Network Security: Original SYN - Project Report

Year 2023/2024

Chinellato Marco - 886217
Pilotto Gabriele - 902388
Pistellato Martino - 886493
Rizzo Elisa - 884784
Vego Scocco Thomas - 884984

Università
Ca'Foscari
Venezia

Contents

1 Abstract

This project explores and implements the "Original SYN" methodology as proposed by X. Zhang, J. Knockel, and J. R. Crandall in their 2015 paper, "Original SYN: Finding Machines Hidden Behind Firewalls," presented at the IEEE Conference on Computer Communications (INFOCOM). The methodology involves sending specially crafted SYN packets to target machines and analyzing the responses to uncover devices concealed by firewalls.

The results demonstrate the capability of the Original SYN methodology to detect hidden machines with a high degree of accuracy without resorting to any type of invasive attack.

2 Introduction

2.1 Background and Context

The Original SYN methodology was introduced by Zhang, Knockel and Crandall in their research paper titled "Original SYN: Finding Machines Hidden Behind Firewalls", presented at the IEEE Conference on Computer Communications (INFOCOM) in 2015.

This innovative approach leverages the inherent properties of TCP (Transmission Control Protocol) SYN (Synchronize) packets to conduct detailed network reconnaissance, which involves gathering information about network structures, hosts, services and potential vulnerabilities.

2.2 Objectives and Scope

The primary objective of this project is to implement and evaluate the practical application of the Original SYN methodology on older versions of the Linux kernel such as those used in the paper, and also using modern kernels in order to understand if the solution proposed is still applicable nowadays.

By leveraging the principles outlined in the paper "Original SYN: Finding Machines Hidden Behind Firewalls", this project aims to enhance our understanding of network security through the following specific goals:

- **Understand SYN Backlog Behavior:** Gain comprehensive insights into how the SYN backlog is managed and behaves within Linux systems;
- **Infer SYN Backlog Size:** Develop and employ iterative probing techniques to accurately estimate the size of the SYN backlog on target machines;
- **Implement Scan Methodology:** Utilize designated "zombie" machines to execute controlled scans and verify the presence of target machines based on response behaviors;
- **Check Compatibility:** Verify if this technique still valid on modern Linux Kernels and, if not, understand why it's outdated.

It should be noted, however, that these objectives will be subject to minor modifications during the course of the project, in accordance with the findings of the tests. In this regards, the report also provides an illustration of:

- **Use of a controlled environment:** Instead of testing this methodology on real networks we decided to utilize a controlled test environment;
- **Discovers in eviction mechanism:** Understand eviction behavior through source code analysis and kernel recompilation.

3 Methodology and code

The solution proposed by Xu Zhang et al. employs a technique to detect machines that are inaccessible directly from the outside by utilising "zombie" machines in order to infer information about hidden machines.

3.1 Finding zombies

The first step consists in finding the so-called “zombie", an Internet-accessible machine that meets certain requirements such as

- **TCP/IP access:** The zombie must be accessible via standard TCP/IP protocols enabled to receive SYN packets and capable of responding with SYN-ACK or RST, depending on the situation;
- **No filtering:** The zombie must not be located on a network that filters incoming packets especially the ones with spoofed IP;
- **SYN backlog:** The zombie must have a SYN backlog, which is a queue used to handle “half-open” TCP connections (where a SYN has been sent, but the connection has not yet been completed). This queue is used to monitor the status of the packets. It is important to note that a suitable zombie has a SYN queue which is not completely full. A backlog with other connections does not guarantee the correct results;
- **Operating Sistem:** The operating system must be Linux with kernel 3.2 or earlier versions. Regarding SYN backlog management, these versions of Linux begin to remove old “half-open” connections when the queue fills up more than halfway. This is exploited to infer information about hidden machines. This requirement, however, will be disproved during the course of tests and experiments.

Zombie Machine Preparation

In order to detect a suitable zombie machine we’ve used a python script, that uses the *scapy* and *nmap* libraries.

In the "Original SYN" paper they used a measurement machine running Ubuntu 14.04, released in late April 2014, which we suppose was the latest Ubuntu version released at the time their research was carried out. A zombie machine is selected if it runs a Linux version *prior to* than 3.3.

The measurement machine was supposed to generate random IP address and send SYN packet to ports $[21, 22, 80, 443, 631]$ of each generated IP and then check if the host associated to that IP could be a suitable zombie machine.
In a real case scenario, this would be the code necessary to perform this.

```
1 def generate_random_ip():
```

```
    res = ""
    for _ in range(4):
        res += f'{random.randint(0,255)}.'
    return res[:len(res)-1]

def search_zombies_candidate(limit=1: int):
    ports = [21, 22, 80, 443, 632]
    candidates = set()

    TCP_SYN_ACK = 0x02 | 0x10
    """
    TCP Flag Values:
        SYN Flag: 0x02
        ACK Flag: 0x10
        SYN-ACK Flag: Combination of SYN and ACK, which is 0x02 | 0x10
            = 0x12
    """
    for _ in range(limit):
        ip = generate_random_ip()
        for port in ports:
            SYN = scapy.IP(dst=ip)/scapy.TCP(dport=port, flags='S', seq
                =1000)
            response = scapy.sr1(SYN, timeout=4, verbose=True)
            if response is not None:
                tcp_header = None
                if response.haslayer(scapy.TCP):
                    tcp_header = response.getlayer(scapy.TCP)
                if tcp_header is not None and tcp_header.flags ==
                    TCP_SYN_ACK:
                    possible_candidate.add((ip, port))


    return possible_candidates

def select_candidate(os_name: str, accuracy: int):
    if accuracy != "100":
        return None
    if "Linux" not in os_name:
        return False
    try:
        linux_version = os_name.split(' ')[1]
        linux_version.split('.')
        if int(linux_version[0]) > 3:
            return False

        if int(linux_version[0] == 3) and int(linux_version[1] >= 3):
            return False

        return True

    except:
```

```
50          return False
51
52  def nmap_os_detection(ip):
53      nm = nmap.PortScanner()
54      res = nm.scan(hosts=ip, timeout=60, arguments="-O")
55      try:
56          os_match = res["scan"][ip]["osmatch"][0]
57          os_name = os_match["name"]
58          accuracy = os_match["osclass"][0]["accuracy"]
59          return select_candidate(os_name, accuracy)
60      except:
61          return 0
62
63
64  def get_zombies(candidates):
65      zombies = []
66      ip_found = set()
67      for candidate in candidates:
68          ip = candidate[0]
69          port = candidate[1]
70          if candidate not in ip_found:
71              ip_found.add(candidate)
72              if nmap_os_detection(ip):
73                  zombies += [{"ip": ip, "port": port, "fingerprint": res
                        == 2}]
74
75      return zombies
```

For the candidate zombies for which Nmap couldn't detect a running OS, in the paper a hard coded property of the Linux machine, with version 3.0 or earlier, was used. Those Linux versions have a TCP timeout period set to three time longer than 3.1 version and later, using this property they could figure out old enough machine which are a suitable zombie.
A possible approach to test the TCP timeout of the zombie machine could be sending a SYN packet and inspecting the received packets using tcpdump. Initially the zombie will answer with a SYN-ACK packet and add this request to the backlog as an half-open connection. We will not complete the handshake by sending an ACK, thus leading the zombie to re transmit SYN-ACK packets until TCP timeout is reached and the half-open connection is removed.

When a zombie sends a spoofed SYN packet to a target machine in the same subnet, it also sends an ARP request; if the target machine does not exist the ARP request will never receive an answer, hence the SYN packet will be removed from the backlog according to the ARP request timeout.
This behaviour is rate limited to 1 per seconds in machines running a linux version earlier than 3.3, with this rate limitation the SYN scan fills more than half of the backlog at a rate of more than one packet per second.

Nonetheless this observation will be ignored in our project because our final test network does not take into account the case where the zombie is in the same subnet as the target machine.

3.2 Inferring SYN Backlog Size

Understanding the SYN Backlog

The Transmission Control Protocol (TCP) is fundamental in order to establish reliable internet communication and a key component of this process is the SYN backlog.

The size of the SYN backlog in Linux systems is influenced by three main kernel variables:

- **Backlog argument:** the backlog parameter used in the listen() system call;
- **net.core.somaxconn:** which determines the maximum number of pending connections;
- **net.ipv4.tcp_max_syn_backlog:** which influences the initial size of the backlog.

When a client tries to establish a connection with a server it initiates a three-way handshake, which involves three distinct steps:

- **Client sends SYN:** the client sends a SYN (synchronized) packet to the server;
- **Server responds with SYN-ACK:** the server responds with a SYN-ACK (synchronized-acknowledgment) packet, indicating that it's ready to establish a connection;
- **Client sends ACK:** the client sends back an ACK (acknowledgment) packet to confirm the connection.

During this handshake process the SYN backlog is used as a temporary queue to hold incoming connection requests and half-open connections on the server. We can identify a connection as half open if the machines has received a SYN packet, has answered back with a SYN-ACK but has yet to receive an ACK reply to its SYN-ACK.
When the server receives a SYN packet it places the request in the backlog and sends a SYN-ACK in response. Once the client completes the handshake by sending an ACK, a RST, ICMP error or ARP timeout error, the connection is either established or dropped accordingly and the request is removed from the backlog.
If no answer is received by the machine within a specific time limit the SYN-ACK is re transmitted a fixed amount of times, with exponentially increasing timeouts, until the connection times out and is subsequently aborted.

In particular in the Linux kernel versions 2.3 and later, if the SYN backlog is more than half full when we receive a new SYN request, some of the older entries in the backlog will be evicted in order to reserve half of the backlog for the young requests, which are those requests that have not been re transmitted yet.

In order to infer its size what we do is send a duplicate SYN packet with identical source and destination details but with a different sequence number, specifically the sequence number of the original SYN request decreased by 1 and then verify if all the sent request are still in the backlog or not.

To verify whether a SYN packet remains in the backlog we analyse the server's response, which reveals the status of the original SYN packet:

- **ACK response:** indicates that the original SYN packet is still in the backlog;
- **SYN-ACK response:** indicates that the original SYN packet has been evicted from the backlog.

By analyzing these responses we can infer the backlog size.

Implementation and code

In order to estimate the SYN backlog size we start by assuming an initial backlog size and test this assumption by sendind a set of number of SYN packets without responding to the server's SYN-ACKs.
If the backlog size is accurately guessed we observe the eviction of older SYN packets as the backlog fills, otherwise we double the assumed backlog size and repeat the test.
This process continues until we determine the backlog size or reach the maximum typical size.

send_syn_packets The `send_syn_packets` function is designed to flood a target machine with SYN packets to test its SYN backlog size.

```
def send_syn_packets(target_ip, target_port, num_packets):
    """
    Send SYN packets to a target to potentially fill its SYN backlog.

    :param target_ip: IP address of the target machine
    :param target_port: Port number on the target machine
    :param num_packets: Number of SYN packets to send
    """
    logging.info(f"Sending {num_packets} SYN packets to {target_ip}:{
       target_port}")

    for i in range(num_packets):
        ip = IP(dst=target_ip)
        tcp = TCP(sport=40000+i, dport=target_port, flags='S', seq
           =1000+i)
        packet = ip / tcp
        sr(packet, verbose=0)
    logging.info("SYN packets sent.")
```

The SYN backlog as stated before is a buffer used to store half-open connections and this function aims to fill it to a point where older entries might be evicted, which is key to inferring the backlog size.

It takes the following parameters:

- *target_ip*: the IP address of the machine being targeted;
- *target_port*: the port number on the target machine to which the SYN packets will be sent;
- *num_packets*: the number of SYN packets to send to the target machine, which is a fraction of the estimated backlog size.

For each packet an IP and TCP layer are created: the IP layer specifies the destination IP while the TCP layer sets the source port, the destination port, the SYN flag and a unique sequence number.
The `send` function from *scapy* is used to dispatch each packet to the target.

