# Mastering Cryptol: A Comprehensive Guide

## - From Basics to Advanced Patterns and Best Practices -

Ji Yong-Hyeon

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

March 5, 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

# Introduction to Cryptol Part I

## 1.1 Introduction to Cryptography and Formal Methods

### Overview

This week provides an introduction to the fundamental concepts of cryptography and formal methods. It sets the foundation required for understanding and utilizing Cryptol, a language specifically designed for creating and analyzing cryptographic algorithms.

#### 1.1.1 Introduction to Cryptography

1. Basic concepts and terminologies:
    - Definition of cryptography and its role in security.
    - Understanding encryption, decryption, keys, and algorithms.
2. Historical perspective and importance:
    - The evolution of cryptographic techniques.
    - Real-world applications and implications of cryptography.

#### 1.1.2 Introduction to Formal Methods

1. What are formal methods?
    - Definition and scope of formal methods in computer science.
    - The significance of formal methods in developing secure systems.
2. Importance in cryptography:
    - The role of formal methods in verifying cryptographic protocols and algorithms.
    - Examples of formal methods applied in cryptography.

#### 1.1.3 Overview of Cryptol

1. History and development:
    - The origins of Cryptol and its development history.

- Key features and capabilities of Cryptol.

2. Cryptol's place in cryptographic research and industry:
   - Applications of Cryptol in industry and academia.
   - Comparison with other cryptographic tools and languages.

## 1.2 Getting Started with Cryptol

This section provides a comprehensive guide for setting up the Cryptol environment on your computer. It covers the installation process for different operating systems and introduces basic commands and operations essential for working with Cryptol.

### 1.2.1 Installation and Setup

1. **Windows:**
   - Download the latest Cryptol release from the official GitHub page (https://github.com/GaloisInc/cryptol/releases).
   - Extract the downloaded ZIP file to your preferred location.
   - Add the extracted Cryptol directory to your system's environment variables:
     (a) Right-click on 'This PC' and select 'Properties'.
     (b) Click on 'Advanced system settings' and then 'Environment Variables'.
     (c) Under 'System Variables', find 'Path', and click 'Edit'.
     (d) Click 'New' and add the path to the extracted Cryptol folder.
     (e) Click 'OK' to close all dialogues.
   - Open Command Prompt and type `cryptol` to start the Cryptol REPL.
2. **macOS:**
   - Install Homebrew if it is not already installed, using the command

```
/bin/bash -c
"$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
  install/HEAD/install.sh)"
```

   - Install Cryptol using Homebrew by executing '`brew install cryptol`'.
   - Verify the installation by opening a terminal and typing '`cryptol`'.
3. **Linux:**
   - For Debian-based distributions, you can install Cryptol using the following commands:

```
@:~$ sudo apt-get update
@:~$ sudo apt-get install cryptol
```

   - For other distributions, download the appropriate binaries from the Cryptol GitHub releases page and follow similar steps as for Windows.
   - Verify the installation by opening a terminal and typing '`cryptol`'.

Setting up the working environment for Cryptol coding and testing:

- Choose a text editor or IDE that you are comfortable with for writing Cryptol scripts. Popular options include Visual Studio Code, Emacs, and Vim.
- Install any necessary plugins or extensions for your chosen editor that support Cryptol syntax highlighting and code completion.
- Create a dedicated workspace or directory for your Cryptol projects to keep your files organized.

## 1.3 Basic Syntax and Types in Cryptol

https://www.youtube.com/watch?v=sC2_5WaavFc

### 1.3.1 Data Types, Functions, and Modules

This subsection delves into the core elements of Cryptol's programming model: its type system, functions, and module system. Understanding these elements is crucial for effective Cryptol programming, particularly for cryptographic applications.

#### Understanding Cryptol's Type System

Cryptol's type system is designed to provide a high degree of safety and expressiveness, especially important in the context of cryptographic applications where precision is paramount.

