DINING CRYPTOGRAPHERS NETWORK IMPLEMENTATION IN GO

ISA 763 – Security Protocol Analysis

Project Report

Date: 04/11/2018

Team Members:
Jaivendra Singh
Jonathan Torchia
Rutwij Kulkarni

Contents

Table of Figure

1. Introduction

The dining cryptographer's problem was proposed by David Chaum in 1988 [1]. It involves three cryptographers sitting down for dinner where their waiter informs them that the bill for their meal must be paid anonymously. The payer could be any one of the three cryptographers or the NSA (U.S. National Security Agency).

The cryptographers use a protocol to resolve the uncertainty if one of the cryptographers is paying or the NSA. Per the protocol, each cryptographer begins by flipping a coin. The outcome of this coin can only be seen by that cryptographer and the one sitting to the right-hand side of him. After each coin flip, the cryptographer says out loud whether the outcome of his coin flip and that of the cryptographer sitting to his left is same or different. If one of the cryptographers is the payer, he states the opposite of what he sees. In the end, a count is done for differences uttered at the table. An odd count indicates that a cryptographer is paying whereas an even count indicates that NSA is paying. The protocol preserves anonymity as even if any of the cryptographer is paying, it is not possible with the disclosures done to find out which cryptographer paid [1].

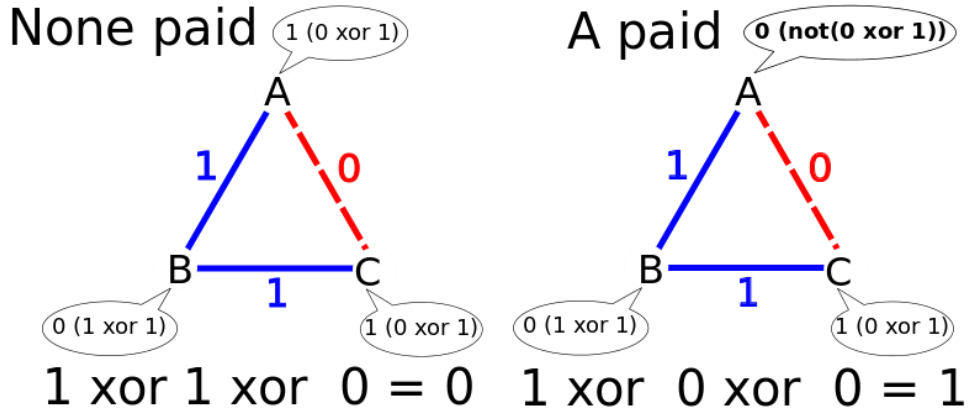

Figure 1: Dining Cryptographers protocol run

2. Protocol Specifications

The following implementation of the dining cryptographer's problem in GO has been simplified to allow easy verification of results. Following are the specifications:

1. There are four Cryptographers (A, B, C, 0), one Observer, one Restaurant owner, and the NSA participating in the protocol.
2. The outcome of each coin flip is either true or false (Boolean)
3. The outcome of each comparison done by every cryptographer is either same (true) or different (false) (Boolean)

4. Cryptographer 0 acts essentially as the verifier. Cryptographer 0 can watch all the coin flip outcomes after the communication between the cryptographers has occurred. Cryptographer 0 can also view the assertions made by the other cryptographers.
5. The Observer can only view the assertions made by the three cryptographers and not the outcomes for each coin flip.
6. The Restaurant owner computes the outcome of all assertions to determine who is paying the bill, the NSA or one of the cryptographers.

3. Code Design

1) **Every coin is its own process**
   a) Uses flip function to toss a coin

2) **Every cryptographer is its own process**
   a) Uses compare function to determine outcome: Same or different

3) **The restaurant owner is its own process**
   a) Uses assertions/outcomes by cryptographers to compute the payer

4) **Channels used to communicate between processes**
   a) Channels allow synchronization without explicit locks
   b) Channels exists between the coin and cryptographer processes
   c) Channels also used between cryptographer and restaurant owner process
   d) Cryptographer 0 has channels with both coins and cryptographer processes.

5) **Used WaitGroups to coordinate between the different actors.**
   a) Example: The restaurant owner can only compute result after all cryptographers declare their assertion.