This function implements the methodology's requirement to flood the SYN backlog by sending a calculated number of SYN packets. By incrementing the sequence numbers and source ports, the function ensures each SYN packet is treated as a unique connection attempt. This aligns with the methodology's approach of using SYN packets to potentially fill the backlog and trigger eviction.

check_eviction The `check_eviction` function is used to determine whether a SYN entry has been evicted from the backlog by sending a duplicate SYN packet with a slightly altered sequence number.

```
def check_eviction(target_ip, target_port, original_seq_num,
    original_sport):
    """
    Check whether a SYN entry has been evicted from the backlog.

    :param target_ip: IP address of the target machine
    :param target_port: Port number on the target machine
    :param seq_num: Sequence number used for the original SYN
    :return: 'ACK' if the original SYN is still in the backlog, 'SYN-
        ACK' if evicted, None if no response
    """
    # Send a duplicate SYN with sequence number decremented by 1
    ip = IP(dst=target_ip)
    tcp = TCP(sport=original_sport, dport=target_port, flags='S', seq=
        original_seq_num-1)
    duplicate_syn = ip / tcp
    ans, _ = sr(duplicate_syn, timeout=1, verbose=0)


    if ans:
        for _, packet in ans:
            tcp_layer = packet.getlayer(TCP)
            if tcp_layer.flags & 0x10:  # ACK flag
                logging.info(f"{original_sport} received ACK")
                return 'ACK'
            elif tcp_layer.flags & 0x12:  # SYN-ACK flag
                logging.info(f"{original_sport} received SYN-ACK")
                return 'SYN-ACK'

    return None
```

It takes the following parameters:

- *target_ip*: the IP address of the machine being targeted;
- *target_port*: the port number on the target machine to which the duplicated SYN packets will be sent;
- *seq_num*: the sequence number used for the original SYN packet.

A duplicated packet with a different sequence number is sent to the target, this packet serves as a test to see whether the original SYN packet is still in the backlog.

The function `SYN_ack_or_ack` is used in order to filter the packets to identify whether the response to the duplicated packet is an ACK or a SYN-ACK. This is crucial in order to understand if the original packet was evicted or not since, as explained before:

- An ACK response indicates that the original SYN packet is still in the backlog;
- A SYN-ACK response indicates that the original SYN packet has been evicted from the backlog and thus the duplicated SYN packet is being processed.

In order to capture the response packet to later on determine whether the original packet has been evicted or not, we used the `sniff` function from the *scapy* library, which uses a filter to capture only relevant TCP packets from the target IP and port.

This function aligns with the methodology's technique for checking eviction status. By sending a duplicate SYN with a slightly modified sequence number, it tests whether the original SYN is still present in the backlog or has been evicted. This method leverages Linux's eviction strategy to infer backlog status.

infer_backlog_size The `infer_backlog_size` function estimates the SYN backlog size of a Linux machine by incrementally testing various backlog sizes and observing the eviction behaviour.

```
def infer_backlog_size(target_ip, target_port):
    """
    Infer the SYN backlog size of a target Linux machine.

    :param target_ip: IP address of the target machine
    :param target_port: Port number on the target machine
    :return: Estimated SYN backlog size
    """
    min_backlog = 16
    max_backlog = 256
    backlog_size = min_backlog

    while backlog_size <= max_backlog:
        logging.info(f"Testing backlog size: {backlog_size}")

        # Send SYN packets to fill the backlog
        send_syn_packets(target_ip, target_port, int(3/4 * backlog_size
            ))

        # Check for eviction
        eviction_detected = False
        for i in range(int(3/4 * backlog_size)):
            response = check_eviction(target_ip, target_port, 1000 + i,
                40000 + i)
            if response == 'SYN-ACK':
                eviction_detected = True
                break

        if eviction_detected:
            logging.info(f"Eviction detected at backlog size: {
                backlog_size}")
            break
        else:
            backlog_size *= 2

    if backlog_size > max_backlog:
        logging.info("Backlog size is greater than 256 or unable to
            determine.")
```

```
35          return None
36      else:
37          logging.info(f"Inferred backlog size: {backlog_size}")
38          return backlog_size
```

It takes the following parameters:

- $target\_ip$: the IP address of the machine being targeted;
- $target\_port$: the port number on the target machine to which the SYN packets will be sent.

The function initializes the backlog size to 16, the smallest possible backlog size according to typical Linux configurations, and the maximum backlog size to 256, as specified in the paper.

It sends $\frac{3}{4}$ of the backlog size being considered in SYN packets to fill the backlog by calling the `send_syn_packets` function with this number of packets passed with the $num\_packets$.

The function iterates through the sent SYN packets and uses `check_eviction` to determine if any of the original SYN packets have been evicted.

If a SYN-ACK response is received it confirms that eviction has occurred, indicating the backlog size is at least as large as the tested size.

On the other hand if no eviction is detected the backlog size is doubled and the test is repeated. This process continues until either eviction is detected or the backlog size exceeds 256.

3.3 Finding target

Once the previous sections are complete, the actual scan can be executed: it takes in input the IP address of the zombie and its port, the zombie backlog size and lastly the IP address of the target machine. The scan relies heavily on the backlog of the zombie machine while trying not to cause Denial of Service (DoS) attacks to it by filling only $\frac{3}{4}$ of the backlog and thus leaving free space for other possible incoming connections.

To implement the scan, $\frac{3}{8}$ of the zombie's backlog will be filled with SYN requests with spoofed source addresses, as if the target machine was the one that sent them, and the other $\frac{3}{8}$ will contain packets used as canaries. Canaries are used to detect if the target machine is reachable by the zombie through the use of probe packets, which are identical to the canaries except for the sequence number which is smaller by 1. Based on the answers to these probes, it is possible to understand whether the target is reachable or not.

To do this, we must first understand the dynamics of the communications; once the zombie has received the spoofed packets, it will send SYN-ACK packets to the target which can result in two different scenarios:

- **Target machine is reachable:** if the target machine is reachable by the zombie, it will respond with RST packets to the SYN-ACKs as it wasn't the one that initiated the connection. This causes the half-open spoofed connections in the backlog to be removed and, since there is enough space to avoid triggering the backlog eviction, the backlog will be filled with canaries, thus sending the probes to the zombie will result in receiving mostly ACKs;
- **Target machine is unreachable:** in this case, the spoofed packets are not removed from the backlog, which will then be at least full by $\frac{3}{4}$, thus triggering eviction. Packets will be eliminated until the backlog is reduced to half its capacity. Consequently some canaries will be evicted and the zombie will respond with SYN-ACKs to the probes.

The scan starts by creating $\frac{3}{8} \cdot backlog$ packets with spoofed IP source addresses and an equal number of canaries (the source port for both types of packets are randomized); then, canaries and spoofed packets are randomly mixed in order to prevent unwanted behaviours (like eviction from the backlog due to exceeded TTL instead of received RST) and the probes are created. Subsequently, the mixed packets are sent in order to fill $\frac{3}{4}$ of the zombie backlog and, after that, the probes are also sent. Lastly the answer to the probes are processed and analyzed.

This process is repeated three times in order to minimize the possible impact of errors, then the results are compared and the best sample is chosen.
The choice is based on three parameters:

- **Packet loss:** defined as $loss := canaries\_number - answers\_received$, it represents the number of lost packets (not considering the probes);
- **Received ACK:** the number of ACKs received from sending the probes;
- **Evicted canaries:** defined as $k := probes\_number - received\_ACKs$, it represents the number of canaries evicted from the zombie's backlog.

Two cases are then possible:

- There is packet loss in all of the trials;
- There is at least one trial in which there is no packet loss.

In the first case, the trial with lowest packet loss is chosen; in the second case, if there is only one trial with no packet loss, that trial is selected, otherwise the trial with the highest number of evicted canaries is chosen, in order to be conservative and avoid making assumptions.

Lastly, in accordance with the original paper, a statistical analysis is conducted to determine with sufficient confidence whether the target machine is reachable or not. According to their study, the number of evicted canaries follows a hyper-geometric probability distribution and the probability of the target machine to be unreachable is given by

$$P(x \leq k) = \sum_{x=1}^{k} \frac{\binom{N}{x}\binom{N-K}{n-x}}{\binom{N}{n}}$$

These analyses were designed for real-world applications where packet loss may occur and the state of the zombie machine is unknown. In our case, packet loss is almost impossible since the communications occur between virtual machines hosted on the same physical server. Therefore, the statistical analysis in our study does not hold the same significance as it did in the original paper, but it was performed nonetheless to remain as faithful as possible to the original work.

Implementation and code

backlog_syn_scan The `backlog_syn_scan` function is designed to use a zombie machine to infer if a target machine is alive by sending a mixture of spoofed packets and canaries.

```
1 def backlog_syn_scan(bcklg_size, zombie_ip, zombie_port, target_ip):
2     """
3     Check whether the target machine is alive and reachable.
4
5     :param bcklg_size: zombie's backlog size
```

```
    :param zombie_ip: zombie's IP address
    :param zombie_port: zombie's port number
    :param target_ip: target's IP address
    :return: p_value, represent the probability of the target machine
       to be unreachable
    """
    trials = []
    packets = []

    # we want to fill only 3/4 of the whole backlog not to DoS the
       zombie
    packets_number = int(bcklg_size*3/4)
    # we send half of the total size of packets as SYN packets
    syn_packets_number = int(packets_number/2)
    # the other half is filled with canaries
    canaries_number = packets_number - syn_packets_number
    # we use random ports, doesn't matter if they're open
    ports = np.random.default_rng().choice([x for x in range(49151,
       65535)], syn_packets_number, replace=False)

    # Creates the spoofed SYN packets
    for port in ports:
        packet = scapy.IP(dst=zombie_ip, src=target_ip)/scapy.TCP(dport
           =zombie_port, sport=port, flags="S")
        packets.append(packet)

    # Creates the canaries
    for port in ports:
        packet = scapy.IP(dst=zombie_ip)/scapy.TCP(dport=zombie_port,
           sport=port, flags="S", seq=1)
        packets.append(packet)

    # shuffle randomly the packets
    packets = random.sample(packets, len(packets))

    # Creates the probes
    probes = []
    for port in ports:
        # probes are equal to canaries with seq-=1
        packet = scapy.IP(dst=zombie_ip)/scapy.TCP(dport=zombie_port,
           sport=port, flags="S", seq=0)
        probes.append(packet)

    # we repeat the process 3 times to get a more accurate result
    for _ in range(3):
        # sending SYN packets mixed with canaries
        answered, _ = scapy.sr(packets, inter=1/5, timeout=0)
        loss = canaries_number - len(answered)

        # "pinging" the canaries by sending the probes
        answered, _ = scapy.sr(probes, timeout=5)
```

```
51
52          # counting the number of ACKs and SYN-ACKs on the answers to
                the probes
53          ack = 0
54          sa = 0
55          for _, response in answered:
56              if response is not None:
57                  tcp_header = response.getlayer(scapy.TCP) if response.
                        haslayer(scapy.TCP) else None
58                  # ACK received
59                  if tcp_header is not None and tcp_header.flags == 0x10:
60                      ack += 1
61                  # SYN-ACK received
62                  elif tcp_header.flags == 0x12:
63                      sa += 1
64
65          logging.info(f"Total canaries: {canaries_number}, ACKs: {ack},
                SYN-ACKs: {sa}")
66
67          trials.append({'loss': loss, 'ack': ack, 'k': canaries_number -
                ack})
68          # wait for the backlog to be processed or cleaned
69          sleep(packets_number/5)
70
71      # we use statistical analysis to determine the p-value
72      return find_p_value(trials, canaries_number, bcklg_size)
```

It takes the following parameters:

- $bcklg\_size$: zombie's backlog size, obtained from the previous functions;
- $zombie\_ip$: zombie's IP address;
- $zombie\_port$: zombie's port number;
- $target\_ip$: target machine's IP address;

The function implements the scan by sending the appropriate number of packets and by analyzing the responses. It starts by creating $\frac{3}{8} \cdot backlog$ spoofed packets using the target machine's source IP, along with an equal number of canaries. These packets are then randomly mixed and probes, that are identical to the canaries except for a lower sequence number, are created. The mix of canaries and spoofed packets are then sent with an rate of 5 packets per second.

Once the packets are sent, the function sends the probes to assess the number of canaries that are still present in the zombie's backlog and calculates the parameters used to evaluate the p_{value}. This process is repeated three times and all the results are collected in the $trials$ list.

