Individual Diploma Thesis

DETECTING GO LANGUAGE FEATURES BY ANALYZING GO BINARIES

Frixos Kallenos

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2024

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Detecting Go Language Features by Analyzing Go Binaries

Frixos Kallenos

Supervisor

Dr. Elias Athanasopoulos

The Individual Diploma Thesis was submitted for partial fulfilment of the requirements for obtaining a degree in Informatics of the Department of Informatics of the University of Cyprus

May 2024

Acknowledgements

I would like to express my deepest gratitude to those who have supported me throughout the journey of writing this thesis and especially my supervisor, Dr. Ilias Athanasopoulos.

A special mention goes to my family and my girlfriend, whose presence in my life has been a source of strength and inspiration. Your patience, understanding, and love have been invaluable, especially during the times when my health challenges seemed insurmountable. You stood by me, providing the emotional and practical support that I needed to persevere.

Abstract

Memory corruption vulnerabilities are one of the most significant categories of vulnerabilities in information security. Such vulnerabilities commonly arise from programming errors or flaws in software development practices, particularly when using programming languages that permit unrestricted memory access, such as C or C++. To counter memory corruption attacks, numerous techniques and defense mechanisms have been devised. Despite the existence of these hardening techniques, numerous attack vectors remain effective. Programming languages like Rust and Go were developed to address those vulnerabilities, while being fast and efficient, providing low-level capabilities and being type-safe and memory-safe. Nevertheless, these languages are not entirely immune to vulnerabilities; they have weaknesses when not used properly. Developing security mechanisms for these languages necessitates tools capable of analyzing programs produced in these languages at the binary level. This thesis will concentrate on the Go programming language, particularly on disassembling its stripped binaries and identifying potential utilization of features available specifically in Go, such as goroutines, channels and closures.

Contents

Chapter 1

Introduction

One of the most serious class of vulnerabilities in information security, is memory corruption vulnerabilities like buffer overflows, buffer overreads and use-after-free vulnerabilities. According to MITRE rankings, memory corruption errors are currently one of the three most dangerous software errors [1]. Even major browsers are being successfully exploited due to memory corruptions, in annual hacking contests like Pwn2Own or Pwnium [2]. This kind of vulnerabilities typically arise due to programming errors or flaws in the way software is developed, especially when using programming languages that allow unrestricted memory access, like C or C++. Although having unrestricted memory access while coding poses significant security risks, it provides a lot of benefits like efficiency, since it allows developers to have fine-grained control over memory management, enabling them to optimize performance by allocating and deallocating memory precisely as needed. Building operating systems, device drivers, and other system-level software, as well as maintaining compatibility with some legacy systems, are some of the reasons other than efficiency, that make unrestricted memory access necessary.

In order to defend against memory corruption attacks, multiple techniques and defense mechanisms have been developed. Software hardening techniques include Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), exception handler validation (SafeSEH), Control-flow Integrity (CFI). Even with those hardening techniques in existence, numerous attack vectors remain effective. That's either because attackers are continuously adapting and refining their attack strategies, or because most hardening techniques are not used in practice due to high performance overhead or not compatible with all used features. Yet another factor hindering the defensive measures, is the hurdle of manually modifying the source code to apply protection or make it compatible with existing code [2]. Note that none of the current methods offer assured protection since they do not fully solve the general problem of memory corruption.

Programming languages like Rust and Go were developed to address those vulnerabilities, while being fast and efficient, providing low-level capabilities and being type-safe and memory-safe [3]. Rust and Go are growing in popularity; IEEE's 2023 Top Programming languages list ranks them 18th and 8th respectively [4], while C and C++ maintain their positions in the top spots, occupying the third and fourth. Code written in Rust or Go can seamlessly coexist with code written in C/C++. This feature enhances the appeal of these languages, as various components of an unsafe program can be delegated to Rust or Go, consequently enhancing overall safety [5].

Go language ensures memory safety by incorporating a minimal runtime support with a lightweight garbage collector and by implementing bounds checks that are activated during runtime. It has also built-in support for parallel programming using goroutines (lightweight threads managed by the Go runtime instead of the OS) and channels. This simplifies the process of writing concurrent code, avoiding the complexities and potential security issues linked to traditional threading models. Golang features a race detector tool that aids developers in pinpointing and resolving data race conditions during the development phase. This significantly contributes to the creation of more robust and secure parallel programs. By tackling common sources of vulnerabilities at the language level, these programming languages aid in preventing a broad spectrum of security issues that have historically affected systems written in languages such as C and C++.

This thesis will focus on the Go programming language, particularly on disassembling its stripped binaries and detecting programming features specific to Go language, such as the use of goroutines, go channels or go closures.

Chapter 2

Background

2.1 Binary Analysis

2.1.1 General

Binary analysis is the scientific examination of the characteristics of binary computer programs, referred to as binaries, along with the machine code and data embedded within them. In simple terms, the main objective of binary analysis is to detect (and potentially alter) the authentic properties of binary programs — essentially, what they really do in contrast to what we think they should do [6]. While many people connect binary analysis with reverse engineering and disassembly, its scope extends beyond those two. Binary analysis techniques can be categorized into static analysis, dynamic analysis, or a combination of these classes. With static analysis, we analyze a binary without ever executing it. We can analyze the whole binary at once, even on a machine of a different architecture than the one for which the binary was compiled. However, we won't have any knowledge of the binary's runtime state. For example, it's very difficult to calculate indirect branches. On the other hand, with dynamic analysis we run the binary and we analyze it as it executes. This approach is usually more straightforward than static analysis because it involves complete awareness of the entire runtime state, including variable values and the results of conditional branches. Nevertheless, it only

provides visibility into the executed code, so it’s possible for the analysis to overlook significant sections of the program.

2.1.2 Challenges

Analyzing binaries is challenging and considerably more demanding than analyzing the equivalent source code. Some binary analysis tasks are even inherently undecidable, making it impossible to build a solution that consistently produces accurate results [6].

During software development we give our constructs (variables, functions, classes) names that have meaning, and we also follow some naming conventions to make the source code more readable. This symbolic information (debugging symbols or symbols) serves no real purpose when the code is compiled. Therefore, it's common practice to strip binaries in production software, namely to remove debugging symbols and other metadata, to reduce the file size and improve performance. It also makes sense that malwares are also stripped, since they need to give away as little information as possible. Without symbols, understanding the code is significantly more challenging.

High-level source code often includes well-defined types, such as int, double, char, or even more complex data structures like struct types. However, it’s extremely hard to infer the purpose and structure of data at the binary level because types are never explicitly specified. At the binary level it’s also difficult to recognize any high-level abstractions like classes and functions since these high-level constructs are being discarded during the compilation process.

Most compilers mix fragments of data with the executable code. This makes it prone to inadvertently interpreting these data fragments as code, or the other way around, leading to false conclusions.

Executable code and data are location-dependent inside a binary file. Making any form of code or data modification is exceptionally hard and carries a risk of breaking the binary. This is because any modification is likely going to shift other code or data around, making memory addresses and references from elsewhere in the code invalid.

2.2 Assembly Code

Considering all mentioned above challenges for binary analysis, assembly code has a significant role in that process. Assembly language serves as a bridge between the human analyst and the complex machine code of a binary, providing a human-readable interface that facilitates profound insights into program behavior, debugging, and vulnerability analysis. There is a one-to-one correspondence between assembly language instructions and machine code instructions executed by a CPU, allowing analysts to understand and interpret the binary's behavior at a level close to hardware. Assembly is essential for debugging binaries, setting breakpoints, and examining the state of the program, to identify bugs or vulnerabilities.

2.3 Disassembly

Disassembling a binary refers to the process of converting machine code, which is the binary representation of executable instructions understood by a computer's central processing unit (CPU), into assembly language. The disassembly process involves analyzing the static binary file or runtime information and extracting the sequence of instructions that make up the program. These instructions are then presented in a symbolic and mnemonical form that is easier for humans to understand (assembly code). The result is a textual representation of the program's executable code.