1. **Bits and Sequences:**
    - Cryptol's basic type is the Bit, representing a single binary digit, which can be either `True` (1) or `False` (0).
    - Sequences are ordered collections of elements of the same type. In Cryptol, sequences are used to represent strings of bits (binary data), which are fundamental in cryptography.
    - A sequence of bits is declared as `[n]Bit`, where `n` specifies the sequence length. For example, `[8]Bit` represents a byte.
    - Cryptol supports sequence manipulations such as indexing, slicing, and concatenation, crucial for various cryptographic operations.
2. **Functions:**
    - Functions in Cryptol are first-class citizens, meaning they can be passed as arguments, returned from other functions, and stored in variables.
    - A function's type signature is defined by its input and output types. For example, a function taking two bytes and returning a byte has the type `[8]Bit -> [8]Bit -> [8]Bit`.
    - Cryptol functions support pattern matching, guards, and recursion, providing a powerful toolset for defining complex cryptographic transformations.

#### Defining and Using Modules in Cryptol

Modules in Cryptol help organize code into reusable and maintainable units, similar to modules or packages in other programming languages.

1. **Module Definition:**
    - A module in Cryptol is defined using the `module` keyword, followed by the module name and the module body enclosed in braces.
    - Inside a module, you can define types, functions, and constants. Modules can also include other modules.
    - Example:

```
module MyModule where
// Module contents go here
myFunction : [8]Bit -> [8]Bit
myFunction x = x ^ 0xFF
```

This defines a module named `MyModule` containing a function `myFunction` that performs a bitwise XOR with the byte value 0xFF.

2. **Using Modules:**
   - To use a module in Cryptol, you can import it using the `import` statement at the beginning of your Cryptol script or within another module.
   - Once imported, you can access the functions, types, and constants defined in the module.
   - Example:

     ```
     import MyModule
     result = myFunction 0xAB
     ```

     This imports `MyModule` and uses `myFunction` to compute the XOR of 0xAB and 0xFF.

The concepts of data types, functions, and modules are fundamental to programming in Cryptol. By mastering these, you will be well-equipped to tackle more advanced cryptographic programming challenges in Cryptol.

### 1.3.2 Type System and Type Inference

Cryptol's type system is one of its most powerful features, ensuring that operations are applied correctly and preventing a wide range of programming errors. Understanding the principles behind this type system and how type inference operates within it is crucial for effective Cryptol programming.

#### Principles Behind Cryptol's Strong Type System

Cryptol's type system is designed with the security and reliability needs of cryptographic algorithms in mind. Here are some of the foundational principles:

- **Strong Typing:** Every value in Cryptol has a type, and all types are checked at compile time. This eliminates a vast array of bugs related to type mismatches.
- **Rich Type Constructs:** Cryptol supports a variety of types beyond the basics, including fixed-size sequences, which are particularly useful in cryptography for representing data like keys and blocks.
- **Parametric Polymorphism:** This allows functions to operate on different types without changing the function's code, similar to generics in other programming languages.
- **Type Synonyms:** Cryptol allows the creation of new names for types, enhancing code readability and maintainability.
- **Type Classes:** These are similar to interfaces in other languages, defining a set of operations that a type must support.

Understanding these principles is vital as they inform the structure and capabilities of Cryptol's type system, allowing it to support complex, high-assurance cryptographic specifications with clarity and precision.

#### How Type Inference Simplifies Code in Cryptol

Type inference in Cryptol refers to the compiler's ability to automatically deduce the types of expressions without explicit type annotations from the programmer. This feature significantly simplifies code in several ways:

- **Less Boilerplate:** Programmers do not need to annotate every expression with a type, leading to clearer and more concise code.
- **Code Flexibility:** Functions can automatically work with different types (as long as they make sense in context), reducing the need for multiple versions of the same function.
- **Error Detection:** The compiler uses type inference to detect inconsistencies and potential bugs without needing exhaustive type specifications from the programmer.
- **Generality:** Inferred types are often more general than what a programmer might specify, allowing for more reusable and abstract code.

However, while type inference simplifies code, understanding how types are determined is important for debugging and for writing correct, secure cryptographic algorithms. In some cases, explicitly stating types can clarify intentions and ensure that the compiler's type deductions align with the programmer's expectations.

In practice, balancing the use of type inference with explicit type annotations can lead to code that is both concise and clear. It allows Cryptol to serve as an effective tool for implementing and verifying cryptographic algorithms, leveraging its strong type system to ensure correctness and security.

### 1.3.3 Type System and Type Inference

Cryptol's type system is integral for ensuring the safety and correctness of cryptographic code. Below, we delve into the principles of Cryptol's type system and demonstrate how type inference simplifies coding through specific examples.