Finally, the `find_p_value` is called to compute the probability that the target machine is unreachable.

find_p_value The `find_p_value` function is designed to compute the probability that the target machine is unreachable based on the results of the previous communications.

```
1  def find_p_value(trials, canaries_number, bcklg_size):
2      """
```

```
    Find the p-value of the target machine to be alive based on the
       results of the trials.

    :param trials: list of dictionaries containing the results of the
       trials
    :param canaries_number: number of canaries sent
    :param bcklg_size: backlog size of the zombie
    :return: p_value, represent the probability of the target machine
       to be unreachable
    """
    loss1, loss2, loss3 = trials[0]['loss'], trials[1]['loss'], trials
       [2]['loss']
    ack1, ack2, ack3 = trials[0]['ack'], trials[1]['ack'], trials[2]['
       ack']
    k1, k2, k3 = trials[0]['k'], trials[1]['k'], trials[2]['k']

    # We choose the trial with the minimum loss, if there are no losses
        we choose the one with the maximum number of evictions k
    if loss1 != 0 and loss2 != 0 and loss3 != 0:
        if loss1 == min(loss1, loss2, loss3):
            loss, ack, k = loss1, ack1, k1
        elif loss2 == min(loss1, loss2, loss3):
            loss, ack, k = loss2, ack2, k2
        else:
            loss, ack, k = loss3, ack3, k3
    else:
        zero_losses = [loss1, loss2, loss3].count(0)
        if zero_losses == 1:
            if loss1 == 0:
                loss, ack, k = loss1, ack1, k1
            elif loss2 == 0:
                loss, ack, k = loss2, ack2, k2
            else:
                loss, ack, k = loss3, ack3, k3
        else:
            if k1 == max(k1, k2, k3) and loss1 == 0:
                loss, ack, k = loss1, ack1, k1
            elif k2 == max(k1, k2, k3) and loss2 == 0:
                loss, ack, k = loss2, ack2, k2
            else:
                loss, ack, k = loss3, ack3, k3

    # number of successes in the population
    K = canaries_number - loss
    # population size, assuming that canaries_loss == spoofed_loss
    N = 2*K
    # number of draws taken without replacement
    n = N - (bcklg_size/2)
    p_value = hypergeom.cdf(k, N, K, n)

    logging.info(f"Loss: {loss}, ACKs: {ack}, Evicted Canaries: {k}, N:
```

```
        {N}, K: {K}, n: {n}, p-value: {p_value}")
    return p_value
```

It takes the following parameters:

- $trials$: list of dictionaries containing the results of questioning the zombie with the probes;
- $canaries\_number$: number of sent canaries;
- $bcklg\_size$: zombie's backlog size;

Based on the results of the previous communications, the best trial is chosen according to the number of lost packets and evicted canaries: if in all trials some packets were lost, then the one with lowest loss is chosen; otherwise the best trial is the one with 0 packet loss and the highest number of evicted canaries.

Based on the chosen trial, the parameters for the hyper-geometric probability distribution are set and the p_{value} is calculated and returned, concluding the scan.

main Finally, all components discussed are integrated into the `main` function, which starts by inferring the backlog size and then executes the scan.

```
def main(zombie_ip, zombie_port, target_ip):
    backlog_size = backlog.main(zombie_ip, zombie_port)
    p_value = SYN_scan.backlog_syn_scan(backlog_size, zombie_ip,
        zombie_port, target_ip)
    print(f"Target ip: {target_ip}, Backlog size: {backlog_size}, P-
        value: {p_value}")

if __name__ == "__main__":
    zombie_ip = sys.argv[1]
    zombie_port = int(sys.argv[2])
    target_ip = sys.argv[3]

    main(zombie_ip, zombie_port, target_ip)
```

It takes the following parameters from command line:

- $zombie\_ip$: zombie's IP address;
- $zombie\_port$: zombie's port number;
- $target\_ip$: target machine's IP address;

4 Testing playground

The following chapter addresses the part of network creation. In particular, we will present each network configuration we have worked on, the motivations behind the choice of each network and the problems encountered during their development.

4.1 Objectives

As the project evolved, the project goals changed and with them the scenario was adapted in order to remain in conformity with the overall objectives. The changes concerned the tools used and the network topologies and configurations. Some changes have also been made in the experimental part, but to a lesser extent.

The idea of developing a dedicated test area arose from the desire not to use the public network: although it is a type of scan that does not cause DoS attacks, our tests involve a partial saturation of resources, which we consider unfair to potential victims.

However, to ensure a plausible scenario and guarantee truthfulness, we have created dedicated test areas that reflect realistic scenarios with multiple hosts, routers and networks.

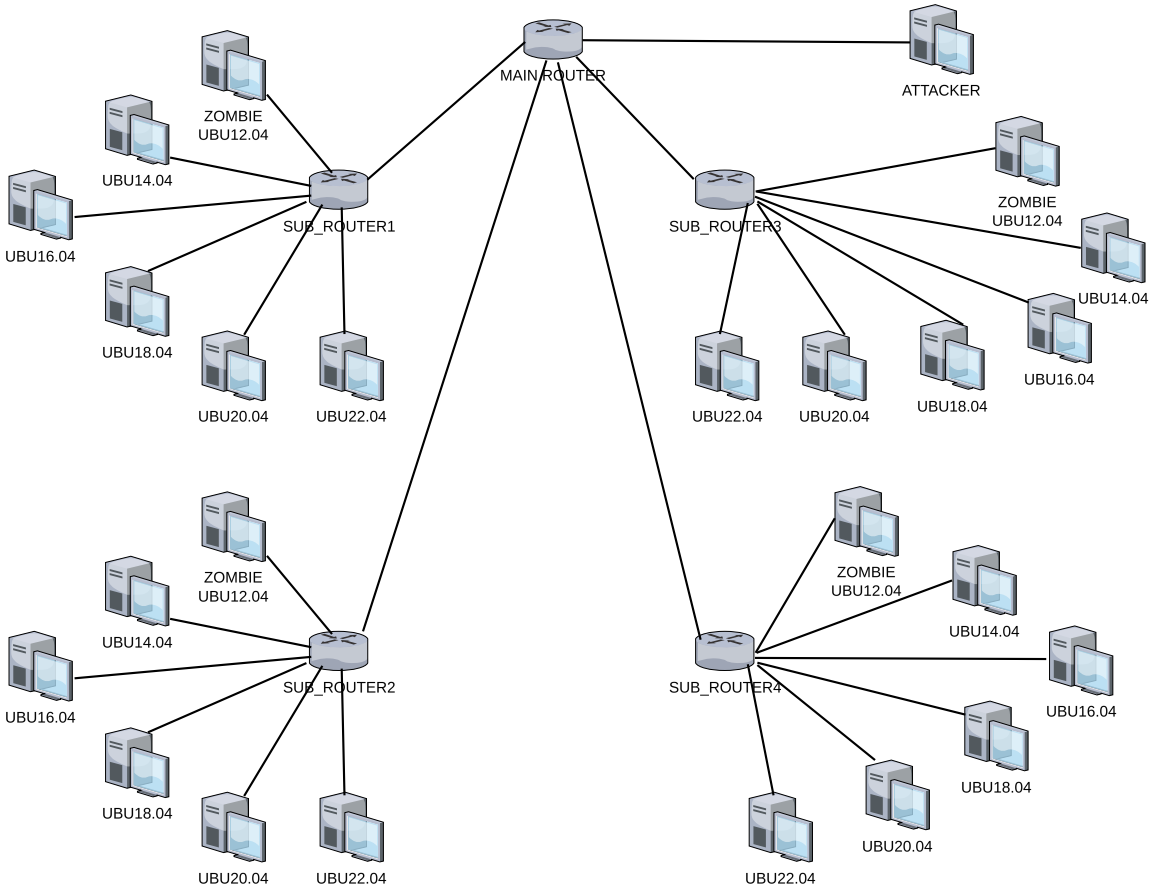
4.2 Containers

The first implementation relies on Docker, a solution that automates the deployment of applications inside portable containers. Those containers are lightweight and allow developers to choose a custom image, manage aspects like networking or packages and use them as testing environments. The benefits of containerization are its simplicity, flexibility and portability, fundamental aspects in our project.

Another big advantage is the availability of ready-to-use OS images on Docker Hub. Our choice has relied on Ubuntu, a Linux distribution based on the unstable branch of Debian. Ubuntu is one of the most popular Linux distribution in enterprise and personal scenarios; furthermore, all previous versions are already available on Docker Hub.

4.2.1 First network topology

The first topology implemented consists of 4 distinguished networks. Every sub-networks has its own router that separates the main-network from the private one where all the host are located.



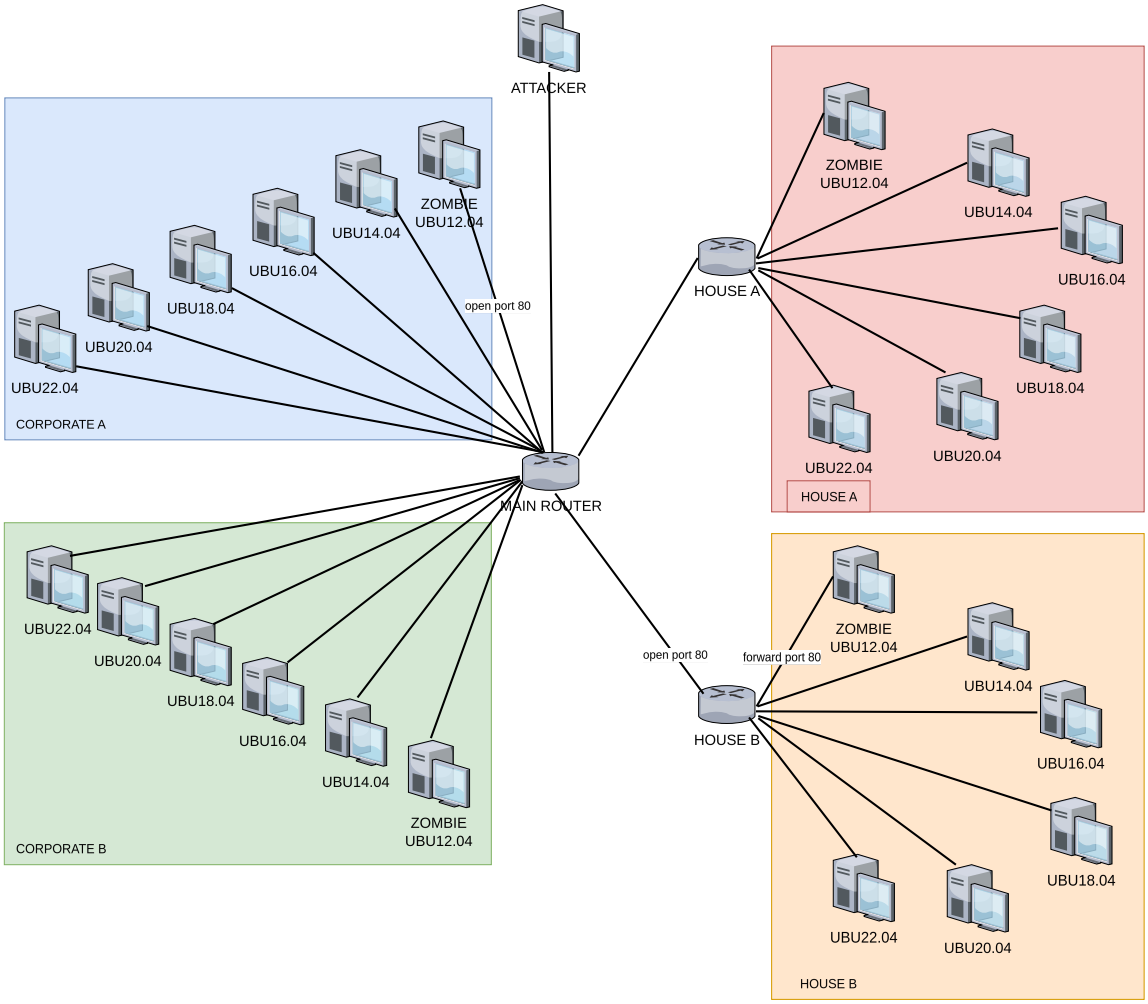
The main-network is managed by the "main-router", which functions as a gateway between the host machine network and the test environment. Each "sub-router" is connected to the main router, and the routing between the different sub-networks is handled using the OSPF (Open Shortest Path First) routing protocol. All routers run FRRouting with standard configurations and no filtering.