2.3.1 Static Disassembly

In static disassembly, the instructions are extracted from a binary without executing it. The objective of every static disassembler is to convert all code in a binary into a human-readable format or a format that can be processed by machines for further analysis [6]. To accomplish this objective, static disassemblers need to execute the following steps: (1) Load the binary with a binary loader, (2) Identify all machine instructions in the binary, (3) Disassemble the identified instructions into assembly code. The tricky part is the second step because, as we mentioned in binary analysis challenges, code is mixed with data within a binary and so if the disassembler misidentify data as code, there is a great chance they could correspond to valid instructions. Two primary approaches exist for static disassembly, linear and recursive

disassembly. Linear disassembly passes through each one of the code segments and decodes all bytes consecutively translating them to assembly instructions. In contrast, recursive disassembly starts from known entry points into the binary and follows recursively the control flow (calls and jumps) from there. The disadvantage of recursive disassembly is that it's not easy to follow every branch, especially indirect jumps, or calls.

2.3.2 Dynamic disassembly

Dynamic disassembly avoids a lot of dangers associated with code being mixed with data, because it relies on a lot of runtime information such as memory and register contents. Therefore, it can disassemble all executed instructions and be certain that there is no data there. The primary drawback of this approach is the code coverage problem, where dynamic disassemblers only encounter and analyze instructions that are executed, rather than all instructions present in the code.

2.4 Reverse Engineering

Reverse Engineering is the understanding of the internals of something made by a human, through analysis, without having access to its design principles and the way its components interact to complete its intended goal. In other words, it's the process of taking apart something that someone else built and understand how he did it. In the context of computer science, we usually refer to software reverse engineering, which is typically done by analyzing the assembly code generated from disassembling the binary under examination. The assembly code is then analyzed using various tools like IDA Pro, Ghidra and OllyDbg, which can perform some automated analysis and provide important insights to the reverse engineer. Normally, a lot of manual effort is required by the reverse engineer, to overcome all the challenges described in section 2.1, plus any anti-reversing tricks or obfuscation techniques imposed by the binary. Anti-Reversing tricks constitute one the main classes of countermeasures designed to combat reverse engineering. These tricks include, amongst others, direct or indirect debugger detection, virtual machine detection, parent process detection and execution time detection. On the other hand, code obfuscation is not exactly an anti-reversing technique, because its primary target is to challenge the human behind the tool and not

the tool itself. For example, logic flow obfuscation complicates the prediction and comprehension of when a program execution should reach conditional branches.

```
test eax, eax
je _eaxWasZero
```

Regular compiled code: If `eax==0` then we jump.

```
and eax, 0xffffffff
je _eaxWasZero
```

Obfuscated code: logical AND operation with all its bits set. If the result is zero, `eax` was zero, so we jump.

The above obfuscated code produces the same effect as the regular compiled code, but its purpose is to slow down the analyst by forcing them to analyze unexpected assembly code before a conditional jump. There is also a technique called NOP obfuscation where a set of instructions added to the binary have no real impact on the execution of the code.

```
push ebx
add ebx, ecx
sub ebx, eax
push eax
sub eax, edx
xor eax, edi
pop eax
pop ebx
```

NOP obfuscation: The above set of instructions has a NOP effect.

2.5 Linkage Phase

The linking phase is the last stage of the compilation process. During this phase, all the object files are linked into a single coherent binary executable [6]. As expected, the tool responsible for executing the linking phase is referred to as a linker or link editor. Typically, it is distinct from the compiler, which implements all preceding phases. Object files are considered relocatable because they are compiled independently from each other. This independence prevents the compiler from assuming that an object will end up at any particular base address. There are two main types of linking: static linking and dynamic linking. Static libraries are incorporated directly into the binary

executable. This integration enables the resolution of any references to them entirely within the executable. The addresses at which dynamic libraries will reside are not known during the linking phase. Therefore, references to them cannot be resolved at this stage. Instead, the linker leaves symbolic references to these libraries in the final executable. These references remain unresolved until the binary is loaded into memory for execution.

2.6 Go Language

The Go language (or Golang) is a relatively new programming language, designed with systems programming in mind, developed by Google. The language offers a statically typed and compiled environment, incorporating modern language features while maintaining a straightforward syntax. According to its official documentation “Go is expressive, concise, clean, and efficient” [7]. Its concurrency mechanisms facilitate the development of programs optimized for multicore and networked machines, while its innovative type system allows for flexible and modular program construction. Go compiles quickly to machine code, while also having the convenience of garbage collection and the capabilities of run-time reflection. It is a fast, statically typed, compiled language that provides a sensation similar to dynamically typed, interpreted languages. While it incorporates concepts from existing languages, Go possesses distinctive properties that result in effective Go programs having a different character compared to programs written in its language relatives. It’s important for Go developers to understand Go’s properties and idioms.

2.6.1 The Go Garbage Collector

The Go language is responsible for organizing the storage of Go values, relieving Go developers from the need to be concerned about where these values are stored or why. These values frequently need to be stored in the physical memory of a computer, which is a finite resource and that’s why memory must be managed with care and recycled to prevent depletion during the execution of a Go program [8]. A garbage collector (GC) is a system that efficiently recycles memory on behalf of the application by identifying portions of memory that are no longer in use. The Go standard toolchain includes a runtime library with each application. This runtime library incorporates a garbage

collector. Go values stored in local variables usually are not managed by the Go garbage collector at all. This is because the Go compiler can predetermine when the associated memory can be freed and generate machine instructions for garbage collection that handle the cleanup. When the Go compiler cannot determine the lifetime of Go values, they are said to *escape to the heap*. The process of reserving memory on the heap is commonly referred to as "dynamic memory allocation". GC is a system designed to specifically identify and clean up dynamic memory allocations. Whether a Go value escapes or not depends on the context in which it is used, and the escape analysis algorithm employed by the Go compiler. Attempting to predict whether a value escapes can be extremely challenging, given that the escape analysis algorithm is quite sophisticated and may undergo changes in different Go releases.

2.6.2 Go statements – goroutines

A "go" statement starts the execution of a function call as an independent concurrent thread of control, referred as goroutine, within the same address space [9].

```
go <expression>
```

The expression must be a function or method call and it cannot be parenthesized. Built-in function calls are restricted.

```
var wg sync.WaitGroup

func count(id string) {
        for i := 1; i <= 10; i++ {
                fmt.Println("goroutine id: "+id+", count:", i)
                time.Sleep(time.Second)
        }
        wg.Done()


func main() {
        wg.Add(3)

        go count("1")
        go count("2")
        go count("3")

        wg.Wait()
```

```
}
```

The above code snipped initiates two concurrent independent goroutines which run in parallel and they both execute the count() function.

The function value and parameters are evaluated in the usual manner in the calling goroutine. However, unlike a regular function call, program execution does not wait for the invoked function to complete. Instead, the function begins executing independently in a new goroutine. That's why we used some thread synchronization functions from the "sync" package in the above example. When the function terminates, its associated goroutine also terminates. If the function produces any return values, they are discarded upon the function's completion. Therefore, if we need a return value or any type of information from the goroutine's execution, we must use channels.

2.6.3 Go Channels

A Go channel serves as a mechanism for concurrently executing functions to communicate by exchanging values of a specified element type [10]. Channels function as first-in-first-out queues. For instance, if one goroutine sends values on a channel while another goroutine receives them, the values are received in the order in which they were sent. It's important to note that the value of an uninitialized channel is nil. The optional <- operator designates the channel direction, either send or receive. When a direction is specified, the channel becomes directional; otherwise, it remains bidirectional. A channel can be restricted to only send or only receive through assignment or explicit conversion.

```
chan T          // can be used to send and receive values of type T
chan<- int      // can only be used to send ints
<-chan float64  // can only be used to receive float64s
```

Using the built-in function make, a new initialized channel value can be created. The channel type and an optional capacity are given as arguments to this function.

```
ch := make(chan int, 100)
```

The capacity, denoting the number of elements, determines the size of the buffer in the channel. In the case of a capacity being zero or absent, the channel is unbuffered, and communication only succeeds when both a sender and receiver are ready. Conversely, if the channel has a capacity, it becomes buffered, allowing communication without blocking if the buffer is not full for sends or not empty for receives. When a channel is nil, it's never ready for communication. A channel can be closed using the built-in function close.

