, 2) = 1 + 2 = 3$, $foldl\ (+)\ 3\ [3, 4]$
5. Continue: $f(3, 3) = 3 + 3 = 6$, $foldl\ (+)\ 6\ [4]$
6. Finally: $f(6, 4) = 6 + 4 = 10$, $foldl\ (+)\ 10\ [\ ]$

Since the sequence is now empty, the result is 10.

Summary

- 'foldl' starts with an initial value and iterates through the sequence from left to right.
- It applies a binary function to the current accumulated value and the current element of the sequence.
- The result of this function application becomes the new accumulated value.
- The process repeats until the sequence is exhausted, at which point the accumulated value is returned as the result.

Understanding foldl helps in performing various reduction operations over sequences in a concise and functional manner in Cryptol.

Algorithm 1: General GCM

```
Input: $N, PT, key, AAD$
Output: $R$
$Zero \leftarrow 0$;
Step1:
    $CT, H, Y \leftarrow AES - CTR(PT, Zero, key)$;
    $t \leftarrow H$;
    $R \leftarrow AAD|0|CT|0|Len$;
end
Step2:
    for $i \leftarrow 1$ to $\log_2(N)$ do
        $t \leftarrow t^2$;
        $*(H + 4i) \leftarrow t$;
    end
end
Step3:
    $R \leftarrow 0$;
    if $N \gg 9 == 1$ then
        $R \leftarrow$ parallel_ghash_512$(R, H)$;
    end
    for $i \leftarrow 8$ to $4$ do
        if $N \gg i == 1$ then
            $temp \leftarrow$ parallel_ghash_2$^i(src, H)$;
            $R \leftarrow R \cdot H^{2^{i+1}} \oplus temp$;
        end
    end
end
$R \leftarrow R \oplus Y$;
return $R$;
```

Bibliography

[1] Galois, Inc. *Cryptol Reference Manual*. Available at https://galoisinc.github.io/cryptol/master/RefMan.html. Accessed April 2024.