Each sub-network contains 6 hosts, each running a different version of Ubuntu. The oldest version included is Ubuntu 12.04, chosen because its original Kernel version was 3.2, meeting the requirements for acting as a zombie. The other 5 containers are based on all the LTS (Long Term Support) Ubuntu versions: 14.04, 16.04. 18.04, 20.04, 22.04. With this plethora of containers acting as targets, we aimed to determine whether the newer versions could still be scanned or if any defensive techniques had been implemented during the years.

Unfortunately, when we launched the scan from an attacker connected to the "main-network", we discovered that it was impossible to reach all the hosts since they where located inside a private network and the routers were not configured for NAT (Network Address Translation) activities. As a result, we decided to redesign the network topology.

4.2.2 Second network topology

The second attempt revisits the basic concept of the first topology, discarding the idea of 4 identical sub-networks.



We imagined a large /24 main-network managed by the "main-router", resembling a small local ISP network. Each customer has its own IP range, which they can use to connect various hosts.

Corporate A

Corporate A simulates a corporation with several hosts (all running Ubuntu LTS machines from 12.04 to 22.04) with an IP belonging to the main-network. One of them, the zombie, provides a web service on port 80 that will be used as communication port by the attacker to assess the Zombie's backlog size and to launch the scan on the Corporation A subnet.

Corporate B

Corporate B has the same configuration as Corporation A, except that no ports are open. We decided to include this scenario because we believe that the Zombie in Corporation A will be able to discover devices in Corporation B as well, since both of them share the same main-network.

House A

House A has a router that creates a private network where the hosts are connected. The router does not provide NAT-ting or port forwarding and has no routes configured to allow external devices to connect to the internal hosts.

House B

House B has the same configuration as House A, except that the Zombie offers a web service on port 80. The router has a port-forwarding rule binding `external_IP:80` to `zombie_ip_80`. All other hosts in House B should not be visible from external devices.

This implementation should enable the attacker to find Zombies in Corporate A and House B. However, during the initial experiments, we discovered a limitation with Docker's virtual switches and an issue with the containers' kernel. Both of these issues are explained in the following section.

4.2.3 Docker limitations

Unfortunately, Docker wasn't our definitive solution. After few tests (and gaining a better understanding of Docker's phylosophy) we decided to transition our testing environment from Docker to Proxmox, another virtualization solution. However, we believe that explaining our motivations will provide clarity on our decision.

Docker networking

Network management in Docker is focused on isolation between containers and subnets. The goal of containerization is to create safe environments to deploy applications and, for this reason, has by default a high isolation degree that requires specific relaxations to allow communication between different components. An aspect that can't be relaxed is the creation of a "ghost gateway" for every network declared in Docker, specifically what happens is that Docker automatically creates an enter/exit point, the gateway, for each of the subnets created which makes it so that the machines inside a specific subnet cannot directly communicate with machines external to said subnet.

This is great since it increases security, but it requires a walk-around solution: the "main-router" previously presented, instead of having a single network interface and a virtual switch where hosts can connect, requires a dedicated NIC (Network Interface Card) for every subnet attached. Furthermore, having those default gateways, made it necessary to create a small bash script to configure the right route everytime the container was launched. Changing the default gateway set by Docker, the container loses the ability to connect to Internet, making operations such as dependency installations difficult. This behaviour persisted also after setting the forwarding rules in the routers and specifying the right route paths. Note that our implementation was based on the "bridge" network mode; moving to one of the other 6 network models may solve those problems, but requires compatible hardware.

Native Kernel

While the networking issues can be partially fixed with some compromises or alternative solutions, the problems related to the Kernels can't be solved.
As previously stated, Docker allows developers to chose a base OS image and run it. However containers do not have their own kernel; they share the host's kernel (for example if the host has Linux 6.8.0, all containers will have as kernel Linux 6.8.0). Isolation is achieved using technologies such as namespaces and cgroups, making containers more lightweight than virtual machines, which require kernel emulation.

Recalling the prerequisites for the zombie machines, we have that the Linux Kernel version must be equal or less than 3.2, a version that has been deprecated and, consequentially, is hard to use nowadays; so, instead of downgrading our host machine, we decided to conduct our tests on virtual machines.

4.3 Virtual Machines

The main difference between containers and virtual machines relies on isolation and resource management. Virtual Machines execute a whole operating system, including its own kernel, over an hypervisor. Obviously, emulating the kernel requires more resources, and since our hardware can not support the same amount of hosts that Docker can run, we need to review our topology.

In order not to waste resources we decided to use Proxmox, an open-source virtualization platform based on a customized Debian distribution.
Since Proxmox has to be installed as an operating system, the performance of the virtual machines should not be downgraded like with other competitors, such as VirtualBox. Virtualization in Proxmox is based on Qemu (Quick Emulator) and KVM (Kernel-based Virtual Machine).

4.3.1 New objectives

The changes to the network structure were primarily driven by the desire to focus on the code and ensure its functionality with recent kernels.
Our main goal was to check if the code worked properly, rather than simulate a realistic environment. Because of this, we removed several elements such as: firewalls, SYN cookies protection and the identification of a suitable zombie (we will assume that the Zombie is always at IP address 192.168.4.100/24). Moreover we maintained the separation of networks to avoid potential ARP-related issues and for convenience and ease of testing.

We can draw our new workflow as follows:

1. **Finding changes between Kernel versions:** understand with which version of the kernel the zombie's behavior stops working as expected;
2. **Understanding the real backlog size:** given a working zombie, understand its backlog size with our implemented code;
3. **Find the target:** this objective remained unchanged;
4. **Understand the eviction behavior:** during our initial tests we discovered that the eviction does not work as expected, so we decided to investigate it further.

4.3.2 New network topology

Our host machine has limited hardware: we can assign to Proxmox only 16GB of RAM and 320GB of disk space. This means that we can support only a few virtual machines, about 8.

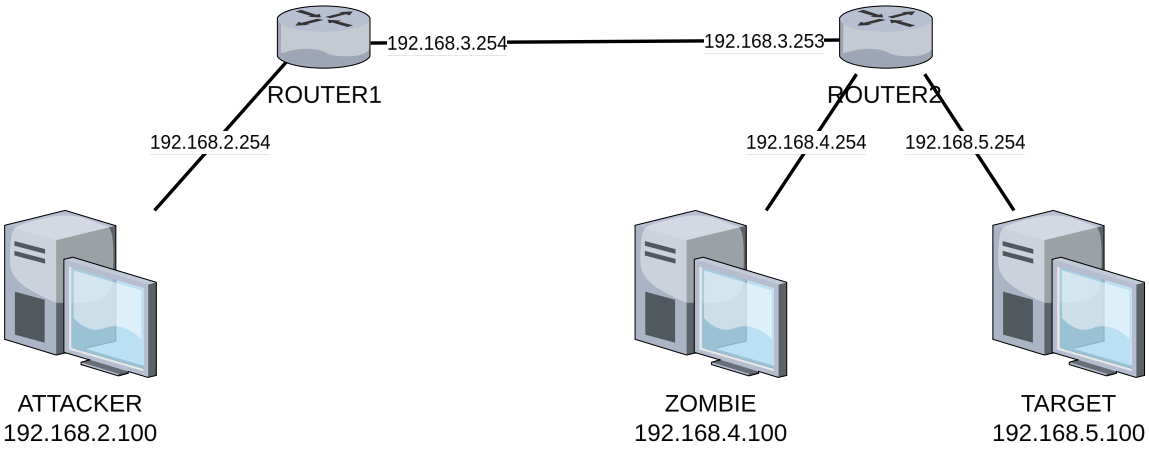
According to the new objectives and the hardware limitations, we decided to review our topology trying to maintain the same behavior of the ones proposed with containers.

Topology

The new topology, that will be the baseline for all the new experiments, is simpler but it keeps a great degree of flexibility in order to perform every type of test.
It is composed by:

- **A subnet for every host:** to avoid ARP timeout issues we decided to create dedicated subnets for every host. All the devices can communicate each other thanks to routing rules in the routers;

- **2 routers:** connected to a dedicated subnet, the two routers make the communication between zombie, target and attacker possible. Both of them have forwarding rules and routing tables configured to allow all types of traffic without filtering;
- **3 hosts:** a zombie, a target and an attacker.



Routers

The two routers are based on Ubuntu 22.04 and are equipped with multiple NICs.

- **Router 1** is configured as follows:

NIC	Network	Assigned IP	Description
ens19	192.168.2.0/24	192.168.2.254/24	Attacker's subnet
ens20	192.168.3.0/24	192.168.3.254/24	Routers' subnet

with the following routing tables:

```
192.168.2.0/24 via 192.168.2.254 dev ens19
192.168.3.0/24 via 192.168.3.254 dev ens20
192.168.4.0/24 via 192.168.3.254 dev ens20
192.168.5.0/24 via 192.168.3.254 dev ens20
```

- **Router 2** is configured as follows:

NIC	Network	Assigned IP	Description
ens18	192.168.3.0/24	192.168.3.253/24	Routers' subnet
ens29	192.168.4.0/24	192.168.4.254/24	Zombie's subnet
ens29	192.168.5.0/24	192.168.5.254/24	Target's subnet

with the following routing tables:

```
default via 192.168.3.254 dev ens19
192.168.3.0/24 via 192.168.3.253 dev ens19
192.168.4.0/24 via 192.168.4.254 dev ens20
192.168.5.0/24 via 192.168.5.254 dev ens20
```

Target

The target machine is a simple Ubuntu Desktop 22.04.4. Since it is the "victim" of this attack it doesn't require any specific configuration.
However, it has to satisfy some requisites:

- **Firewalls:** the target can be protected by a network firewall but not by an application firewall. As a rule of thumb, the firewall rules must allow RST responses to unsolicited SYN-ACK;
- **Internal protection:** IDS (Intrusion Detection Systems) and defensive techniques as SYN_Cookies may interfere with the expected behavior of the scan;
- **Reachable IP:** having a public IP is recommended but not mandatory; it is necessary that the zombie can reach the target in order to send SYN-ACK packets.

Our target machine has IP address 192.168.5.100/24, the `SYN_cookies` are disabled and it has no IDS.

Attacker

The attacker is the host that will launch the scan, it does not have any specific requirements, except for:

- **Spoofing capabilities:** the attacker must be connected to a network that allows sending spoofed IP packets. A network without any kind of filtering is preferred;
- **Basic knowledge of target network:** knowing the type of filtering implemented on the target's network in advance can speed up the process and reduce false negatives;
- **Python, Scapy and Nman:** our implementation requires Python3, Scapy 2.5.0 and Nmap 7.95 to run properly. We suggest using a Linux machine as the attacker since Nmap for Windows and MacOS has not been tested.

Our attacker has Ubuntu Desktop 22.04.4 and its IP address is 192.168.2.100/24.

Zombie

The zombie mutates during our experiments. Since one of our new goals is to find out if the scan also works with newer kernels, we tried multiple Ubuntu versions, starting from 11.04 up to 24.04.1. Almost every time, `SYN_cookies` were disabled, but some experiments included them (in those cases it will be explicitly mentioned). Notice that some prerequisites (such as lower Kernel version) are no longer required, but it is still important that the TCP behavior and SYN backlog management respects the rules presented in Chapter 3. The zombie's IP address is 192.168.4.100/24.

4.3.3 In-depth: common Ubuntu issues

Packages availability

As previously stated, we perform our tests on machines running older Ubuntu versions, including deprecated and no longer maintained releases. This led to significant issues in finding packages and dependencies, as the mirrors for these versions are no longer available.
Furthermore, most of the older versions have incompatible TLS suites, making it impossible to establish connections with alternative mirrors or GitHub.
This was a key reason for choosing Ubuntu versions with preinstalled desktop environments: generally, those distribution include the necessary softwares for our analysis.

When possible, we used the official Debian archive to retrieve older `.deb` files.
However, many developers have removed older packages compatible with deprecated Ubuntu versions.