A value can be sent on a channel by a send statement, provided that the channel expression's core type must be a channel, the channel direction must permit send operations, and the type of the value to be sent can be assigned to the channel's element type. In the process of communication, both the channel and the value expression are evaluated before the transmission commences. Communication blocks until the send operation can proceed. Specifically, a send on an unbuffered channel can proceed if a receiver is prepared to receive. Meanwhile, a send on a buffered channel can proceed if there is available space in the buffer. Attempting to send on a closed channel results in a run-time panic and attempting to send on a nil channel leads to indefinite blocking.

```
ch <- 7  // send value 7 to channel ch
```

For an operand ch whose core type is a channel, the value of the receive operation <-ch is the value received from the channel ch. It's essential that the channel direction allows receive operations, and the type of the receive operation is the same as the element type of the channel. The expression blocks until a value becomes available. Attempting to receive from a nil channel results in indefinite blocking. On the other hand, a receive operation on a closed channel can always proceed immediately, providing the element type's zero value after any previously sent values have been received.

```
v1 := <-ch
v2 = <-ch
```

Receive operator examples.

A receive expression used in an assignment provides an additional untyped boolean result indicating whether the communication was successful.

```
x, ok = <-ch
x, ok := <-ch
var x, ok = <-ch
var x, ok T = <-ch
```

The value of `ok` is true if the received value resulted from a successful send operation to the channel. If it is false, it indicates a zero value generated because the channel is closed and empty.

2.6.4 Go Closures

Go functions have the capability to be closures. A closure is a function value that is able to reference variables from outside its body. The function can access and modify the referenced variables; in this context, the function is considered "bound" to the variables. Closures are essentially formed by a special type of anonymous function.

```
import "fmt"

func adder() func(int) int {
      sum := 0
      return func(x int) int {
            sum += x
            return sum
      }
}

func main() {
      pos, neg := adder(), adder()

      arr := [10]int{1,-2,4,-5,8,4,-4,1,-2,5}

      for i := 0; i < 10; i++ {
            if arr[i] > 0{
                  fmt.Println(“pos sum:”, pos(arr[i]))
            }else{
                  fmt.Println(“neg sum:”, neg(arr[i]))
            }
      }
}
```

In this example, the `adder` function returns a closure. Each closure is bound to its own `sum` variable. That’s why, as we iterate the array `arr`, the sum variable persists its value, for each closure separately, through multiple calls of the closure. The sum of positive numbers and sum of negative numbers of the array, are being printed.

Chapter 3

Investigative Process

Our primary objective was to understand how various golang features like goroutines, channels and closures are implemented at the binary level. To accomplish this, we created many mock go programs, disassembled them and analyze the assembly code. In our analytical procedure, we intentionally kept debugging symbols within the various binary executables to facilitate the correlation between assembly code and corresponding source code, thereby assisting the identification of patterns. However, note that our primary objective is developing tools, capable of effectively parsing and analyzing stripped binaries devoid of any symbolic information.

In the disassembling process, we used the objdump from GNU developer tools and the go objdump tool, which is a command line tool included with the Go installation. Both tools produce nearly identical output. The go objdump tool lacks the capability to disassemble stripped binaries, unlike the GNU objdump. Nonetheless, as we have mentioned, we didn't analyze stripped binaries at this phase. The following figures illustrate the output from the GNU objdump.

By default, the go compiler produces statically compiled binaries, resulting in static linking against libraries, thereby including all their code into the binary. Consequently, this leads to an increase in the size of the binaries. We kept the default behavior of the go compiler, thus analyzing statically linked binaries.

3.1 Goroutines

Every Go binary, including a simple “Hello, World!” program with no additional functionality, initiates a minimum of five (5) goroutines. Among these, one executes the main function while the remaining ones handle garbage collection tasks. Developers can create additional goroutines as needed. Our investigation specifically targets user-created goroutines and excludes those provided by the default Go runtime package.

We employed the go program presented in section 2.6.2, which spawns three goroutines, each invoking the counter function. Upon executing the compiled program, the output will resemble the following:

```
goroutine id: 3, count: 1
goroutine id: 1, count: 1
goroutine id: 2, count: 1
goroutine id: 2, count: 2
goroutine id: 3, count: 2
goroutine id: 1, count: 2
goroutine id: 1, count: 3
goroutine id: 2, count: 3
goroutine id: 3, count: 3
...
```

By examining the assembly code produced from objdump, we noticed that for each goroutine creation there was a call to the newproc() function, from the runtime package which belongs to the standard library of Go. A memory address is passed as a parameter through $rax for each call to the newproc(). Upon inspecting the source code of newproc() [11], we observe that it takes a function pointer as argument. Notably, this function pointer designates the entry point from which the newly instantiated goroutine will start its execution.

```
// Create a new g running fn.
// Put it on the queue of g's waiting to run.
// The compiler turns a go statement into a call to
this.
func newproc(fn *funcval) {
      gp := getg()
      pc := getcallerpc()
      systemstack(func() {
            newg := newproc1(fn, gp, pc)
            pp := getg().m.p.ptr()
            runqput(pp, newg, true)
            if mainStarted {
```

```
                wakep()
            }
        })
}
```

Source code of `newproc` function from go runtime package.

The go instruction to create a goroutine, as seen in 2.6.2, is "go <function_name>", thus we expected that the function pointer given to `newproc()` should be the address of the function used in this instruction. Counterintuitively, that wasn't the case, since the `newproc()` function gets the address of a new function created by the compiler, which in turn calls the initial function used in the instruction.

For example, the source code in 2.6.2., creates the goroutines using "`go count()`". In the assembly code, for each created goroutine, the `newproc()` function gets the address of a distinct compiler created function (e.g. `main.func1`, `main.func2`, etc) which has in its code a `call` instruction to the `count()` function.

After analyzing multiple other binaries, we also discovered that the go compiler, generates one of two distinct assembly code patterns, which execute preceding the invocation of the `newproc()` function. That is where the `$rax` register will get its value to be passed as an argument.
The first pattern is just a `lea` instruction that loads the function pointer address from the .rodata section to `$rax` and a then a call to `newproc()` right after. The second assembly code pattern undertakes the following sequence of actions: (1) call of the `newobject()` function from the runtime package, which allocates some memory and returns a pointer to it in `$rax`, (2) the address of the compiler generated function is loaded in `$rcx`, (3) the value of `$rcx` is moved in the memory of the object created in step 1, (4) various instructions probably about garbage collection are executed and (5) the `newproc()` function is called, while `$rax` points to the memory address of the object that contains the compiler generated function.

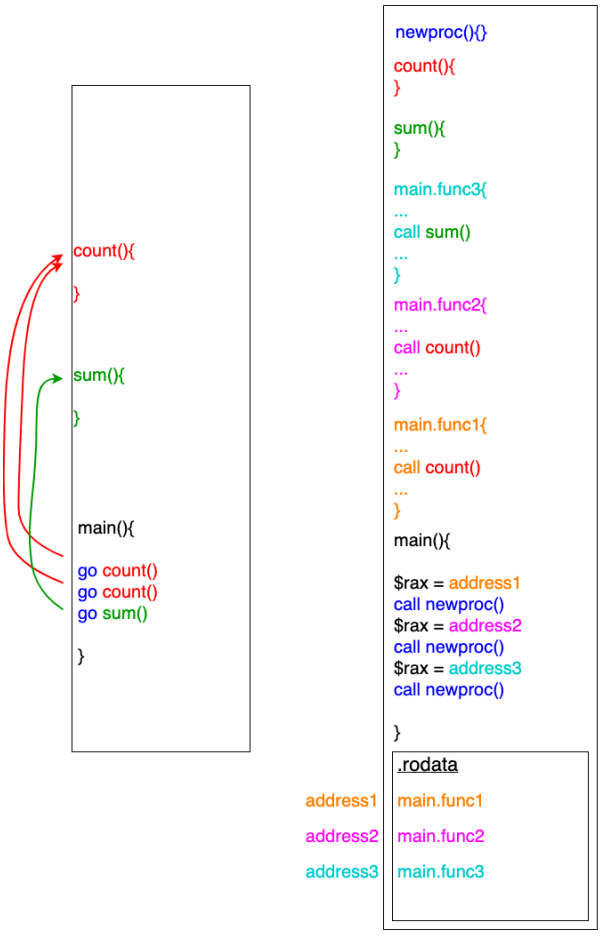
```
lea    0x1e7fc(%rip),%rax        # 49cda8 <go.func.*+0x24b>
callq  43a9a0 <runtime.newproc>
lea    0x1e7f8(%rip),%rax        # 49cdb0 <go.func.*+0x253>
callq  43a9a0 <runtime.newproc>
lea    0x1e7f4(%rip),%rax        # 49cdb8 <go.func.*+0x25b>
callq  43a9a0 <runtime.newproc>
```