#### Principles Behind Cryptol's Strong Type System

Cryptol's type system is meticulously designed to cater to cryptographic needs, ensuring operations are correctly applied and eliminating common programming errors. The following principles underpin Cryptol's type system:

- **Strong Typing:** In Cryptol, every value is associated with a type, which is determined at compile time. This strict approach helps to prevent errors related to incorrect type usage.
- **Rich Type Constructs:** Cryptol supports types like fixed-size sequences, crucial for representing cryptographic data such as keys and message blocks.
- **Parametric Polymorphism:** Functions in Cryptol can be written to work with any type, enhancing code reusability and flexibility.
- **Type Synonyms and Type Classes:** Cryptol allows for the creation of new names for existing types and the definition of type classes, facilitating clearer and more maintainable code.

For example, consider the type signature for a function that performs an exclusive or (XOR) on two bit sequences:

```
xor : {n} (fin n) => [n] -> [n] -> [n]
```

This signature indicates that ‘xor‘ is a function that takes two sequences of bits of the same length ‘n‘ and returns a new sequence of bits of the same length. The ‘n‘ syntax indicates that ‘n‘ is a parameter to the type, and ‘(fin n)‘ ensures that ‘n‘ is finite, which is necessary for the operation to make sense in the context of fixed-size sequences.

### How Type Inference Simplifies Code in Cryptol

Type inference in Cryptol reduces the need for explicit type annotations, allowing the compiler to deduce types automatically. This simplification can be illustrated through practical examples:

- **Less Boilerplate:** Without type inference, every variable and expression would need an explicit type, making the code verbose and harder to read. With type inference, many of these annotations are unnecessary. For instance, when defining a new variable, ‘a = 5‘, Cryptol automatically understands ‘a‘ to be of type ‘Integer‘.
- **Code Flexibility:** A function written without specific type annotations can operate over different types. For example, the function ‘inc x = x + 1‘ can be used to increment an integer, a floating-point number, or even a sequence of numbers, without defining separate functions for each type.
- **Error Detection:** Type inference helps detect inconsistencies in code. If you attempt to use a sequence where a number is expected, Cryptol’s type system flags an error, preventing potential bugs.
- **Generality:** By inferring the most general type possible, Cryptol makes functions more reusable. Consider a function ‘f x y = x && y‘. Without specifying types, Cryptol infers the most general form, allowing ‘f‘ to be applied to any two arguments of the same type that supports the ‘&&‘ operation.

To solidify understanding, let’s look at a specific example illustrating type inference:

```
let add a b = a + b
```

In this case, Cryptol infers that ‘a‘ and ‘b‘ must be of a type that supports addition. If ‘add‘ is used with two integers, Cryptol infers the type of ‘a‘ and ‘b‘ as ‘Integer‘. If it is used with two sequences of bits, it infers their types accordingly, showcasing the power and flexibility of type inference in making code more concise and versatile.

By leveraging these examples, it becomes evident how Cryptol’s type system and type inference mechanism work hand in hand to ensure that cryptographic algorithms are both secure and correct, while also being concise and readable.

1. Basic commands and operations:

    - Introduction to the Cryptol command line interface:
        - The Cryptol command line interface (CLI) allows you to interact with Cryptol directly from your terminal or command prompt.
        - You can enter the interactive mode by simply typing `cryptol` without any arguments. In this mode, you can type Cryptol expressions and commands directly and see their results.
        - Exit the interactive mode by typing `:q` or pressing Ctrl-D.
    - Common commands for interpreting and compiling Cryptol scripts:
        - To load a Cryptol script from the CLI, use the `:load` command or its shorthand `:l`, followed by the file name, e.g., `:load MyScript.cry`.
        - To check the type of an expression, use the `:type` command or its shorthand `:t`, followed by the expression, e.g., `:type [1..10]`.
        - To evaluate an expression or execute a function, simply type it into the CLI and press Enter.
        - Use the `:help` command to display a list of all available commands and their descriptions.

1. Environment setup:

    - Instructions for installing Cryptol on various operating systems.
    - Setting up the working environment for Cryptol coding and testing.

2. Basic commands and operations:

    - Introduction to the Cryptol command line interface.
    - Common commands for interpreting and compiling Cryptol scripts.