Login loop

The login loop is a well-documented issue that affects some Ubuntu version from 14.04 LTS to Ubuntu 15.10: when an user attempts to log-in with the correct credentials, after a few seconds the login form will be displayed again.
This problem makes it impossible to log-in into the user-space (mostly) due to driver graphic issues that can't be fixed when using Proxmos. During our tests we tried to disable the desktop environment and try to log-in and use only the recovery terminal, but some actions still required the user to log-in into its account.
Due to this issue, we weren't able to determine if the changes in Kernel first occurred between versions 3.13.0-32 and 4.4.0-21.

5 Experiments

5.1 Finding kernel's changes in zombies

In this section, we explore a series of tests that revealed unexpected behavior in Linux kernel versions starting from 4.4.0 onwards. Our testing uncovered a critical issue where these kernel versions handled the backlog parameter in an unanticipated manner, filling the backlog with only a single request. This anomaly prompted a detailed investigation into the kernel's source code.

To understand the root of this issue, we focused on comparing the kernel code of the last version that does not present the unexpected behavior to the first version exhibiting it. This comparative analysis allowed us to identify significant modifications in the functions responsible for allocating and defining the backlog size.

5.1.1 Opening ports with Netcat

Initially, we decided to utilize Netcat, a versatile networking utility that allows users to set up a listening service on specified ports.
Its simplicity and broad functionality made it an ideal choice for our needs.

To listen on an open port with Netcat, we used the following command, `nc -l -k -p <port>`, where `<port>` represents the port number we wish to open.
This command instructs Netcat to listen (`-l`) for incoming connections on the specified port (`-p`) and forces itself to stay listening for another connection after its current connection is completed (`-k`).

5.1.2 Test methodology

In our effort to assess the correct functioning of the code designed to infer the backlog size, we conducted a series of tests involving the attacker and zombie machines. The objective was to validate whether the code correctly determined the backlog size by observing the interaction between these machines.

The attacker was configured to execute the code responsible for inferring the backlog size, while the zombie machine was prepared to handle incoming connection requests and respond accordingly. We included logging in the code in order to capture the type of response packets received, specifically observing whether ACK or SYN-ACK flags were present.

In the first batch of tests we followed the approach specified in the paper.
We sent $\frac{3}{4}$ of the backlog size we were testing in SYN packets to the zombie, each originating from a different attacker port, and then we sent the same number of duplicate SYN packets from the same ports, but with the sequence numbers decreased by one.

For the second batch of tests, aimed at analyzing the inconsistency observed in the first set of tests, we modified the original code. In this modified version the attacker completed the handshake process by saving and analyzing the response from the zombie in order to build the corresponding ACK packet. This allowed us to retrieve the necessary fields, such as the `ack` field and the correct sequence number, in order to build an appropriate final ACK packet and successfully complete the connection between the machines.

5.1.3 Small code adjustments

As stated before, in the second batch of tests, we modified the code in order to complete the handshake, in particular we modified the function `send_syn_packets` as follows:

```
def send_syn_packets(target_ip, target_port, num_packets):
    logging.info(f"Sending {num_packets} SYN packets to {target_ip}:{
       target_port}")

    for i in range(num_packets):
        ip = IP(dst=target_ip)
        tcp = TCP(sport=40000+i, dport=target_port, flags='S', seq
           =1000+i)
        packet = ip / tcp
        received = sr1(packet, timeout=0, verbose=0)
        logging.info(f"Sent TCP packet from port {40000+i}")

        if received and received.haslayer(TCP) and received[TCP].flags
           == 'SA':
            ack = TCP(sport=received[TCP].dport, dport=target_port,
               flags='A', ack=received[TCP].ack+1, seq=1000+i+1)
            send(ip/ack, verbose=0)
            logging.info(f"Sent ACK packet to complete the handshake")

        time.sleep(1/5)
    logging.info("SYN packets sent.")
```

For each packet, an IP and TCP layer are created, with the IP layer specifying the destination IP and the TCP layer setting the source port, destination port, SYN flag and a unique sequence number. The `sr1` function from `scapy` is used to send the SYN packet and receive the first response from the target, typically a SYN-ACK packet if the connection is accepted and still present in the backlog.

This function differs from the original methodology by completing the three-way handshake when a SYN-ACK is received. After capturing the SYN-ACK, the code constructs an ACK packet by incrementing the sequence and acknowledgment numbers as per the TCP protocol. The `send` function then dispatches the ACK packet to the target, completing the connection.

Furthermore we modified the `infer_backlog_size` in order to call only the `send_syn_packets` function inside of the while loop.

5.1.4 Test results - when the kernel changes

Our testing on kernel versions 3.13.0 and 4.4.0 revealed significant differences in the behavior of SYN flood detection, specifically:

- **Kernel Version 3.13.1:** In the older kernel version SYN flood detection occurred when the 16th packet was received. This behavior was consistent with the backlog size we inferred from our tests, suggesting that the system could handle a backlog of at least 16 before initiating SYN flood protection measures.

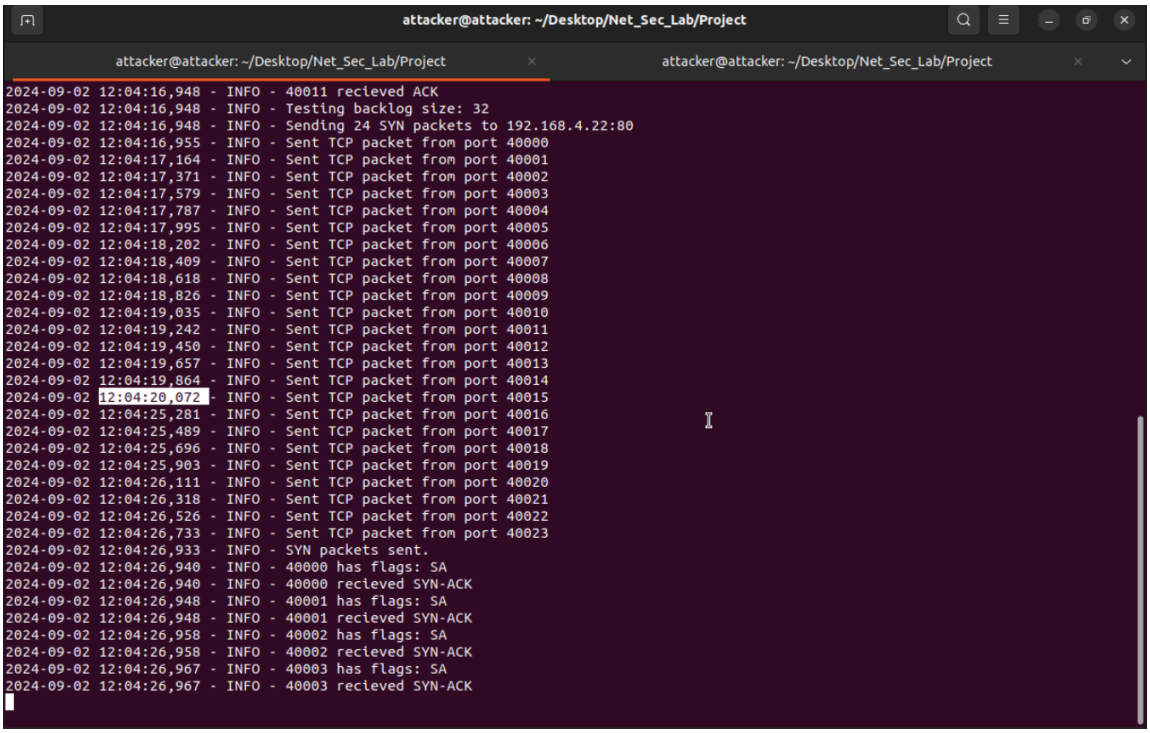


Figure 5.1: SYN flooding test results on kernel version 3.13.1 from the attacker's point of view

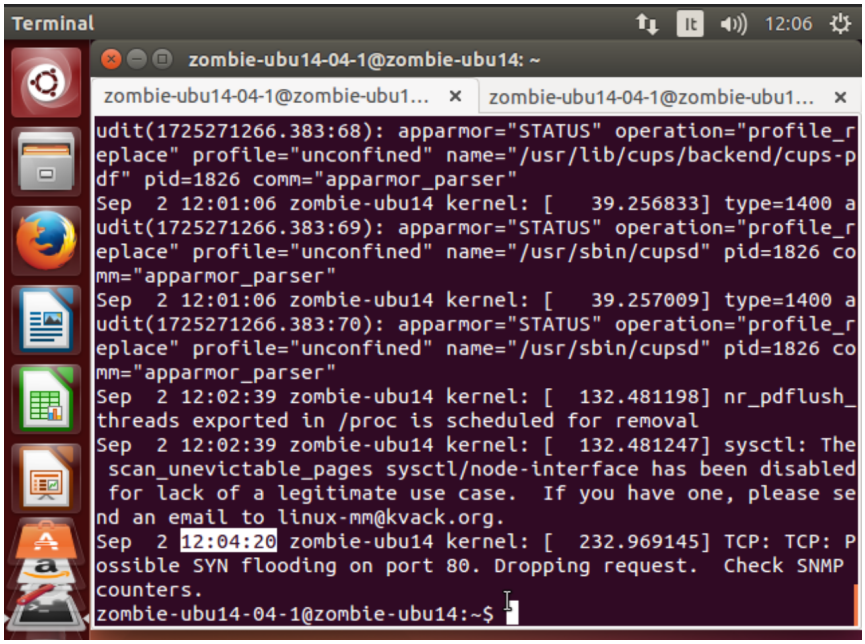


Figure 5.2: SYN flooding test results on kernel version 3.13.1 from the zombie's point of view

It can be observed from the highlighted timestamps in the screenshots above, that as soon as the 16th packet is sent at 12.04.20 by the attacker, the zombie machine detects a possible SYN flood and thus starts dropping all the subsequent packets;

- **Kernel Version 4.4.0:** In the newer kernel version SYN flood detection was triggered immediately upon receiving the first packet. This early detection was unexpected and suggested that the backlog handling was significantly different from that in the older version.

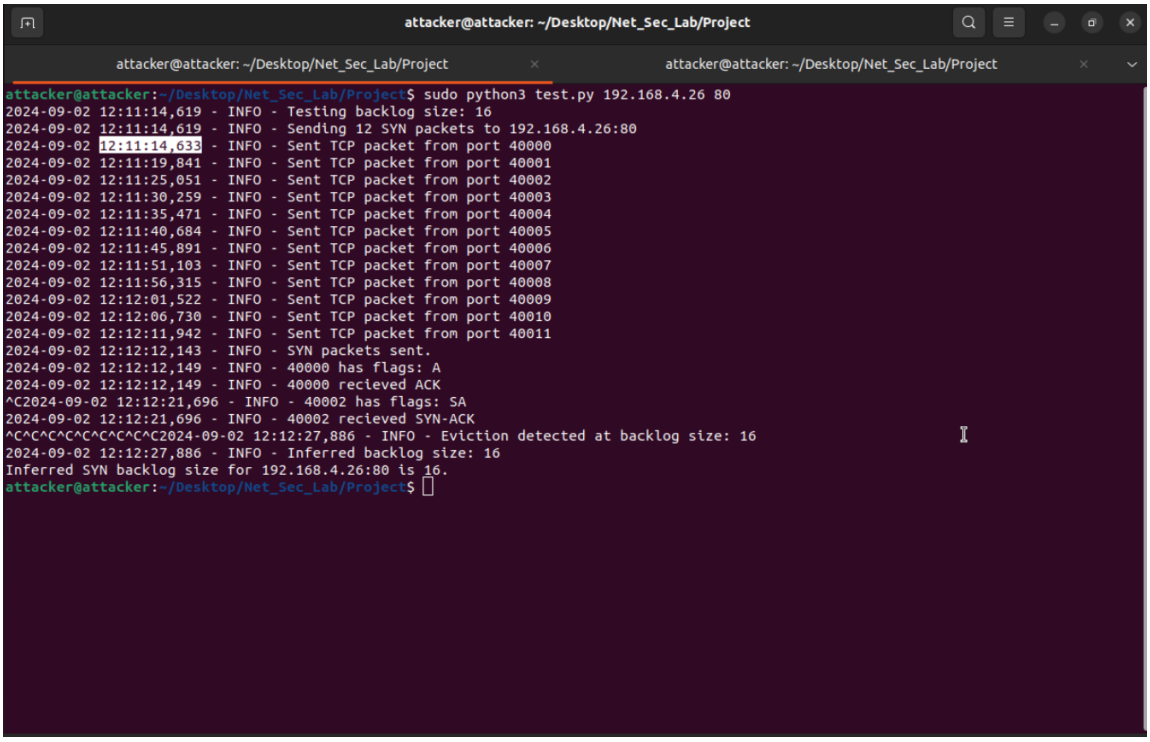


Figure 5.3: SYN flooding test results on kernel version 4.4.0 from the attacker's point of view