First assembly code pattern before calling `newproc()`

```
callq  40bdc0 <runtime.newobject>
lea    0x63(%rip),%rcx        # 47e180 <main.main.func1>
mov    %rcx,(%rax)
cmpl   $0x0,0xd1fc9(%rip)        # 5500f0 <runtime.writeBarrier>
jne    47e134 <main.main+0x54>
mov    0x10(%rsp),%rcx
mov    %rcx,0x8(%rax)
jmp    47e145 <main.main+0x65>
lea    0x8(%rax),%rdi
mov    0x10(%rsp),%rcx
nopl   (%rax)
callq  45aa60 <runtime.gcWriteBarrierCX>
callq  43a9a0 <runtime.newproc>
```

Second assembly code pattern before calling `newproc()`

Following an extensive analysis of go binaries that spawn goroutines, it has come to our attention that the second abovementioned assembly code pattern, emerges when Go channels are employed, for the communication among goroutines by exchanging values. The first assembly code pattern is generated when channels aren't used.



3.2. Channels

After examining goroutines, our focus shifted to Go channels, which enable communication among different goroutines. Once again, we constructed mock Go binaries and disassembled them, allowing us to scrutinize how the Go compiler implements channels at the assembly level.

```
package main

import (
      "fmt"
)

func get_func(ch <-chan int) {
      var x int = <-ch
      fmt.Println(x + 2)
}

func give_func(ch chan<- int) {
      var y int = 3
      ch <- y
}

func main() {
      ch := make(chan int, 2)
      go give_func(ch)
      get_func(ch)
}
```

The above go program creates a channel of type `int` with buffer size two. It has two functions that get the created channel as argument and use it to communicate (send/receive values). The `get_func()` is called by the main goroutine, and the `give_func()` is called from a new goroutine that runs in parallel with the main one. The `give_func()` creates a local variable and sends its value through the channel, while the `get_func()` receives the value from the channel, adds two and prints it (the number 5 is printed).

At the binary level, the implementation of go channels relies on three functions from the runtime package of the Go standard library, akin to the utilization of `newproc()`

function for goroutines. Those three functions are: (1) `makechan()` which is used initially to create a new channel, (2) `chansend1()` which is used to send variables into a channel and (3) `chanrecv1()` which is used to receive variables from a channel.

The initial step involves creating the channel by invoking the `makechan()` function, which returns a pointer to it. Essentially, this function initializes an `hchan struct` [12].

```
type hchan struct {
      qcount   uint           // total data in the queue
      dataqsiz uint           // size of the circular queue
      buf       unsafe.Pointer // points to an array of dataqsiz
elements
      elemsize uint16
      closed   uint32
      timer    *timer // timer feeding this chan
      elemtype *_type // element type
      sendx    uint   // send index
      recvx    uint   // receive index
      recvq    waitq  // list of recv waiters
      sendq    waitq  // list of send waiters

      // lock protects all fields in hchan, as well as several
      // fields in sudogs blocked on this channel.
      lock mutex
}
```

At the assembly level, when a channel is used to send some value, the chansend1() function is invoked. The channel pointer returned by makechan() is given as an argument via the $rax register. The value to be send is loaded onto the stack, and a pointer to it is also provided as an argument, via the $rbx register. Similarly, when a channel is utilized to receive a value, the channel pointer is given as an argument to the chanrecv1() function through $rax. The received value is obtained through stack after chanrecv1() is returned.

3.3. Closures

Similar to our previous examinations of Goroutines and Go channels, we studied the assembly code generated by simple Go programs incorporating closures. We identified

two distinct approaches through which the Go compiler facilitates the creation of closures at the assembly level. The selection of an approach by the compiler is contingent upon whether the closure (or the function that creates it) invokes any library functions, like for example fmt.Println() or math.Pow().

In the first approach, the compiler generates a distinct function for every created closure. Here, "created" refers to each invocation of the function that returns a closure (in the high level go source code). At the assembly level, the function that returns the closure is not created as a separate function, probably as an optimization strategy. The address of the closure is pushed onto the stack, followed by variables bound to the closure, in the subsequent stack positions. Anytime the program wants to call the closure, it does the following: (1) moves the closure address from the stack to $rcx, (2) moves the closure argument to $rax, (3) loads the address of stack containing the address of the closure to $rdx, and (4) executes an indirect call instruction to the closure, e.g. call $rcx. Of course, the registers may differ depending on the number of the closure's arguments, although $rdx seems to always get the stack address where the closure is stored.

```
import "fmt"

func adder(x int) func(int) int {
      var base int = x

      return func(y int) int {
            return base + y
      }
}

func main() {
      var add5 func(int) int
      var add10 func(int) int

      add5 = adder(5)
      fmt.Println(add5(2))
      fmt.Println(add5(3))

      add10 = adder(10)
      fmt.Println(add10(2))
      fmt.Println(add10(3))
}
```

In this code snipped, the closure does not invoke any library functions, so the compiler will select the first approach. There are two calls of the adder function which returns a closure, so the compiler would generate two functions.

```
<main.main.func2>:
47e1a0:  add    0x8(%rdx),%rax
47e1a4:  retq

<main.main.func1>:
47e1c0:  add    0x8(%rdx),%rax
47e1c4:  retq
```

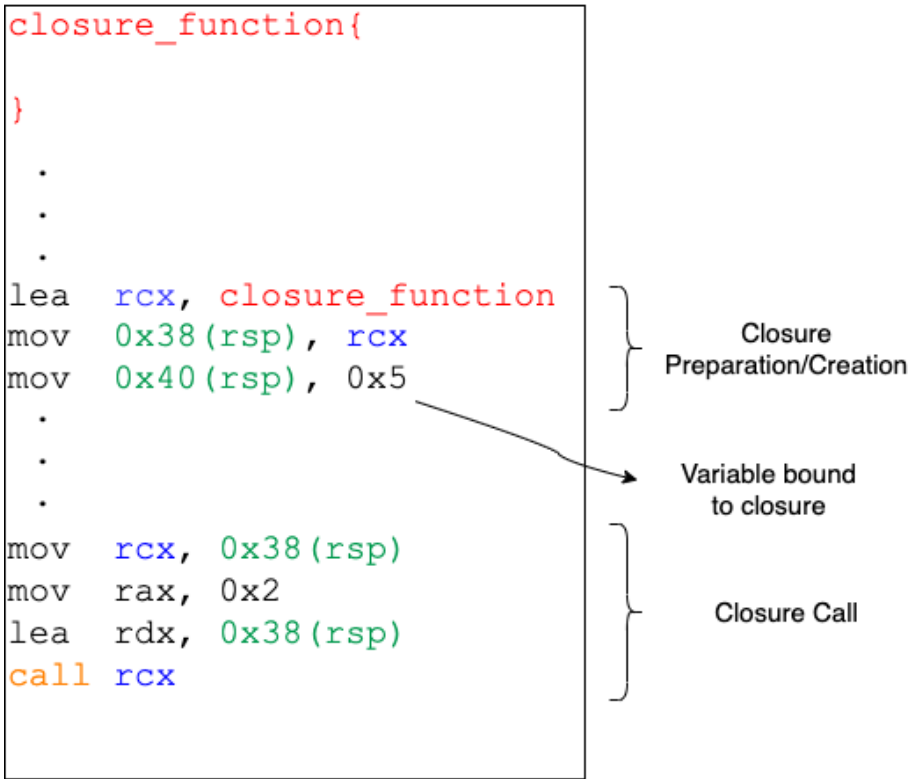
The two closures as they are implemented by the compiler.

```
47e00c:  lea    0x1ad(%rip),%rcx      # 47e1c0 <main.main.func1>
47e013:  mov    %rcx,0x38(%rsp)
47e018:  movq   $0x5,0x40(%rsp)
```

The assembly code for creating the first closure. The closure's address is moved to the stack and then the number 5, which is the argument for the adder function, is also placed into the subsequent stack position. The closure is bound to the base variable, which gets the value of the argument.

```
47e021:  mov    0x38(%rsp),%rcx
47e026:  mov    $0x2,%eax
47e02b:  lea    0x38(%rsp),%rdx
47e030:  callq  *%rcx
```

This is the assembly code for invoking a closure. The closure's address is retrieved from the stack to $rcx. Additionaly, $eax gets the closure's argument and $rdx gets a pointer to the stack address where the closure, namely its address and the variables that is bound to, are stored. Lastly, we have an indirect call to the closure.