### 1.3.4 Basic Commands and Operations

Once Cryptol is installed, you can start using it through the command line interface (CLI). Here are some basic commands to get you started:

1. **Starting the Cryptol REPL:**

    - Open your command line interface (Command Prompt, Terminal, etc.).
    - Type `cryptol` and press Enter to start the Cryptol Read-Eval-Print Loop (REPL).
    - You should see the Cryptol version number and a welcome message.

2. **Loading Cryptol modules:**

    - In the REPL, use the `:load` command (or `:l` for short) followed by the filename to load a Cryptol module, e.g., `:load MyModule.cry`.
    - Once loaded, you can access the functions defined within the module.

3. **Interpreting Cryptol scripts:**

    - You can interpret individual commands directly in the REPL by typing them and pressing Enter.

- For example, to define a new function, simply type the function definition and press Enter.
- Use the `:?` command to display a list of all available commands within the REPL.

4. **Exiting the REPL:**

   - To exit the REPL, type `:quit` or `:q` and press Enter.
   - Alternatively, you can press Ctrl+D (EOF) on Linux and macOS or Ctrl+Z followed by Enter on Windows.

By following these steps and familiarizing yourself with the basic commands, you will be well-prepared to begin exploring the vast capabilities of Cryptol for cryptographic verification and formal proofs.

### 1.3.5 Basic Syntax and Types in Cryptol

1. Data types, functions, and modules:
    - Understanding Cryptol's type system - bits, sequences, and functions.
    - How to define and use modules in Cryptol.
2. Type system and type inference:
    - The principles behind Cryptol's strong type system.
    - How type inference simplifies code in Cryptol.

### 1.3.6 First Cryptol Programs

1. Writing and testing simple programs:
    - Creating basic Cryptol scripts and functions.
    - Running and testing Cryptol code for correctness.
2. Understanding Cryptol's REPL (Read-Eval-Print Loop):
    - Interactive mode vs. script mode in Cryptol.
    - Practical exercises to reinforce learning and understanding.

# 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:**

```c
uint32_t f(uint32_t x, uint32_t y) {
    return x + y;
}
```

**Haskell:**

```haskell
f :: Word32 -> Word32 -> Word32
f x y = x + y
```

**Cryptol:**

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

Large or “weird” numbers.

```
f : [17] -> [17]
f x = -x

g : [23424] -> [548683776]
g x = x * x
```

Sequence are fundamental

```
[0 .. 3] = [0, 1, 2, 3]
f : { x } x -> [4]x
f x = [x, x, x, x]
// Define a sequence of 5 integers
sequenceOfInts : [5]Integer
sequenceOfInts = [1, 2, 3, 4, 5]

// Define a sequence of bits (binary sequence)
sequenceOfBits : [8]Bit
sequenceOfBits = [1, 0, 1, 1, 0, 0, 1, 0]

// Define a nested sequence (matrix of bits)
matrixOfBits : [3][4]Bit
matrixOfBits = [[1, 0, 1, 0], [0, 1, 0, 1], [1, 1, 1, 1]]

// Accessing elements and slices
firstElement = sequenceOfInts @ 0      // Access the first element (index starts
    from 0)
firstTwoBits = sequenceOfBits @ [0..1] // Access the first two bits
secondRow = matrixOfBits @ 1           // Access the second row of the matrix

// Sequence operations
reversedSequence = reverse sequenceOfInts    // Reverse the sequence
concatenatedSequence = sequenceOfInts # [6] // Concatenate [6] to the end of the
    sequence
```

```
// Define a sequence of 16 bits
myBits : [16]Bit
myBits = [1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1]

// Split the sequence into chunks of 4 bits
splitChunks : [4][4]Bit
splitChunks = split(myBits)

// Define a list of sequences
myChunks : [4][4]Bit
myChunks = [[1, 0, 1, 1], [0, 0, 1, 0], [1, 1, 1, 0], [1, 0, 0, 1]]

// Join the chunks into a single sequence
joinedSequence : [16]Bit
joinedSequence = join(myChunks)
```

## 2.2 Split and Join

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

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
```

```
split : { parts, each, a}
    (fin each
    ) => [parts * each]a -> [parts][each]a

g: (a, b) (fin a) => [2 * a]b -> [2][a]b
g z = {x, y}
  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