4. GO Program Modules

4.1. Cryptographer and NSA Objects

```
// Define Cryptographer Actor
type Crypto struct {
        coin    bool    // Result of coin flip
        paid    bool    // Is the cryptographer payer?
        xorval  bool    // Outcome of comparision
        identity        string  // Identity of cryptographer
}

//Define NSA Actor
type NSA struct {
        paid bool       // Is the NSA paying?
        identity        string
}
```

Figure 2: Crytographers and NSA structures

To implement the various properties concerning the cryptographers like the coin toss value, payer status, identity etc. we used structures. Similarly, for NSA actor we defined a structure to track payer status and its identity.

4.2. WaitGroup

```
//Waitgroup to sync processes
var wg[4] sync.WaitGroup
```

Figure 3: WaitGroup Array

To synchronize the separate processes corresponding to coins, cryptographers, restaurant owner as well as the cryptographer0 we used WaitGroups. Each actor or entity group was added to a WaitGroup. For example, the process for cryptographers that compared the coin values was initiated only after the process group for coin processes finished.

4.3. Flip Function

```
//Simulates flipping a coin
func (c *Crypto) Flip(channel chan bool, channel_0 chan bool) {
        random := rand.Int()
        if random%2 == 0 {
                c.coin = true
        } else {
                c.coin = false
        }
        fmt.Println(c.identity," coin value: ",c.coin)
        channel <- c.coin // Use this channel to share coin value with cryptographer on right
        channel_0 <- c.coin // Use this channel to share coin value with cryptographer0
        close(channel)
        wg[0].Done() // Signal to waitgroup of completion
}
```

Figure 4: Flip function for coin processes

The flip function essentially selects a random integer and then takes a modulus 2 to get 0 or 1 (true or false). Once the coin value is generated, it is shared to the Cryptographer process using the channel "channel". This channel is later closed as no additional values need to be written to it ensuring integrity. Moreover, the value is also shared with Cryptographer0 using channel "channel_0".

4.4. Compare Function

```
// Compare function
func (c *Crypto) compare(left_coin bool, channel_out chan bool, channel_0 chan bool) {
        if c.paid {
                c.xorval = reverse_xor(c.coin, left_coin)
        } else {
                c.xorval = xor(c.coin, left_coin)
        }
        if c.xorval {
                fmt.Println(c.identity," declares outcome: Different")
        } else {
                fmt.Println(c.identity," declares outcome: Same")
        }
        channel_out <- c.xorval // Use this channel to share outcome with Restaurant Owner or Observer
        channel_0 <- c.xorval // Use this channel to share outcome with Cryptographer0
        wg[1].Done() // Signal to waitgroup of completion
}
```

Figure 5: Compare function for cryptographer processes

The cryptographer process receives the coin flip values using the channels established. If the concerned cryptographer is the payer, the compare function computes a xor of the two-coin values of the cryptographer himself and the cryptographer on the left and then returns the opposite of that. Whereas, if the cryptographer is not paying he publishes the outcome as it is. The outcome is then shared with Restaurant owner using the channel "channel_out". The outcome is also shared with cryptographer0 using the channel "channel_0".

4.5. Restaurant Owner Function

```
// Restaurant Owner Function
func restaurant_owner(a bool, b bool, c bool) {
        var same, diff int
        var outcome[3] bool
        outcome[0]=a // Assign A's outcome
        outcome[1]=b // Assign B's outcome
        outcome[2]=c // Assign C's outcome
        same = 0
        diff = 0
        // Count the number of Different and Same utterances
        for i :=0; i<3; i++ {
                if outcome[i] {
                        diff = diff + 1
                } else {
                        same = same + 1
                }
        }
        fmt.Println("Result Count -> ||Same:",same,"||  ||Different:",diff,"||\n")
        // If "different" utterance is even --> NSA Paid, if odd --> cryptographer paid
        if (diff % 2 != 0){
                fmt.Println("*** Odd count of \"Different\" Uttered. So.. ***")
                fmt.Println("=> Restaurant Owner declares a cryptographer paid !!")
        } else {
                fmt.Println("*** Even count of \"Different\" Uttered. So.. ***")
                fmt.Println("=> Restaurant Owner declares the NSA Paid !!")
        }
        wg[2].Done() // Signal to waitgroup of completion
}
```

Figure 6: Restaurant owner function for Restaurant Process

The restaurant function is implementing a three input XOR to find out if the NSA paid or if a cryptographer paid. The input is the three outcomes from the compare function.

