ECE 443/518 – Computer Cyber Security
Lecture 05 Go

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

September 3, 2025

Outline

Go Introduction

Cryptography in Go

Reading Assignment

▶ This lecture: Go introduction

▶ Next lecture (Fri. 9/5): UC 11.2, 11.3, 11.5, 12, 5.1.6

Outline

Go Introduction

Cryptography in Go

Go

▶ The Go programming language.
  ▶ Version 1.0: March 2012
▶ Modernization of C for simplicity, safety, and readability.
  ▶ Package management, garbage collection, concurrency, etc.
  ▶ Simplified C syntax with standard tool to format code.
  ▶ Exactly the same value semantics as C.
  ▶ Adopt common C patterns to support array/slice and OOP.

Hello World

```go
// hw/hw.go
package main

import "fmt"

func main() {
        fmt.Println("Hello world!")
}
```

- ▶ Go uses the same entrypoint `main` as C.
  - ▶ It has to be inside `package main`
- ▶ Save the code to `hw.go` and run it via `go run hw.go`
- ▶ Language features
  - ▶ Both `//` and `/**/` work for comments
  - ▶ Use `import` instead of `#include`
  - ▶ Use `func` to define a function
  - ▶ No need to use `;`
  - ▶ `{` must be at the end of the line

Variable

```go
// swap/main.go
package main

import "fmt"

func main() {
        var a int = 1
        b := 2
        fmt.Printf("before swap: a = %d, b = %d\n", a, b)
        swap(&a, &b)
        fmt.Printf("after swap: a = %d, b = %d\n", a, b)
}
```

- A variable can be defined using `var` and then initialized.
- Or you can use `:=` to define and initialize a variable.
  - Without the need to specify a type.
  - The variable still has a type and cannot be changed.
- Usually, library names are lowercase while library functions are uppercase.

Pointer

```go
// swap/swap.go
package main

func swap(pa, pb *int) {
        *pa, *pb = *pb, *pa
}
```

- ▶ Pointers `*T` are addresses to variables of type `T`
    - ▶ Allow you to change a variable outside of the current function.
    - ▶ Same as C, use `&` to take address for a variable and use `*` to refer to the variable using the pointer.
- ▶ Types can be omitted for the function parameters if they have the same type.
- ▶ Multiple variables can be assigned at the same time.

Go Module

- Since `swap` is in a different file as `main`, we cannot run this more complicated program directly.
- Use `go mod init swap` to initialize a Go module to manage multiple go files.
- Run it as `go run .`
  - You can also debug it in VSCode or other IDEs.

Array and Slice

```go
// slice/slice.go
package main

import "fmt"

func main() {
        var a [10]int
        s := make([]int, 0)
        for i := 0; i < 10; i++ {
                a[i] = i
                s = append(s, i*i)
        }
        for i, val := range s {
                fmt.Printf("s[%d]=%d=%d*%d\n", i, val, a[i], a[i])
        }
}
```

- ▶ Arrays like `a`, as those in C/C++/Java, are of fixed size.
- ▶ Slices like `s` are more flexible.
    - ▶ Use `make` to create a slice with initial size.
    - ▶ Use `append` to append an element to the end.
- ▶ Use `[]` to access elements using 0-based indices.

for Loops

```go
for i := 0; i < 10; i++ {
  a[i] = i
  s = append(s, i*i)
}
for i, val := range s {
  fmt.Printf("s[%d]=%d=%d*%d\n", i, val, a[i], a[i])
}
```

▶ The most simple `for` loops use three statements
   `for initialization; condition; postcondition`
  ▶ Similar to C/C++/Java but no parentheses
  ▶ You'll need to use `i++` instead of `++i`
▶ The range `for` loops allow to obtain both the index and the element at the same time.
▶ Use `break` to exit the loop.
▶ Use `continue` to exit the current iteration.

More for Loops

```go
// a while loop
for condition {
  ...
}
// an infinite loop
for {
  ...
}
```

- ▶ There is no `while` or `do while` loop in Go. Every loop is a `for` loop.

What is a slice?

```
func assign() {
        a := []int{0, 1, 2, 3, 4}
        b := a

        b[0] = 100

        fmt.Printf("after assign: a=%v, b=%v\n", a, b)
}
```

- ▶ A slice stores the address of the first element and the number of elements.
  - ▶ A memory area is allocated from the heap to store the elements.
  - ▶ No, you don't need to call `malloc`, `free`, etc. like in C or other languages.
  - ▶ `[]` will be able to check if the index is out of bound or not.
- ▶ Assignment `=` will only copy the address and the length so now `a` and `b` refer to the same memory area.