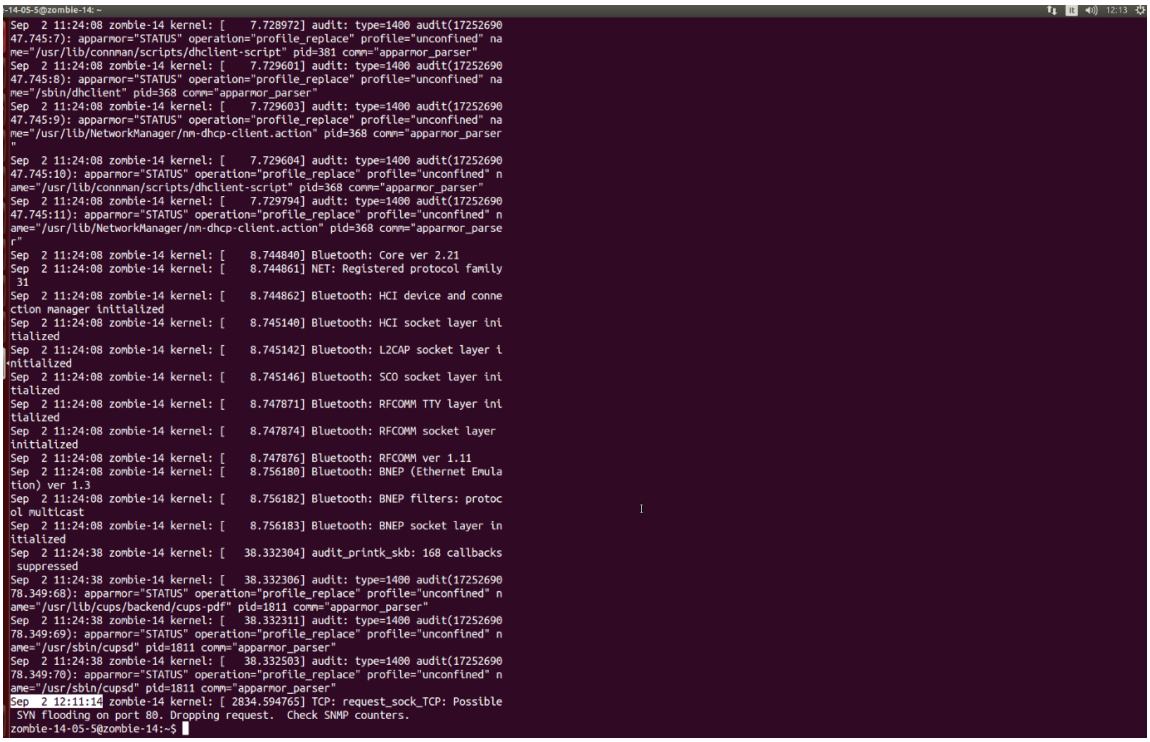


Figure 5.4: SYN flooding test results on kernel version 4.4.0 from the zombie's point of view

It can be observed from the highlighted timestamps in the screenshots above, that as soon as the first packet is sent at 12.11.14 by the attacker, the zombie machine detects a possible SYN flood and thus starts dropping all the subsequent packets;

To further investigate this discrepancy, we modified the testing code to include the complete TCP handshake, where the attacker would send a final ACK packet following the initial SYN and SYN-ACK exchanges, but despite completing the handshake, the newer kernel versions still detected a SYN flood immediately after the first handshake packet.
This behavior reinforced our hypothesis that the backlog size was set to 1, as the system failed to accommodate more than a single connection request before triggering SYN flood protection.

5.1.5 Why Netcat is (part of) the problem

Although Netcat has always passed a backlog value of 1 to the listen system call across all kernel versions, starting with version 4.4.0-21, this behavior began to cause significant issues with the handling of incoming connections.

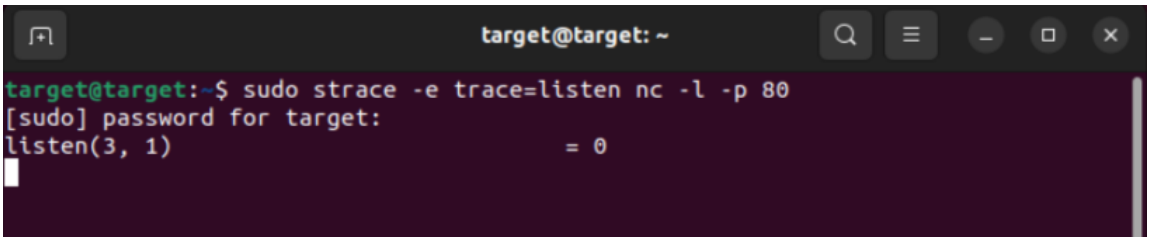


Figure 5.5: Listen function call performed opening ports with Netcat

The listen function is used to prepare a socket to accept incoming connections, and it takes two parameters: the file descriptor of the socket and the backlog size, which specifies the maximum number of pending connections that can be queued before the socket begins rejecting new connections.
This problem did not occur in older kernel versions due to the different handling of the backlog parameter.

To understand the underlying changes, we conducted a detailed analysis of the kernel code to identify the function responsible for managing backlog allocation and size specification.
Starting the definition of listen()'s syscall in *net/socket.c*, our back-tracking led us to the `reqsk_queue_alloc` function in *net/core/request_sock.c*, which plays a critical role in allocating memory for the request socket queue based on the backlog size. We determined that the `nr_table_entries` parameter in this function corresponds to the backlog value passed to the listen function.

In the older kernel versions, the `reqsk_queue_alloc` function handled `nr_table_entries` with the following approach:

```
1 int sysctl_max_syn_backlog = 256;
2 EXPORT_SYMBOL(sysctl_max_syn_backlog);
3
4 int reqsk_queue_alloc(struct request_sock_queue *queue,
5                       unsigned int nr_table_entries)
6 {
7     size_t lopt_size = sizeof(struct listen_sock);
8     struct listen_sock *lopt;
9
```

```
    nr_table_entries = min_t(u32, nr_table_entries,
        sysctl_max_syn_backlog);
    nr_table_entries = max_t(u32, nr_table_entries, 8);
    nr_table_entries = roundup_pow_of_two(nr_table_entries + 1);
    lopt_size += nr_table_entries * sizeof(struct request_sock *);
    if (lopt_size > PAGE_SIZE)
        lopt = vzalloc(lopt_size);
    else
        lopt = kzalloc(lopt_size, GFP_KERNEL);
    if (lopt == NULL)
        return -ENOMEM;

    for (lopt->max_qlen_log = 3;
         (1 << lopt->max_qlen_log) < nr_table_entries;
         lopt->max_qlen_log++);

    get_random_bytes(&lopt->hash_rnd, sizeof(lopt->hash_rnd));
    rwlock_init(&queue->syn_wait_lock);
    queue->rskq_accept_head = NULL;
    lopt->nr_table_entries = nr_table_entries;

    write_lock_bh(&queue->syn_wait_lock);
    queue->listen_opt = lopt;
    write_unlock_bh(&queue->syn_wait_lock);

    return 0;
}
```

In this code:

- `nr_table_entries` is first constrained by `sysctl_max_syn_backlog`, which is 256, so any value greater than 256 would be limited to 256;
- It is then ensured to be at least 8 using `max_t(u32, nr_table_entries, 8)`;
- Finally, `nr_table_entries` is incremented by one and then rounded up to the nearest power of two with `roundup_pow_of_two(nr_table_entries + 1)`.

Therefore, if `nr_table_entries`, or backlog, was initially 1, the value would be adjusted as follows:

- It remains 1 after the first constraint;
- It is adjusted to 8 by `max_t`;
- It is increased to 9 and then rounded up to the next power of two, which would result in 16.

This means that a backlog value of 1 would effectively be treated as 16 in the older kernels due to these adjustments.

In contrast, newer kernel versions simplified the `reqsk_queue_alloc` function to:

```
void reqsk_queue_alloc(struct request_sock_queue *queue)
{
    queue->fastopenq.rskq_rst_head = NULL;
    queue->fastopenq.rskq_rst_tail = NULL;
    queue->fastopenq.qlen = 0;
```

```
    queue->rskq_accept_head = NULL;
}
```

In this updated code, no further adjustments were made to the backlog value, and it was used directly as provided. Consequently, Netcat's backlog value of 1 was applied without modification, leading to inadequate backlog handling and the issues observed in these newer kernel versions.

5.2 Writing our "netcat"

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/wait.h>

#define PORT 8080
#define BACKLOG 16

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);

    // Creating socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Forcefully attaching socket to the port 8080
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
        &opt, sizeof(opt))) {
        perror("setsockopt failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Binding the socket to the port 8080
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) <
         0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
```

```
41      }
42
43      // Start listening on the socket with a backlog of 16
44      if (listen(server_fd, BACKLOG) < 0) {
45          perror("listen failed");
46          close(server_fd);
47          exit(EXIT_FAILURE);
48      }
49
50      // Accept a new connection
51      if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (
            socklen_t*)&addrlen)) < 0) {
52          perror("accept failed");
53          exit(EXIT_FAILURE);
54      }
55
56      close(new_socket);
57      close(server_fd);
58
59      return 0;
60  }
```

This code creates a simple socket on port 8080, listening for an incoming connection.
Before binding the socket to the IP address and port, this program sets socket options to allow the re-utilization of the address and port, as it prevents the "address already in use" error.
Once bound, the socket starts listening for incoming connections with a backlog size defined with the macro **BACKLOG**.

It is important to note that this program handles only a single *established* connection. However this is not an issue because we are interested on the backlog, which tracks all the *pending* connections.
The number of closed connections is irrelevant, as pending connections can still be accepted.

5.2.1 Verifying our hypothesis

By running this code, we were able to confirm our previous hypotheses and discovered two distinct behaviors of the socket depending on the Linux kernel version in which it is running:

- **Kernel version 3.13.1:** when using the **BACKLOG** set to 1, the SYN flood detection started after the 17th packet was sent from the attacker, we can conclude the minimum value of the backlog is 16. We have also tested with **BACKLOG** set to 30, as expected the SYN flood started after the 33rd packet was sent;
- **Kernel version 4.4.0 or later**: in this case after the **BACKLOG** value is set to 1, the SYN flood started when the second packet was sent; from this moment onwards it dropped all other requests until the backlog was not full. When using a **BACKLOG** set to 30 the SYN flood started after we sent the 31st packet; hence the actual backlog for that port was the backlog passed to the listen function.

Then we executed the original `infer_backlog_size` function using our new socket written in C. With newer Linux versions and a backlog size of 32 these are our results:

```
attacker@attacker:~/Desktop/Net_Sec_Lab/Project$ sudo python3 test2.py 192.168.4.26 8080
2024-09-04 11:39:43,558 - INFO - Testing backlog size: 8
2024-09-04 11:39:43,558 - INFO - Sending 6 SYN packets to 192.168.4.26:8080
2024-09-04 11:39:44,802 - INFO - SYN packets sent.
2024-09-04 11:39:49,814 - INFO - 40000 has flags: A
2024-09-04 11:39:49,814 - INFO - 40000 recieved ACK
2024-09-04 11:39:49,821 - INFO - 40001 has flags: A
2024-09-04 11:39:49,821 - INFO - 40001 recieved ACK
2024-09-04 11:39:49,829 - INFO - 40002 has flags: A
2024-09-04 11:39:49,829 - INFO - 40002 recieved ACK
2024-09-04 11:39:49,836 - INFO - 40003 has flags: A
2024-09-04 11:39:49,836 - INFO - 40003 recieved ACK
2024-09-04 11:39:49,843 - INFO - 40004 has flags: A
2024-09-04 11:39:49,843 - INFO - 40004 recieved ACK
2024-09-04 11:39:49,853 - INFO - 40005 has flags: A
2024-09-04 11:39:49,853 - INFO - 40005 recieved ACK
```

Figure 5.6: Testing backlog size of 8

Receiving all 8 ACKs means we didn't fill half of the backlog, hence the backlog must be greater.