4.6. Cryptographer0 Function

```
//Cryptographer 0 Function
func crypt0(aCoin bool, bCoin bool, cCoin bool, axor bool, bxor bool, cxor bool ) {
        fmt.Println("-----------------------------------")
        fmt.Println("     ==> Cryptographer A <==       ")
        fmt.Println("-----------------------------------")
        fmt.Println("Computed Result: A xor C =>",xor(aCoin, cCoin))
        fmt.Println("Declared Result by A     =>",axor)

        fmt.Println("\n-----------------------------------")
        fmt.Println("        Cryptographer B        ")
        fmt.Println("-----------------------------------")
        fmt.Println("Computed Result: B xor A =>",xor(bCoin, aCoin))
        fmt.Println("Declared Result by B     =>",bxor)

        fmt.Println("\n-----------------------------------")
        fmt.Println("        Cryptographer C        ")
        fmt.Println("-----------------------------------")
        fmt.Println("Computed Result: C xor B =>",xor(cCoin, bCoin))
        fmt.Println("Declared Result by C     =>",cxor)
        fmt.Println("-----------------------------------\n")
        if reverse_xor(aCoin, cCoin) == axor {
                fmt.Println("*** Cryptographer A said opposite --> Cryptographer A Paid! ***")
        } else if reverse_xor(bCoin, aCoin) == bxor {
                fmt.Println("*** Cryptographer B said opposite --> Cryptographer B Paid!")
        } else if reverse_xor(cCoin, bCoin) == cxor {
                fmt.Println("*** Cryptographer C said opposite --> Cryptographer C Paid!")
        } else {
                fmt.Println("*** Nobody said opposite so The NSA paid! ***\n\n")
        }
        wg[3].Done() // Signal to waitgroup of completion
}
```

Figure 7: Cryptographer0 function for Cryptographer0 process

The Cryptographer0 function is basically reverse engineering the protocol since it has access to all the diners coin results, as well as XOR results. Based upon the reverse engineering, the function can deduct who paid accurately.

4.7. XOR Functions

```
// XOR operation
func xor(a bool, b bool) bool {
        return a != b
}

// Reverse XOR operation
func reverse_xor(a bool, b bool) bool {
        return !(a != b)
}
```

Figure 8: XOR Function implementation

The XOR and reverse XOR are common basic implementations.

4.8. Main Function

```
func main() {

        // Channels for communication between cryptographers
        channel_A := make(chan bool, 1) // Channel for Crytographer A's coin shared to B
        channel_B := make(chan bool, 1) // Channel for Crytographer B's coin shared to C
        channel_C := make(chan bool, 1) // Channel for Crytographer C's coin shared to A
        channel_O := make(chan bool, 3) // Channel for Crytographers Outcome shared with Restaurant Owner
        channel_0A := make(chan bool, 2) // Channel shared by Crytographers 0,A
        channel_0B := make(chan bool, 2) // Channel shared by Crytographers 0,B
        channel_0C := make(chan bool, 2) // Channel shared by Crytographers 0,A

        //Get random seed as current time
        rand.Seed(time.Now().UTC().UnixNano())

        // Set Defaults
        a := Crypto{paid: false, identity: "Cryptographer A"}
        b := Crypto{paid: false, identity: "Cryptographer B"}
        c := Crypto{paid: false, identity: "Cryptographer C"}
        n := NSA{paid: false, identity: "NSA"}

        //determine payer randomly
        payer := rand.Int() % 4
        fmt.Println("-------------------------------------")
        fmt.Println("  ==> Randomly Selecting Payer <==           ")
        fmt.Println("-------------------------------------")
        if payer == 0 {
                n.paid = true
                fmt.Println("  Selected Payer:", n.identity)
        }
        if payer == 1 {
                a.paid = true
                fmt.Println("  Selected Payer:", a.identity)
        }
        if payer == 2 {
                b.paid = true
                fmt.Println("  Selected Payer:", b.identity)
        }
        if payer == 3 {
                c.paid = true
                fmt.Println("  Selected Payer:", c.identity)
        }
        fmt.Println("-------------------------------------\n")
```