In the second approach, the compiler generates only a single function, regardless of the number of closures created. Unlike the first approach, the function that returns the closure, is distinctively generated in assembly. The closure's address and the variables that the closure is bound are not passed onto the stack like the first approach. All these are passed, with the same order as the first approach, on the memory of an object, allocated with newobject(). In this case, whenever the closure needs to be invoked, the same process is followed as the first approach, with the distinction that the closure's address and the other variables are retrieved via the object's memory rather than the stack.

Chapter 4

Implementation

4.1. Raw bytes of specific addresses

We created a python script (elf2bytes.py) that takes as input an ELF binary, a start addresses and an end address and prints the raw bytes of all the instructions between those two addresses. Of course, this could be done with other tools as well, but we wanted the output in a specific format, so after making minimal changes, it can be inserted in the code of other python scripts we developed. We needed to use the “elftools” and “capstone” python libraries. Elftools contains functions and data structures for analyzing ELF files, while capstone is a lightweight multi-platform and multi-architecture disassembly framework.

The script initially opens the given ELF file, iterates through its sections until it finds the .text section and then reads all the bytes of that section. Using some functions from the capstone framework, it creates a disassembler object and then disassembles all the bytes read from the .text section. A generator object is returned, through which we can iterate all instructions and print only the bytes of the instructions between the addresses given by the user (start – end addresses). The bytes are printed in a hex escaped format.

4.2. Detecting goroutines

Since the creation of a goroutine is indicated by a call to the newproc() function from the runtime package (see 3.1), we developed a script to identify (1) the presence of this function inside an ELF binary, (2) any calls to that function, (3) the addresses from which each goroutine will start execution and (4) the actual user function that is called by each goroutine. Similar to “elf2bytes.py”, this script also uses the Elftools and Capstone libraries and can analyze stripped binaries.

Initially, we used the elf2bytes.py script with various non-stripped binaries to extract all the raw bytes of the newproc() function. We determined the exact starting and ending addresses of the function using objdump. The resulting bytes were not identical for all binaries, as the newproc() function in each binary had slightly different assembly instructions. For example, the newproc() function needs to call the systemstack() function, which is another Go runtime function. For this task, one binary might have the instruction call 0x458b20, while another binary might have call 0x458860. This occurs because the systemstack() function is not located at the same address in every binary or the call instruction is not on the same address in every binary.

In our “goroutineDetection.py” script, we used regular expressions to specify a pattern that will always match and find the newproc() function in a series of bytes. For example, the above assembly instructions call 0x458b20 and call 0x458860 are could be comprised from the bytes e8 2f e1 01 00 and e8 6f de 01 00 respectively. A regex pattern like “e8 {4}.”, where “{4}.” means any four bytes, is able to match both instructions. In that way, by studying the assembly code of the newproc() function in multiple binaries, we build a regex pattern that can identify it in any series of raw bytes.

The actual creation of a goroutine is indicated by a call to the newproc() function. Our script makes a list with all the call instructions found in the binary, by checking the mnemonic of each instruction. Then checks which of those instructions call an address where the newproc() function is detected. After that, we need to identify which of the two assembly code patterns, described in 3.1 is present, in order to detect the start address of the goroutine.

If the instruction before “call newproc()” is a lea instruction, we calculate the address that is loaded into $rax. This address stores a function pointer to the compiler-generated function, which is where the goroutine will start execution. We then retrieve the function pointer, navigate to that function, and identify the first call instruction, which invokes the actual user function.

If the instruction before “call newproc()” is not a lea instruction, there are probably channels in the binary, and thus we have the second assembly code pattern described on 3.1. Our script then checks one by one all instructions backwards until it finds a “mov qword ptr [rax], rcx” instruction. This instruction is used to move the function pointer stored in $rcx to the memory of a newly allocated object, pointed by $rax. The previous instruction should be a lea instruction loading the function pointer to $rcx. In that way our script identifies the address from which the goroutine will start execution and following that, the call instruction to the actual user function.

Since we are focusing only on goroutines invoked from user code, we wanted to ignore all default goroutines created by the go runtime package (e.g. for garbage collection). Therefore, we identify all calls to the newproc() function but do not further analyze code from libraries.

4.3. Detecting Channels

To identify the use of channels, our “channelDetection.py” script must detect the presence of the following functions: makechan(), chansend1() and chanrecv1(), along with calls to these functions (see 3.2). Using a similar approach as our goroutine detection script, we construct regular expression patterns to match these functions in any byte stream.

After identifying the addresses of each of these three functions, we procced to search all call instructions to these addresses. This allows us to determine the exact addresses where a channel is created and when a channel it is used for sending or receiving values.

Using the GDB (GNU Debugger) Python API, we enhanced our Python script with runtime analysis capabilities. The GDB Python API enables extending and automating GDB using Python scripts. For a Python script to interact with GDB, it must be executed with the following command: "gdb -x script.py".

Initially we set breakpoints at every address where we have found a call to makechan(), chansend1() or chanrecv1(). Each time a breakpoint is hit, we check which function the call instruction invokes. If it's a makechan call, we know it will return a pointer to the newly created channel through $rax, and we capture this information. If it's a chansend call, we retrieve the $rax value, which is the pointer of the channel to be used, and the $rbx value, which is the value to be sent. If it's a chanrecv call, we get the $rax value which is the pointer of the channel to be used, and we retrieve the received value from the stack. To find the stack address, we examine the instruction preceding the chanrecv call.

```
RUNTIME ANALYSIS
----------------

Channel created at address: 0x47e081
Channel pointer: 0xc000116060

Channel created at address: 0x47e094
Channel pointer: 0xc0001160c0

chansend1 call at 0x47e021
Value 0x000000000000000c sent through channel 0xc000116060

chansend1 call at 0x47e021
Value 0x0000000000000011 sent through channel 0xc0001160c0

chanrecv1 call at 0x47e20f
Value 0x000000000000000c received through channel 0xc000116060

chanrecv1 call at 0x47e231
Value 0x0000000000000011 received through channel 0xc0001160c0
```

4.4. Detecting Closures

Initially, our "closureDetection.py" script searches through the entire binary for two specific instructions appearing consecutively. As discussed in section 3.3, the instructions "lea rcx, 0x1ad(rip)" followed by "mov 0x38(rsp), rcx" suggest a potential closure preparation. The offset in both instructions can vary, and the rcx register could also be rax or rbx.
For each potential closure setup, we locate all indirect calls within the same function by searching consecutively until a ret instruction is encountered. For each of these indirect calls, we examine the preceding instruction. If it's a lea instruction that loads an address from the stack, from the same offset where the closure was stored, into rdx, we mark this indirect call as a closure call. We can then trace back to the closure preparation and calculate the closure function's address.