```
2024-09-04 12:02:20,804 - INFO - Testing backlog size: 16
2024-09-04 12:02:20,804 - INFO - Sending 12 SYN packets to 192.168.4.22:8080
2024-09-04 12:02:23,288 - INFO - SYN packets sent.
2024-09-04 12:02:28,302 - INFO - 40000 has flags: A
2024-09-04 12:02:28,302 - INFO - 40000 recieved ACK
2024-09-04 12:02:28,309 - INFO - 40001 has flags: A
2024-09-04 12:02:28,309 - INFO - 40001 recieved ACK
2024-09-04 12:02:28,317 - INFO - 40002 has flags: A
2024-09-04 12:02:28,317 - INFO - 40002 recieved ACK
2024-09-04 12:02:28,324 - INFO - 40003 has flags: A
2024-09-04 12:02:28,324 - INFO - 40003 recieved ACK
2024-09-04 12:02:28,331 - INFO - 40004 has flags: A
2024-09-04 12:02:28,331 - INFO - 40004 recieved ACK
2024-09-04 12:02:28,339 - INFO - 40005 has flags: A
2024-09-04 12:02:28,339 - INFO - 40005 recieved ACK
2024-09-04 12:02:28,346 - INFO - 40006 has flags: A
2024-09-04 12:02:28,346 - INFO - 40006 recieved ACK
2024-09-04 12:02:28,355 - INFO - 40007 has flags: A
2024-09-04 12:02:28,355 - INFO - 40007 recieved ACK
2024-09-04 12:02:28,363 - INFO - 40008 has flags: A
2024-09-04 12:02:28,363 - INFO - 40008 recieved ACK
2024-09-04 12:02:28,370 - INFO - 40009 has flags: A
2024-09-04 12:02:28,370 - INFO - 40009 recieved ACK
2024-09-04 12:02:28,378 - INFO - 40010 has flags: A
2024-09-04 12:02:28,378 - INFO - 40010 recieved ACK
2024-09-04 12:02:28,385 - INFO - 40011 has flags: A
2024-09-04 12:02:28,385 - INFO - 40011 recieved ACK
```

Figure 5.7: Testing backlog size of 16

Receiving all 16 ACKs signifies that we didn't occupy half of the backlog and hence it must be greater

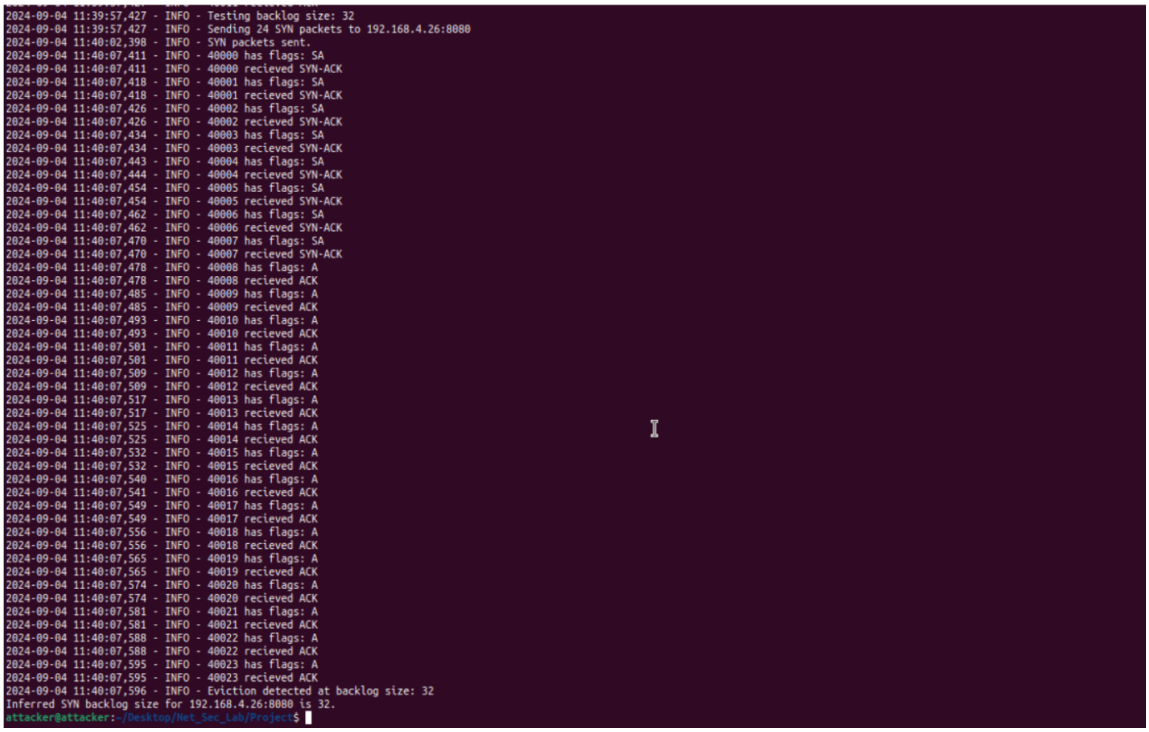


Figure 5.8: Testing backlog size of 32

We got exactly 8 SYN-ACK, meaning that we sent more packets than half of the backlog, hence the backlog must be equal to 32.

Observations

For newer Linux kernel version, the size of backlog is the actual value passed to the listen function. In the paper, in order to infer the backlog size, they start from 16 and then multiply by 2 at each iteration until the backlog value tested reach 256. For this behaviour, this methodology might not work correctly anymore.

5.2.2 Adjusting topology: Zombie upgrade

After testing the backlog size with a custom C program, we confirmed that the method used works consistently across all Linux kernel versions. As a result, we decided to upgrade our Zombie machine to the latest Kernel version to obtain results that may be more relevant and useful nowadays. The Zombie machine is now equipped with Ubuntu 24.04.1 with Kernel version: 6.8.0-41.

5.3 Finding the target

Once the tests on the zombie's backlog were successfully completed, the full scan could be executed. For simplicity, as previously mentioned, the topology included only one target machine and the scan was conducted directly on it. In more complex network setups, the code can be easily generalized in order to perform a complete scan of the subnet, rather than focusing on a specific target.

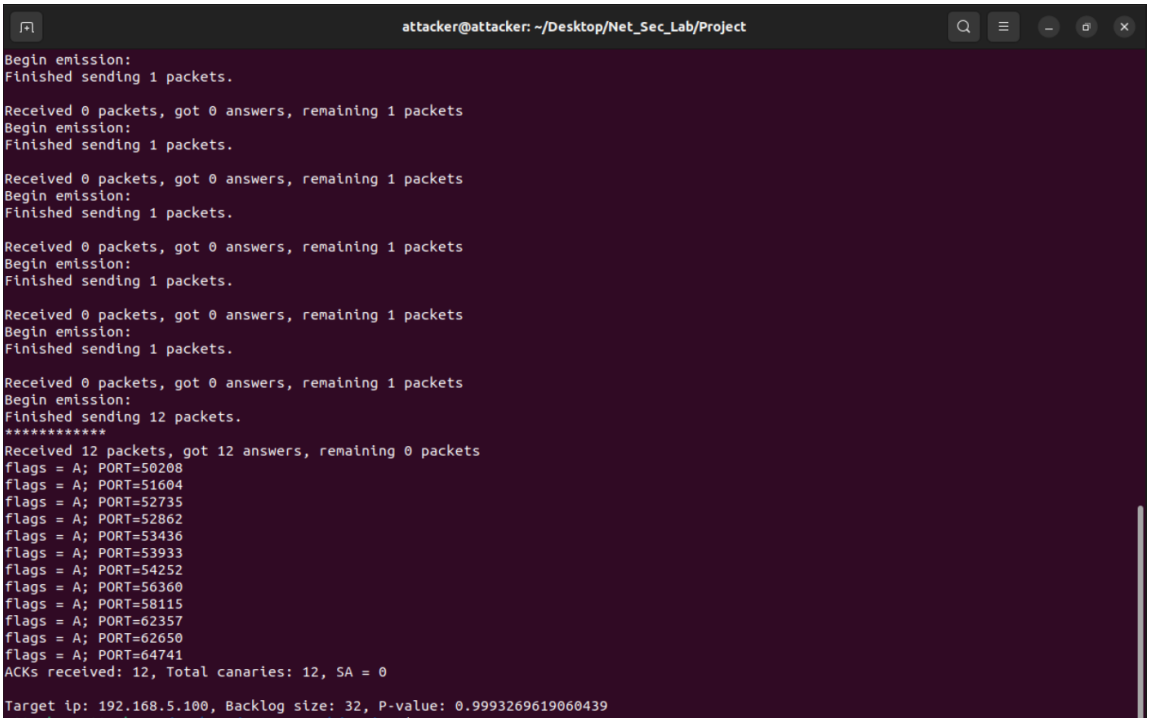


Figure 5.9: Completed scan with backlog size of 32

For this final scan we imposed a backlog size of 32 in the zombie and, by doing some calculations, we can understand that the attacker should send $32 \cdot \frac{3}{4} = 24$ packets, divided in 12 spoofed packets and 12 canaries. Then, by sending 12 probes, it received 12 ACKs and, with a p_{value} of $1 - 0.9993 = 0.0007 \approx 0$, it can safely assume that the target machine is up and reachable.

6 How eviction supposedly works

The paper claims that eviction occurs once at least half of the backlog is filled. Our analysis of the Linux Kernel code confirmed that this behaviour holds true for both newer and older kernel versions. We conducted some flood tests on our zombies machines running both older and newer Linux version, using a fixed BACKLOG size of 32 for all tests. Based on the function `infer_backlog_size` in Section 3.2 we adjusted our approach from filling only $\frac{3}{4}$ of the backlog to filling it completely.

- **First test:** we sent the first batch of 32 packets simultaneously, without waiting for any delay t between each packet. After this, we sent a second batch of 32 duplicated packets. Despite having sent more than half of the backlog with the first batch, all the answers were ACK even though we would have expected at the least 16 SYN-ACK responses if eviction has occurred as anticipated;
- **Second test:** the test setup was the same as the previous one, but with a $\frac{1}{5}$ second delay between each packet sent in the first batch of 32 packets. After sending the second batch of 32 duplicated packets the attacker received 16 SYN-ACK responses, indicating that eviction occurred as expected when a delay was introduced between packets.

These findings demonstrate that eviction is not triggered immediately once half of the backlog is filled, instead eviction is influenced by additional factors such as packet timing and delay.

Furthermore analyzing the Linux kernel we found the following functions:

```
static void reqsk_timer_handler(struct timer_list *t)
{
    ...
    timer_setup(&nreq->rsk_timer, reqsk_timer_handler, TIMER_PINNED);
    ...

    icsk = inet_csk(sk_listener);
    net = sock_net(sk_listener);
    max_syn_ack_retries = icsk->icsk_syn_retries ? :
        READ_ONCE(net->ipv4.sysctl_tcp_synack_retries);
    /* Normally all the openreqs are young and become mature
     * (i.e. converted to established socket) for first timeout.
     * If synack was not acknowledged for 1 second, it means
     * one of the following things: synack was lost, ack was lost,
     * rtt is high or nobody planned to ack (i.e. synflood).
     * When server is a bit loaded, queue is populated with old
     * open requests, reducing effective size of queue.
     * When server is well loaded, queue size reduces to zero
     * after several minutes of work. It is not synflood,
     * it is normal operation. The solution is pruning
     * too old entries overriding normal timeout, when
```