Copy a Slice

```go
func mycopy() {
        a := []int{0, 1, 2, 3, 4}

        b := make([]int, len(a))
        copy(b, a)

        b[0] = 100

        fmt.Printf("after copy: a=%v, b=%v\n", a, b)
}
```

▶ The copy function is able to make a copy of the slice so that you can have two slices referring to two separated memory areas.

Slicing a Slice

```go
func slicing() {
        a := []int{0, 1, 2, 3, 4}
        b := a[1:3]
        c := a[:len(a)-1]
        d := a[2:]

        fmt.Printf("a=%v, b=%v, c=%v, d=%v\n", a, b, c, d)
}
```

- ▶ Use [begin:end] to slicing a slice.
    - ▶ Half close half open (begin included, end excluded).
    - ▶ begin = 0 if omitted, end = len() if omitted.
    - ▶ No negative indices like in Python.
- ▶ Slicing is essentially pointer arithmetics in C so all the slices a, b, c, d now share the same memory area.
    - ▶ What if we change a[2] to 100? b[1], c[2], and d[0] will all change to 100
    - ▶ If we append to a later, We should not use b, c, and d any more!

Branches

```go
// rand/rand.go
package main

import (
        "fmt"
        "math/rand"
)

func main() {
        d := rand.Float64()
        if d < 0.4 {
                fmt.Println("Win!")
        } else if d > 0.6 {
                fmt.Println("Lose!")
        } else {
                fmt.Println("Tie!")
        }
}
```

- Similar to C/C++/Java but no parentheses.
  - Recall that `{` must be at the end of the line
  - If there is an `else`, then `}` must be on the same line as well.

More Tutorials

- Tutorials can be found at `https://go.dev/doc/tutorial/`
- Use the Go Playground `https://go.dev/play/`

Outline

The Go crypto Package

```go
// crypto/crypto.go
package main

import (
        "crypto/aes"
        "crypto/cipher"
        "crypto/rand"
        "encoding/hex"
        "fmt"
        "io"
)
```

- ▶ The Go crypto package provides many standardized cryptographg functions.
    - ▶ Together with many other packages like `hex` that allows to handle bytes and messages conveniently.

AES in CBC Mode

```go
key, _ := hex.DecodeString("000102030405060708090A0B0C0D0E0F")

plaintxt := "0123456789ABCDEF0123456789ABCDEF"
pbuf := []byte(plaintxt)
iv := make([]byte, 16)
rand.Read(iv)

aesblock, _ := aes.NewCipher(key)

cbcEnc := cipher.NewCBCEncrypter(aesblock, iv)
ciphertxt := make([]byte, len(pbuf))
cbcEnc.CryptBlocks(ciphertxt, pbuf)

cbcDec := cipher.NewCBCDecrypter(aesblock, iv)
pbuf2 := make([]byte, len(ciphertxt))
cbcDec.CryptBlocks(pbuf2, ciphertxt)
decrypted := string(pbuf2)
```

- ▶ Padding is ignored – the message is of multiples of 16 bytes
- ▶ Need to convert between strings and bytes for text messages.
- ▶ Use the `crypto/rand` package to generate cryptographically secure pseudorandom IV.

AES in Counter Mode

```go
key, _ := hex.DecodeString("000102030405060708090A0B0C0D0E0F")

plaintxt := "0123456789ABCDEF0123456789"
pbuf := []byte(plaintxt)
iv := make([]byte, 16)
rand.Read(iv)

aesblock, _ := aes.NewCipher(key)

ctrStream := cipher.NewCTR(aesblock, iv)
ciphertxt := make([]byte, len(pbuf))
ctrStream.XORKeyStream(ciphertxt, pbuf)

ctrStream2 := cipher.NewCTR(aesblock, iv)
pbuf2 := make([]byte, len(ciphertxt))
ctrStream2.XORKeyStream(pbuf2, ciphertxt)
decrypted := string(pbuf2)
```

▶ No padding is needed.

Summary

- ► Why Go?
  - ► A modern language supporting many easy-to-use features.
  - ► Able to work with memory bytes at low level.
  - ► A crypto library incorporating standardized cryptography practices.