Our script can only identify closures that are generated with the first approach described in 3.3.. For the second approach we would need runtime analysis capabilities, since the closures address is stored and retrieved from the memory of an object allocated at runtime.

```
Possible closure set-up detected at address: 0x47e00c
Closure call at 0x47e030. Closure function at 0x47e1c0
Closure call at 0x47e080. Closure function at 0x47e1c0

Possible closure set-up detected at address: 0x47e0c5
Closure call at 0x47e0e9. Closure function at 0x47e1a0
Closure call at 0x47e136. Closure function at 0x47e1a0
```

Chapter 5

Tool Evaluation

To evaluate our tool's performance, we initially tested it by analyzing the mock binaries used during development (source code presented in Annex A). Before testing, we stripped all these binaries to confirm that the tool does not rely on the presence of symbols and can identify Go features even in stripped binaries. Disassembling the corresponding non-stripped versions was particularly helpful, as it allowed us to compare the results and measure accuracy.

Our "goroutineDetection.py" script performs really well with all mock binaries, including those designed for testing channels, as they also create goroutines. The sript can identify the newproc() function, all call instructions to it, the address where the goroutine will start execution, and the address of the actual function that is being invoked at the source code level.

Our tool for analyzing Go channels works as expected with mock binaries. Since these binaries are simple and do not depend on user interaction or input, the runtime analysis is effective and can capture values being sent and received through channels.

The Go compiler implements closures using two different approaches, as described in section 3.3. Our tool can only identify the first approach. Since when we have the second one, the closure's address is stored in the memory of an object allocated at runtime. Thus, a tool must have runtime capabilities to identify it.

We also tested our scripts against real-world applications written in Go, such as Caddy (an open-source web server) and Task (a Make alternative). While we identified many potential closures, we could not verify their validity since these binaries were stripped, preventing correlation between the source code and the equivalent assembly code. Initially, we did not identify any goroutines or channels. However, after updating our

Go compiler and adjusting our regular expression patterns, we were able to detect some goroutines and channels. Nonetheless, as with closures, we could not verify their validity.

Chapter 6

Related Work

This thesis aims to contribute to software security. In the paper "Exploiting Mixed Binaries" by Michalis Papaevripides, it was shown that mixed binaries built using C/C++ and Go are, counterintuitively, less secure than binaries built with hardened C/C++. This highlights the need for implementing defense mechanisms for Go code as well. To achieve this, tools that can analyze binaries and identify Go-specific features are necessary.

Similar work has been done by Stylianos Sofokleous, focusing on the Rust language. In his paper "Challenges in Disassembling Rust Binaries," he explored methods to identify Trait Objects, which are specific to Rust [13].

Chapter 7

Future Work

7.1. Automate Regular Expression Patterns

Our regular expression patterns to match functions like newproc() or makechan() require substantial manual work. We had to locate the exact address of the function in a non-stripped binary, extract all its raw bytes, and then manually go through each instruction to modify certain bytes to "any byte." This process could be automated to some extent.

7.2. Runtime Analysis

When a binary is run, not every instruction is executed, as some may depend on user input. Therefore, to analyze a binary at runtime, as we did when searching for Go channel utilization, we need fuzzing capabilities. This will ensure that a large percentage of the code is analyzed.

7.3 Single integrated tool

We could combine all three separate python scrips to a single tool that can analyze a Go binary and detect any Go feature.

Chapter 8

Contributions - Conclusion

In this paper, we emphasize the importance of tools for binary analysis, particularly to discover programming features unique to the Go language, for enhancing software security. We crafted several Go programs and examined their implementation at the binary level. Using this information, we developed Python scripts to identify Go features such as goroutines, channels, and closures in stripped binaries. Our tools primarily rely on static analysis, with the exception of our channel detection script, which includes some runtime capabilities. Finally, we evaluated the performance of these tools against simple stripped binaries and real-world Go applications.

If anyone attempts to develop software security defense mechanisms, these tools or our methodology may help provide valuable information about binaries.

Bibliography

[1] https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html

[2] Szekeres, L. (2017). Memory corruption mitigation via hardening and testing (Doctoral dissertation, Stony Brook University). The Graduate School, Stony Brook University.

[3] Fulton, K. R., Chan, A., Votipka, D., Hicks, M., & Mazurek, M. L. (n.d.). Benefits and drawbacks of adopting a secure programming language: Rust as a case study. University of Maryland.

[4] https://spectrum.ieee.org/the-top-programming-languages-2023

[5] Papaevripides, M., & Athanasopoulos, E. (n.d.). Exploiting mixed binaries. University of Cyprus.

[6] Andriesse, D. (2018). Practical binary analysis. No Starch Press.

[7] https://go.dev/doc/

[8] https://go.dev/doc/gc-guide

[9] https://go.dev/ref/spec#Go_statements

[10] https://go.dev/ref/spec#Channel_types

[11] https://go.dev/src/runtime/proc.go

[12] https://go.dev/src/runtime/chan.go

[13] Sofokleous, S. (2023). Challenges in disassembling Rust binaries. Computer Science Department, University of Cyprus

Annex A

Goroutine detection python script

```
import goFunctions
from elftools.elf.elffile import ELFFile
from capstone import *
import os
import sys
import re

dirname = os.path.dirname(__file__)
bin_path = os.path.join(dirname, sys.argv[1])
print("bin path:", bin_path, "\n")

f = open(bin_path, 'rb')
elffile = ELFFile(f)

class Pattern:
    def __init__(self, name, pattern):
        self.name = name
        self.pattern = pattern

def getInstructionAt(address):
    for i in instructions:
        if i.address == address:
            return i

def getPreviousInstruction(address):
    for i in range(len(instructions)):
        if instructions[i].address == address:
            if i == 0:
                print("returning -1")
                return -1
            else:
                return instructions[i-1]

def getDataAt(size, address, rawBytes, section):
    data = ""
    for i in range(size):
        byte = str(hex(rawBytes[address+i - section['sh_addr']]))[2:]
        if len(byte) == 1:
            byte = "0"+ byte
        data = byte + data
    return data

def getIP(address):
```

```
    for i in instructions:
        if i.address == address:
            return address + i.size

def getNextCallInstruction(address):
    for i in range(len(instructions)):
        if instructions[i].address == address:
            for j in range(i, len(instructions)):
                if instructions[j].mnemonic == "call":
                    return instructions[j]

def findCallInstructionsTo(address):
    instructions = disasm.disasm(code_bytes, code_section['sh_addr'])
    call_addresses = []
    for instr in instructions:
        if (instr.mnemonic == 'call' and instr.op_str == address):
            call_addresses.append(instr.address)
    return call_addresses

for section in elffile.iter_sections():
    if section.name == '.text':
        code_section = section
    if section.name == '.rodata':
        rodata_section = section

# Read the code bytes from the section
code_bytes = code_section.data()
rodata_bytes = rodata_section.data()

# Create a disassembler object
disasm = Cs(CS_ARCH_X86, CS_MODE_64)

# Disassemble the code bytes and create a list
instructions = disasm.disasm(code_bytes, code_section['sh_addr'])
instructions = list(instructions)


patterns = [
    Pattern('newproc',
b'\\x49\\x3b\\x66.\\x76.\\x48\\x83\\xec.\\x48\\x89\\x6c\\x24.\\x48\\x8d\\x6c\\
x24.\\x44\\x0f\\x11\\x7c\\x24.\\x44\\x0f\\x11\\x7c\\x24.\\x48\\x8d\\x0d.{4}\\x
48\\x89\\x4c\\x24.\\x48\\x89\\x44\\x24.\\x4c\\x89\\xf0\\x48\\x89\\x44\\x24.\\x
48\\x8b\\x44\\x24.\\x48\\x89\\x44\\x24.\\x48\\x8d\\x44\\x24.\\x48\\x89\\x04\\x
24\\xe8.{4}\\x45\\x0f\\x57\\xff\\x64\\x4c\\x8b\\x34\\x25.{4}\\x48\\x8b\\x6c\\x
24.\\x48\\x83\\xc4.\\xc3\\x48\\x89\\x44\\x24.\\xe8.{4}\\x48\\x8b\\x44\\x24.\\x
eb.'),
```

```
    Pattern('newproc',
b'\\x49\\x3b\\x66.\\x76.\\x55\\x48\\x89\\xe5\\x48\\x83\\xec.\\x44\\x0f\\x11\\x
7c\\x24.\\x44\\x0f\\x11\\x7c\\x24.\\x48\\x8d\\x0d.{4}\\x48\\x89\\x4c\\x24.\\x4
8\\x89\\x44\\x24.\\x4c\\x89\\xf0\\x48\\x89\\x44\\x24.\\x48\\x8b\\x44\\x24.\\x4
8\\x89\\x44\\x24.\\x48\\x8d\\x44\\x24.\\x48\\x89\\x04\\x24\\xe8.{4}')
]

for p in patterns:
    # Use re.finditer() to search for the pattern in the bytes
    matches = re.finditer(p.pattern, code_bytes)

    # Iterate through the matches and print the results
    for match in matches:
        start, end = match.start(), match.end()
        matched_bytes = code_bytes[start:end]
        start_address = hex(code_section['sh_addr'] + start)
        end_address = hex(code_section['sh_addr'] + end - 1)
        print(f"{p.name} function detected at address: {start_address}\n")

        call_addresses = findCallInstructionsTo(start_address)
        if len(call_addresses) == 0:
            continue
        for call_addr in call_addresses:
            print(f'call to {p.name} function detected at 0x{call_addr:x}')
            #if call_addr < 0x47dfe0:
                #print()
                #continue
            prev_instr = getPreviousInstruction(call_addr)
            prev_instr2 = getPreviousInstruction(prev_instr.address)
            if prev_instr == -1:
                continue
            mnemonic = prev_instr.mnemonic
            if mnemonic == "lea" or prev_instr2.mnemonic == "lea":
                if mnemonic != "lea":
                    prev_instr = prev_instr2
                op_str = prev_instr.op_str
                index = int(op_str[op_str.find('+ ')+1:op_str.find(']')],16)
                reg = op_str[op_str.find('[')+1:op_str.find('+')-1]
                if reg == "rip":
                    rip = getIP(prev_instr.address)
                    addr    =    getDataAt(8,rip    +    index,    rodata_bytes,
rodata_section)
                    print("Goroutine starts execution at address 0x" + addr)
                    next_call = getNextCallInstruction(int("0x"+addr,16))
                    print("User function at address " + next_call.op_str)
            else:
```

```
                found = False
                while(not found):
                    if prev_instr.mnemonic +" "+ prev_instr.op_str == 'mov
qword ptr [rax], rcx':
                        found = True
                        prev_instr                                          =
getPreviousInstruction(prev_instr.address)
                        if prev_instr == -1:
                            break
                        if prev_instr.mnemonic == "lea":
                            op_str = prev_instr.op_str
                            index          =          int(op_str[op_str.find('+
')+1:op_str.find(']')],16)
                            reg  =  op_str[op_str.find('[')+1:op_str.find('+')-
1]
                            if reg == "rip":
                                rip = getIP(prev_instr.address)
                                addr = rip + index
                                print("Goroutine starts execution at address "
+ str(hex(addr)))
                                next_call = getNextCallInstruction(addr)
                                print("User   function   at   address   "   +
next_call.op_str)

                    prev_instr = getPreviousInstruction(prev_instr.address)
                    if prev_instr == -1 or prev_instr.mnemonic == "ret":
                        break
            print("")
```