Figure 9: Initialization of objects and channels in main function

```
        // All crytographers flip coins simultaneously
        fmt.Println("-----------------------------------------------")
        fmt.Println(" ==> Flip results visible to Crytographer0 <==  ")
        fmt.Println("-----------------------------------------------\n")
        wg[0].Add(3) // Add the three coin processes to WaitGroup
        go a.Flip(channel_A, channel_0A)
        go b.Flip(channel_B, channel_0B)
        go c.Flip(channel_C, channel_0C)
        wg[0].Wait() // Wait for the coin processes to complete
        fmt.Println("---------------------------------------\n")

        // All crytographers compare coins and broadcast
        fmt.Println("-----------------------------------------")
        fmt.Println(" ==> Cryptographers declare outcomes <==   ")
        fmt.Println("-----------------------------------------")
        fmt.Println("** Visible to: Observer, Restaurant Owner, Cryptographer 0,A,B,C **\n")
        wg[1].Add(3) // Add the three cryptographer processes to WaitGroup
        go a.compare(<-channel_C, channel_O, channel_0A)
        go b.compare(<-channel_A, channel_O, channel_0B)
        go c.compare(<-channel_B, channel_O, channel_0C)
        wg[1].Wait() // Wait for the cryptographer processes to complete
        fmt.Println("---------------------------------------\n")
        fmt.Println("-----------------------------------------")
        fmt.Println(" ==> Restaurant Owner computes payer <==    ")
        fmt.Println("-----------------------------------------")

        //Restaurant Owner Process
        wg[2].Add(1) // Add the restaurant owner process to WaitGroup
        go restaurant_owner(<-channel_O,<-channel_O,<-channel_O)
        wg[2].Wait() // Wait for the restaurant processes to complete
        fmt.Println("---------------------------------------\n")
        fmt.Println("---------------------------------------")
        fmt.Println(" ==> Cryptographer0 verifies payer <== ")
        fmt.Println("---------------------------------------")

        // Cryptographer0 Process
        wg[3].Add(1) // Add the cryptographer 0 process to WaitGroup
        go crypt0(<-channel_0A , <-channel_0B, <-channel_0C, <-channel_0A , <-channel_0B, <-channel_0C)
        wg[3].Wait() // Add the cryptographer 0 process to complete
}
```

Figure 10: Running coin, cryptographer, restaurant owner and cryptographer0 process

5. Results

```
osboxes@osboxes:~/project3$ go run dining.go
-------------------------------------
      Randomly Selecting Payer
-------------------------------------
  Selected Payer: Cryptographer A
-------------------------------------

-------------------------------------
 Flip results visible to Crytographer0
-------------------------------------

Cryptographer C  coin value:  true
Cryptographer A  coin value:  false
Cryptographer B  coin value:  true
-------------------------------------

-------------------------------------
    Cryptographers declare outcomes
-------------------------------------
** Visible to: Observer, Restaurant Owner, Cryptographer 0,A,B,C **

Cryptographer C  declares outcome: Same
Cryptographer A  declares outcome: Same
Cryptographer B  declares outcome: Different
-------------------------------------

-------------------------------------
    Restaurant Owner computes payer
-------------------------------------
Result Count -> ||Same: 2 ||  ||Different: 1 ||

*** Odd count of "Different" Uttered. So.. ***
=> Restaurant Owner declares a cryptographer paid !!
-------------------------------------

-------------------------------------
    Cryptographer0 verifies payer
-------------------------------------
-------------------------------------
    Cryptographer A
-------------------------------------
Computed Result: A xor C => true
Declared Result by A     => false

---------------------------------
    Cryptographer B
---------------------------------
Computed Result: B xor A => true
Declared Result by B     => true

---------------------------------
```

Figure 11: Result Window 1

Figure 12: Result Window 2

6. Conclusion

The results prove that Dining cryptographers protocol satisfies the anonymity requirement with respect to the Observer and restaurant owner. Even without knowing the value of each cryptographer's coin (secret), the restaurant owner can compute who among the NSA or the cryptographers paid the bill.

This essentially protects the anonymity of an individual cryptographer.

The verifier Cryptographer 0 reaches the same conclusion as the protocol result.

References

[1] D. Chaum. The dining cryptographers problem: unconditional sender and recipient untraceability. In Journal of Cryptology, 1(1), pp. 65-75, 1988.