Annex B

Channel detection python script

```
import os
import sys

sys.path.append('/usr/local/anaconda3/lib/python3.8/site-packages')

import re
from elftools.elf.elffile import ELFFile
from capstone import *
import gdb

bin_path = "/path/of/binary/to/be/analyzed"

f = open(bin_path, 'rb')
elffile = ELFFile(f)

class Pattern:
    def __init__(self, name, pattern):
        self.name = name
        self.pattern = pattern

def findCallInstruction(address):
    instructions = disasm.disasm(code_bytes, code_section['sh_addr'])
    call_addresses = []
    for instr in instructions:
        if (instr.mnemonic == 'call' and instr.op_str == address):
            call_addresses.append(instr.address)
    return call_addresses

def getChanrecvStackAddr(address):
    for i in range(len(instructions)):
        if instructions[i].address == address:
            if i == 0 or i == 1:
                return 0
            else:
                instr = instructions[i-2]
                op_str = instr.op_str
                index = int(op_str[op_str.find("+")+1:op_str.find("]")],16)
                return index

def getNextInstrAddr(addr):
    for i in instructions:
```

```
        if i.address == addr:
            return addr + i.size

def getPreviousInstruction(address):
    for i in range(len(instructions)):
        if instructions[i].address == address:
            if i == 0:
                return -1
            else:
                return instructions[i-1].address

for section in elffile.iter_sections():
    if section.name == '.text':
        code_section = section
        break

disasm = Cs(CS_ARCH_X86, CS_MODE_64)

# Read the code bytes from the section
code_bytes = code_section.data()

# Create a disassembler object
disasm = Cs(CS_ARCH_X86, CS_MODE_64)

# Disassemble the code bytes and print the results
instructions = disasm.disasm(code_bytes, code_section['sh_addr'])
instructions = list(instructions)

patterns = [

Pattern("makechan",b'\\x49\\x3b\\x66.\\x0f\\x86.{4}\\x48\\x83\\xec.\\x48\\x89\
\x6c\\x24.\\x48\\x8d\\x6c\\x24.\\x48\\x8b\\x50.\\x48\\x8b\\x32\\x90\\x48\\x81\
\xfe.{4}\\x0f\\x83.{4}\\x80\\x7a.{2}\\x0f\\x87.{4}\\x48\\x89\\x54\\x24.\\x48\\
x89\\xd8\\x48\\xf7\\xe6\\x0f\\x80.{4}\\x48\\xba\\xa0\\xff\\xff\\xff\\xff\\xff\
\x00\\x00\\x48\\x39\\xd0\\x0f\\x87.{4}\\x0f\\x1f\\x44\\x00\\x00\\x48\\x85\\xdb
\\x0f\\x8c.{4}\\x48\\x89\\x5c\\x24.\\x48\\x85\\xc0\\x0f\\x84.{4}\\x48\\x8b\\x5
4\\x24.\\x48\\x83\\x7a.\\x00\\x75.\\x48\\x83\\xc0.\\x31\\xdb\\xb9.{4}\\xe8.{4}
\\x84\\x00\\x48\\x8d\\x50.\\x83\\x3d.{4}\\x00'),

Pattern("makechan",b'\\x49\\x3b\\x66.\\x0f\\x86.{4}\\x55\\x48\\x89\\xe5\\x48\\
x83\\xec.\\x48\\x8b\\x50.\\x48\\x8b\\x02'),
    Pattern('chansend1',
b'\\x48\\x83\\xec.\\x48\\x89\\x6c\\x24.\\x48\\x8d\\x6c\\x24.\\xb9.{4}\\x48\\x8
b\\x7c\\x24.\\xe8.{4}\\x48\\x8b\\x6c\\x24.\\x48\\x83\\xc4.\\xc3'),
```

```
    Pattern('chansend1',
b'\\x55\\x48\\x89\\xe5\\x48\\x83\\xec.\\xb9.{4}\\x48\\x8b\\x7c\\x24.\\xe8.{4}\
\x48\\x83\\xc4.\\x5d\\xc3'),
    Pattern('chanrecv1',
b'\\x48\\x83\\xec.\\x48\\x89\\x6c\\x24.\\x48\\x8d\\x6c\\x24.\\xb9\\x01\\x00\\x
00\\x00\\xe8.{4}\\x48\\x8b\\x6c\\x24.\\x48\\x83\\xc4.\\xc3'),
    Pattern('chanrecv1',
b'\\x55\\x48\\x89\\xe5\\x48\\x83\\xec.\\xb9.{4}\\xe8.{4}\\x48\\x83\\xc4.\\x5d\
\xc3')
]

#RUNTIME ANALYSIS
print("STATIC ANALYSIS")
print("----------------\n")

for p in patterns:
    # Use re.finditer() to search for the pattern in the bytes
    matches = re.finditer(p.pattern, code_bytes)

    # Iterate through the matches and print the results
    for match in matches:
        start, end = match.start(), match.end()
        matched_bytes = code_bytes[start:end]
        start_address = hex(code_section['sh_addr'] + start)
        end_address = hex(code_section['sh_addr'] + end - 1)
        print(f"{p.name} function detected - address: {start_address}")

        call_addresses = findCallInstruction(start_address)
        if len(call_addresses) != 0:
            for a in call_addresses:
                print(f'call to {p.name} function detected at 0x{a:x}')
        if p.name == "makechan":
            makechan_calls = call_addresses
        elif p.name == "chansend1":
            chansend_calls = call_addresses
        elif p.name == "chanrecv1":
            chanrecv_calls = call_addresses
    print("\n")


#RUNTIME ANALYSIS
print("RUNTIME ANALYSIS")
print("----------------\n")
gdb.execute("file "+bin_path, to_string=True)

limit = 0x47dfe0
```

```
# Set breakpoints
for i in range(len(makechan_calls)):
    makechan_calls[i] = getNextInstrAddr(makechan_calls[i])
for c in makechan_calls:
    if c <  limit:
        continue
    gdb.execute("break *"+str(c), to_string=True)
for c in chansend_calls:
    if c <  limit:
        continue
    gdb.execute("break *"+str(c), to_string=True)
for c in chanrecv_calls:
    if c <  limit:
        continue
    gdb.execute("break *"+str(c), to_string=True)
next_chanrecv_calls = list(chanrecv_calls)
for i in range(len(next_chanrecv_calls)):
    next_chanrecv_calls[i] = getNextInstrAddr(next_chanrecv_calls[i])
for c in next_chanrecv_calls:
    if c <  limit:
        continue
    gdb.execute("break *"+str(c), to_string=True)

bpoint_num                                                                    =
len(makechan_calls)+len(chansend_calls)+len(chanrecv_calls)+len(next_chanrecv_
calls)
numbers = ""
for i in range(bpoint_num):
    numbers = numbers + str(i+1)
    if i+1 != bpoint_num:
        numbers = numbers + " "

commands = f"""
commands {numbers}
silent
end
"""
gdb.execute(commands, to_string=True)

# Run the program
gdb.execute("run", to_string=True)

recv_channels = {}

while(True):
    try:
```

```
        rip = gdb.selected_frame().pc()
    except:
        sys.exit()

    if rip in makechan_calls:
        rax = gdb.parse_and_eval("$rax")
        rip = getPreviousInstruction(rip)
        print(f"Channel created at address: {hex(rip)}")
        print(f"Channel pointer: {hex(rax)}\n")
        gdb.execute("c", to_string=True)

    elif rip in chansend_calls:
        rax = gdb.parse_and_eval("$rax")
        rbx = gdb.parse_and_eval("$rbx")
        inferior = gdb.selected_inferior()
        value = inferior.read_memory(rbx,8)
        print(f"chansend1 call at {hex(rip)}")
        print("Value  0x"  +  str(value.tobytes()[::-1].hex()),  "sent  through
channel", hex(rax), "\n")
        gdb.execute("c", to_string=True)

    elif rip in chanrecv_calls:
        rax = gdb.parse_and_eval("$rax")
        recv_channels[rip] = rax
        gdb.execute("c", to_string=True)

    elif rip in next_chanrecv_calls:
        index = getChanrecvStackAddr(rip)
        rsp = gdb.parse_and_eval("$rsp")
        inferior = gdb.selected_inferior()
        value = inferior.read_memory(rsp+index,8)
        rip = getPreviousInstruction(rip)
        print(f"chanrecv1 call at {hex(rip)}")
        print("Value 0x" + str(value.tobytes()[::-1].hex()), "received through
channel", hex(recv_channels[rip]), "\n")
        gdb.execute("c", to_string=True)
```