[2] Online GO Documentation: http://golang.org

[3] TutorialsPoint GO Language: https://www.tutorialspoint.com/go/index.htm

[4] Dining Cryptographers GO implementation: http://cpansearch.perl.org/src/SHEVEK/Crypt-Dining-1.01/lib/Crypt/Dining.pm

[5] Dining Philosophers GO implementation: http://f.souza.cc/2011/10/go-solution-for-dining-philosophers.html

[6] Dining Cryptographers Problem Wikipedia: https://en.wikipedia.org/wiki/Dining_cryptographers_problem

Appendix

I. Code:

```
package main

import (
    "fmt"
    "math/rand"
    "time"
    "sync"
)

// Define Cryptographer Actor
type Crypto struct {
    coin    bool   // Result of coin flip
    paid    bool   // Is the cryptographer payer?
    xorval  bool   // Outcome of comparision
    identity       string  // Identity of cryptographer
}

//Define NSA Actor
type NSA struct {
    paid bool      // Is the NSA paying?
    identity       string
}

//Waitgroup to sync processes
var wg[4] sync.WaitGroup

//Simulates flipping a coin
func (c *Crypto) Flip(channel chan bool, channel_0 chan bool) {
    random := rand.Int()
```

```
    if random%2 == 0 {
            c.coin = true
    } else {
            c.coin = false
    }
    fmt.Println(c.identity," coin value: ",c.coin)
    channel <- c.coin // Use this channel to share coin value with cryptographer on right
    channel_0 <- c.coin // Use this channel to share coin value with cryptographer0
    close(channel)
    wg[0].Done() // Signal to waitgroup of completion
}

// Compare function
func (c *Crypto) compare(left_coin bool, channel_out chan bool, channel_0 chan bool) {
    if c.paid {
            c.xorval = reverse_xor(c.coin, left_coin)
    } else {
            c.xorval = xor(c.coin, left_coin)
    }
    if c.xorval {
            fmt.Println(c.identity," declares outcome: Different")
    } else {
            fmt.Println(c.identity," declares outcome: Same")
    }
    channel_out <- c.xorval // Use this channel to share outcome with Restaurant Owner or
Observer
    channel_0 <- c.xorval // Use this channel to share outcome with Cryptographer0
    wg[1].Done() // Signal to waitgroup of completion
}

// Restaurant Owner Function
func restaurant_owner(a bool, b bool, c bool) {
    var same, diff int
    var outcome[3] bool
    outcome[0]=a // Assign A's outcome
    outcome[1]=b // Assign B's outcome
    outcome[2]=c // Assign C's outcome
    same = 0
    diff = 0
    // Count the number of Different and Same utterances
    for i :=0; i<3; i++ {
            if outcome[i] {
                    diff = diff + 1
            } else {
                    same = same + 1
            }
    }
    fmt.Println("Result Count -> ||Same:",same,"||  ||Different:",diff,"||\n")
    // If "different" utterance is even --> NSA Paid, if odd --> cryptographer paid
    if (diff % 2 != 0){
            fmt.Println("*** Odd count of \"Different\" Uttered. So.. ***")
            fmt.Println("=> Restaurant Owner declares a cryptographer paid !!")
    } else {
            fmt.Println("*** Even count of \"Different\" Uttered. So.. ***")
            fmt.Println("=> Restaurant Owner declares the NSA Paid !!")
    }
    wg[2].Done() // Signal to waitgroup of completion
}

//Cryptographer 0 Function
func crypt0(aCoin bool, bCoin bool, cCoin bool, axor bool, bxor bool, cxor bool ) {
    fmt.Println("-----------------------------------")
```

```
    fmt.Println("     ==> Cryptographer A <==       ")
    fmt.Println("-----------------------------------")
    fmt.Println("Computed Result: A xor C =>",xor(aCoin, cCoin))
    fmt.Println("Declared Result by A     =>",axor)

    fmt.Println("\n-----------------------------------")
    fmt.Println("        Cryptographer B        ")
    fmt.Println("-----------------------------------")
    fmt.Println("Computed Result: B xor A =>",xor(bCoin, aCoin))
    fmt.Println("Declared Result by B     =>",bxor)

    fmt.Println("\n-----------------------------------")
    fmt.Println("        Cryptographer C        ")
    fmt.Println("-----------------------------------")
    fmt.Println("Computed Result: C xor B =>",xor(cCoin, bCoin))
    fmt.Println("Declared Result by C     =>",cxor)
    fmt.Println("-----------------------------------\n")
    if reverse_xor(aCoin, cCoin) == axor {
            fmt.Println("*** Cryptographer A said opposite --> Cryptographer A Paid! ***")
    } else if reverse_xor(bCoin, aCoin) == bxor {
            fmt.Println("*** Cryptographer B said opposite --> Cryptographer B Paid!")
    } else if reverse_xor(cCoin, bCoin) == cxor {
            fmt.Println("*** Cryptographer C said opposite --> Cryptographer C Paid!")
    } else {
            fmt.Println("*** Nobody said opposite so The NSA paid! ***\n\n")
    }
    wg[3].Done() // Signal to waitgroup of completion
}

// XOR operation
func xor(a bool, b bool) bool {
    return a != b
}

// Reverse XOR operation
func reverse_xor(a bool, b bool) bool {
    return !(a != b)
}

func main() {

    // Channels for communication between cryptographers
    channel_A := make(chan bool, 1) // Channel for Crytographer A's coin shared to B
    channel_B := make(chan bool, 1) // Channel for Crytographer B's coin shared to C
    channel_C := make(chan bool, 1) // Channel for Crytographer C's coin shared to A
    channel_O := make(chan bool, 3) // Channel for Crytographers Outcome shared with
Restaurant Owner
    channel_0A := make(chan bool, 2) // Channel shared by Crytographers 0,A
    channel_0B := make(chan bool, 2) // Channel shared by Crytographers 0,B
    channel_0C := make(chan bool, 2) // Channel shared by Crytographers 0,A

    //Get random seed as current time
    rand.Seed(time.Now().UTC().UnixNano())

    // Set Defaults
    a := Crypto{paid: false, identity: "Cryptographer A"}
    b := Crypto{paid: false, identity: "Cryptographer B"}
    c := Crypto{paid: false, identity: "Cryptographer C"}
    n := NSA{paid: false, identity: "NSA"}

    //determine payer randomly
    payer := rand.Int() % 4
```

```
    fmt.Println("-------------------------------------")
    fmt.Println("  ==> Randomly Selecting Payer <==         ")
    fmt.Println("-------------------------------------")
    if payer == 0 {
           n.paid = true
           fmt.Println("  Selected Payer:", n.identity)
    }
    if payer == 1 {
           a.paid = true
           fmt.Println("  Selected Payer:", a.identity)
    }
    if payer == 2 {
           b.paid = true
           fmt.Println("  Selected Payer:", b.identity)
    }
    if payer == 3 {
           c.paid = true
           fmt.Println("  Selected Payer:", c.identity)
    }
    fmt.Println("-------------------------------------\n")

    // All crytographers flip coins simultaneously
    fmt.Println("-----------------------------------------------")
    fmt.Println(" ==> Flip results visible to Crytographer0 <==  ")
    fmt.Println("-----------------------------------------------\n")
    wg[0].Add(3) // Add the three coin processes to WaitGroup
    go a.Flip(channel_A, channel_0A)
    go b.Flip(channel_B, channel_0B)
    go c.Flip(channel_C, channel_0C)
    wg[0].Wait() // Wait for the coin processes to complete
    fmt.Println("---------------------------------------\n")

    // All crytographers compare coins and broadcast
    fmt.Println("-----------------------------------------")
    fmt.Println(" ==> Cryptographers declare outcomes <==   ")
    fmt.Println("-----------------------------------------")
    fmt.Println("** Visible to: Observer, Restaurant Owner, Cryptographer 0,A,B,C **\n")
    wg[1].Add(3) // Add the three cryptographer processes to WaitGroup
    go a.compare(<-channel_C, channel_O, channel_0A)
    go b.compare(<-channel_A, channel_O, channel_0B)
    go c.compare(<-channel_B, channel_O, channel_0C)
    wg[1].Wait() // Wait for the cryptographer processes to complete
    fmt.Println("---------------------------------------\n")
    fmt.Println("-----------------------------------------")
    fmt.Println(" ==> Restaurant Owner computes payer <==    ")
    fmt.Println("-----------------------------------------")

    //Restaurant Owner Process
    wg[2].Add(1) // Add the restaurant owner process to WaitGroup
    go restaurant_owner(<-channel_O,<-channel_O,<-channel_O)
    wg[2].Wait() // Wait for the restaurant processes to complete
    fmt.Println("---------------------------------------\n")
    fmt.Println("---------------------------------------")
    fmt.Println(" ==> Cryptographer0 verifies payer <== ")
    fmt.Println("---------------------------------------")

    // Cryptographer0 Process
    wg[3].Add(1) // Add the cryptographer 0 process to WaitGroup
    go crypt0(<-channel_0A , <-channel_0B, <-channel_0C, <-channel_0A , <-channel_0B, <-
channel_0C)
    wg[3].Wait() // Add the cryptographer 0 process to complete
}
```