Annex C

Closures detection python script

```
import goFunctions
from elftools.elf.elffile import ELFFile
from capstone import *
import os
import sys
import re

dirname = os.path.dirname(__file__)
bin_path = os.path.join(dirname, sys.argv[1])
print("bin path:", bin_path, "\n")

f = open(bin_path, 'rb')
elffile = ELFFile(f)

class Pattern:
    def __init__(self, name, pattern):
        self.name = name
        self.pattern = pattern

def findCallInstruction(address):
    instructions = disasm.disasm(code_bytes, code_section['sh_addr'])
    call_addresses = []
    for instr in instructions:
        if (instr.mnemonic == 'call' and instr.op_str == address):
            call_addresses.append(instr.address)
    return call_addresses

def getInstrAt(addr):
    for i in instructions:
        if i.address == addr:
            return i

def getPrevInstr(addr):
    for i in range(len(instructions)):
        if instructions[i].address == addr:
            if i == 0:
                return -1
            return instructions[i-1]
    return -1

def getNextInstr(addr):
```

```
    for i in range(len(instructions)):
        if instructions[i].address == addr:
            if i == len(instructions)-1:
                return -1
            return instructions[i+1]
    return -1

def findIndirectCall(addr):
    indirectCalls = []
    indirectCallInstructions = ["call rax", "call rbx", "call rcx"]
    for i in range(len(instructions)):
        if instructions[i].address == addr:
            for j in range(i, len(instructions)):
                mnemonic = instructions[j].mnemonic
                op_str = instructions[j].op_str
                if mnemonic == "ret":
                    return indirectCalls
                if mnemonic+" "+op_str in indirectCallInstructions:
                    indirectCalls.append(instructions[j].address)
    return indirectCalls

def getStackOffset(op_str):
    reg = op_str[op_str.find("[")+1:op_str.find(" +")]
    if reg != "rsp":
        return 0
    offset = op_str[op_str.find('+')+2:op_str.find(']')]
    return int(offset, 16)

def getIP(address):
    for i in instructions:
        if i.address == address:
            return address + i.size

def calculateRIP(addr, op_str):
    reg = op_str[op_str.find("[")+1:op_str.find(" +")]
    if reg != "rip":
        return 0
    rip = getIP(addr)
    offset = int(op_str[op_str.find('+')+2:op_str.find(']')],16)
    return rip + offset


for section in elffile.iter_sections():
    if section.name == '.text':
        code_section = section
        break
```

```
# Read the code bytes from the section
code_bytes = code_section.data()

# Create a disassembler object
disasm = Cs(CS_ARCH_X86, CS_MODE_64)

# Disassemble the code bytes and print the results
instructions = disasm.disasm(code_bytes, code_section['sh_addr'])
instructions = list(instructions)

patterns = [
    Pattern('closure',b'\\x48\\x8d\\x0d.{4}\\x48\\x89\\x4c\\x24.'),
    Pattern('closure',b'\\x48\\x8d\\x05.{4}\\x48\\x89\\x44\\x24.'),
    Pattern('closure',b'\\x48\\x8d\\x1d.{4}\\x48\\x89\\x5c\\x24.')
]

closure_create = []

for p in patterns:
    # Use re.finditer() to search for the pattern in the bytes
    matches = re.finditer(p.pattern, code_bytes)

    # Iterate through the matches and print the results
    for match in matches:
        start, end = match.start(), match.end()
        matched_bytes = code_bytes[start:end]
        start_address = code_section['sh_addr'] + start
        #if start_address < 0x47dfe0:
            #continue
        closure_create.append(start_address)

for addr in closure_create:
    next_instr = getNextInstr(addr)
    if next_instr == -1:
        continue
    closure_stack_offset = getStackOffset(next_instr.op_str)
    ind_calls = findIndirectCall(addr)
    closure_calls = {}
    for ind_call in ind_calls:
        priv = getPrevInstr(ind_call)
        op_str = priv.op_str
        if priv.mnemonic == "lea" and op_str[:3] == "rdx":
            offset = getStackOffset(op_str)
            if (offset == closure_stack_offset):
                closure_address = calculateRIP(addr, getInstrAt(addr).op_str)
```

```
                closure_calls[ind_call] = closure_address
    if len(closure_calls) == 0:
        continue
    print(f"Possible closure set-up detected at address: {hex(addr)}")
    for i in closure_calls:
        print(f"Closure call at {hex(i)}. Closure function at
{hex(closure_calls[i])}")
    print()
```