```
22          * situation becomes dangerous.
23          *
24          * Essentially, we reserve half of room for young
25          * embrions; and abort old ones without pity, if old
26          * ones are about to clog our table.
27          */
28         queue = &icsk->icsk_accept_queue;
29         qlen = reqsk_queue_len(queue);
30         if ((qlen << 1) > max(8U, READ_ONCE(sk_listener->sk_max_ack_backlog
               ))) {
31             int young = reqsk_queue_len_young(queue) << 1;
32
33             while (max_syn_ack_retries > 2) {
34                 if (qlen < young)
35                     break;
36                 max_syn_ack_retries--;
37                 young <<= 1;
38             }
39         }
40         syn_ack_recalc(req, max_syn_ack_retries, READ_ONCE(queue->
               rskq_defer_accept),
41                        &expire, &resend);
42         req->rsk_ops->syn_ack_timeout(req);
43         if (!expire &&
44             (!resend ||
45              !inet_rtx_syn_ack(sk_listener, req) ||
46              inet_rsk(req)->acked)) {
47             if (req->num_timeout++ == 0)
48                 atomic_dec(&queue->young);
49             mod_timer(&req->rsk_timer, jiffies + reqsk_timeout(req,
                   TCP_RTO_MAX));
50
51             if (!nreq)
52                 return;
53
54             if (!inet_ehash_insert(req_to_sk(nreq), req_to_sk(oreq), NULL))
                    {
55                 /* delete timer */
56                 inet_csk_reqsk_queue_drop(sk_listener, nreq);
57                 goto no_ownership;
58             }
59
60             __NET_INC_STATS(net, LINUX_MIB_TCPMIGRATEREQSUCCESS);
61             reqsk_migrate_reset(oreq);
62             reqsk_queue_removed(&inet_csk(oreq->rsk_listener)->
                   icsk_accept_queue, oreq);
63             reqsk_put(oreq);
64
65             reqsk_put(nreq);
66             return;
67         }
```

```
68
69      /* Even if we can clone the req, we may need not retransmit any
            more
70       * SYN+ACKs (nreq->num_timeout > max_syn_ack_retries, etc), or
             another
71       * CPU may win the "own_req" race so that inet_ehash_insert() fails
72       */
73      if (nreq) {
74          __NET_INC_STATS(net, LINUX_MIB_TCPMIGRATEREQFAILURE);
75  no_ownership:
76          reqsk_migrate_reset(nreq);
77          reqsk_queue_removed(queue, nreq);
78          __reqsk_free(nreq);
79      }
80
81  drop:
82      inet_csk_reqsk_queue_drop_and_put(oreq->rsk_listener, oreq);
83  }
```

```
1  static void reqsk_queue_hash_req(struct request_sock *req,
2                   unsigned long timeout)
3  {
4      timer_setup(&req->rsk_timer, reqsk_timer_handler, TIMER_PINNED);
5      mod_timer(&req->rsk_timer, jiffies + timeout);
6  }
```

We examined the `timer_setup` function, noting that its second parameter is the function discussed earlier. We have concluded that the function that handles the request and its eviction is called repeatedly until the request is removed from the backlog, either because the handshake is completed, it is removed due to TCP timeout or evicted because the backlog size was greater than $backlog/2$ and new younger requests were being added to the backlog.

```
1  /**
2   * timer_setup - prepare a timer for first use
3   * @timer: the timer in question
4   * @callback: the function to call when timer expires
5   * @flags: any TIMER_* flags
6   *
7   * Regular timer initialization should use either DEFINE_TIMER() above,
8   * or timer_setup(). For timers on the stack, timer_setup_on_stack()
        must
9   * be used and must be balanced with a call to destroy_timer_on_stack()
        .
10  */
11 #define timer_setup(timer, callback, flags)          \
12     __init_timer((timer), (callback), (flags))
```

We wanted to add some `printk` to the `reqsk_timer_handler` and recompile the whole Linux kernel in order to debug it with more precision, unfortunately we failed doing that, see Appendix B for more details.

7 Considerations and limitations

This project aimed to evaluate the relevance and applicability of the Original SYN methodology on both older and modern Linux kernel versions.
By focusing on understanding backlog behavior, estimating backlog size through iterative probing, implementing a scanning methodology using zombie machines and assessing the technique's compatibility with contemporary systems, the project sought to determine whether the Original SYN approach remains valid in today's environment.

Some considerations we want to point out are the following:

- **Backlog behaviour:** our study confirmed that SYN backlog management in Linux systems varies across different kernel versions.
  On older kernels, like v3.13.1, we observed that the minimum backlog size was effectively at least 16, even when the backlog was set to 1 in the `listen()` function.
  In contrast modern kernels, such as v4.4.0 and later, immediately responded to SYN-FLOOD attempts once the backlog was filled, leading us to the conclusion that the backlog parameter of the `listen()` function is treated as the effective backlog size;
- **Eviction Behaviour:** a critical discovery during our tests was that entries in the backlog are not immediately evicted.
  Contrary to our initial hypothesis that eviction would occur once half the backlog is filled, our findings revealed that eviction is influenced by additional factors such as packet timing and delay;
- **Compatibility on Modern Kernels:** the Original SYN method remains viable on modern Linux kernels, though adjustments are necessary.
  Our experiments showed that while the core principles of the method still apply, modern kernels introduce new challenges, such as stricter backlog management, which requires slightly different approaches for reliable executions.

The limitations encountered during the project are:

- **Controlled Testing Environment:** our decision to use a controlled test environment, rather than real networks, ensured the safety and ethical compliance of our experiments, but may have limited the realism of our findings.
  While we recreated plausible network scenarios with multiple hosts, routers and networks, the absence of real-world variables such as diverse network traffic, firewalls and SYN cookies may mean our results do not fully reflect real-world behaviours;
- **Challenges in Kernel Debugging:** our attempts to delve deeper into the eviction mechanism by adding `printk` statements to the `reqsk_timer_handler` and recompiling the Linux kernel were unsuccessful.
  This limited our ability to precisely pinpoint the causes behind the observed behaviors, leaving some questions unanswered;

- **Experimental Constraints:** some experimental choices, such as disabling firewalls and assuming the presence of a specific zombie machine, were necessary to isolate and test specific behaviors. However, these constraints also simplified the network environment, potentially overlooking real-world scenarios complexity, where such factors would play a significant role.

In conclusion, while this project successfully demonstrated that the Original SYN methodology retains its relevance, it also underlines the need for continuous adaptation in response to ongoing developments in Linux kernel architecture.
The limitations encountered during the project highlight the importance of further refinements of these techniques, ensuring their robustness, consistency and applicability across various environments and kernel versions.

A Experiment logs: testing different Kernel version

Collection of test results with different Kernel versions.

A.1 Ubuntu 11.04

Parameters configuration:

Zombie OS	Ubuntu 11.04
Kernel version	2.6.38-8
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets everything works as expected.

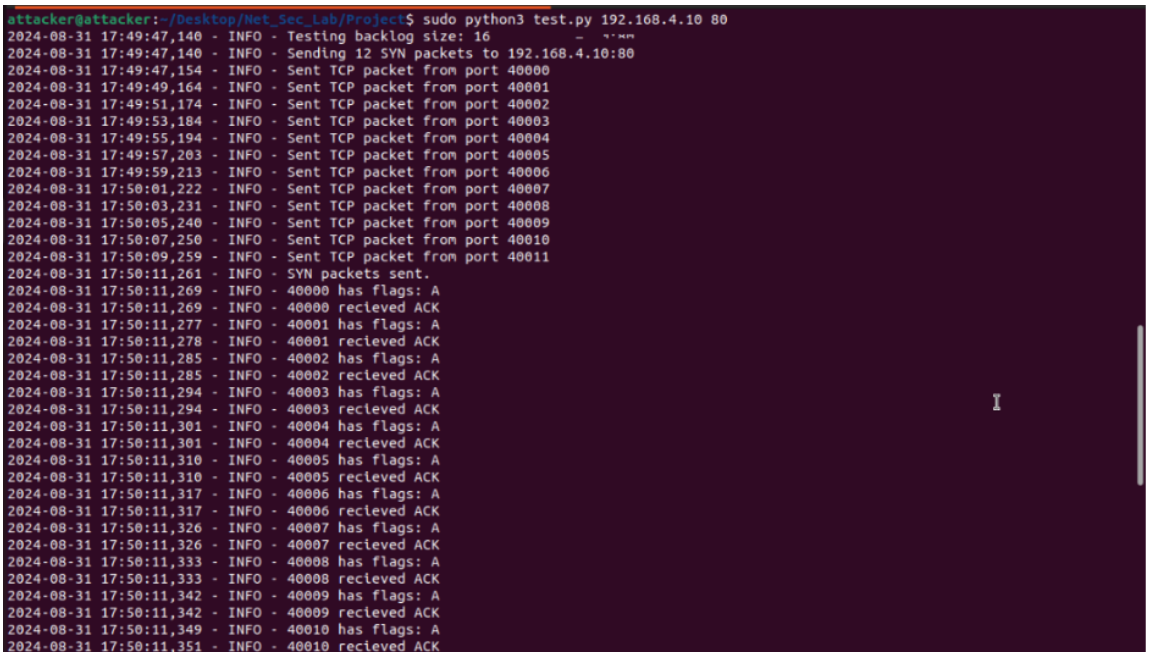


Figure A.1: Test with 16 packets

With 32 packets, after few packets, SYN-ACK are received due to SYN flooding, as confirmed by Zombie's

Kernel logs.

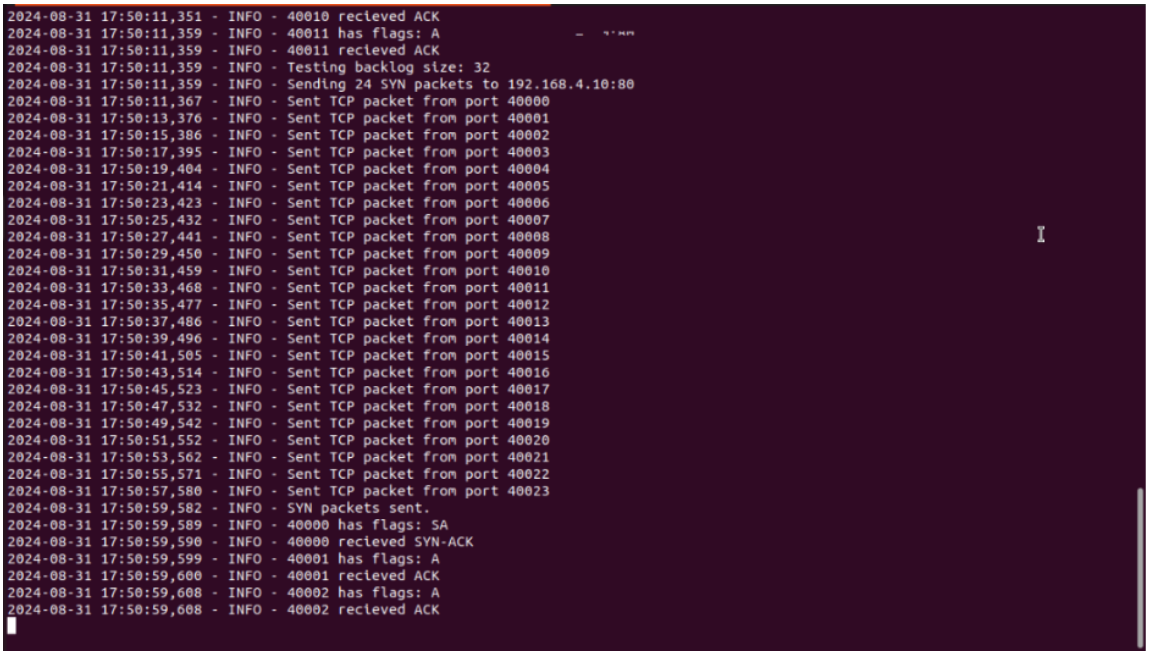


Figure A.2: Test with 32 packets

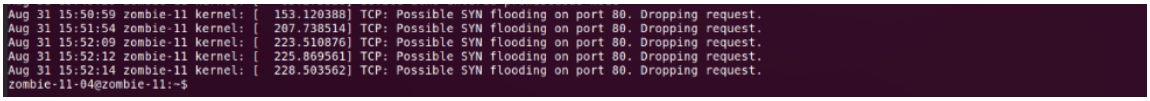


Figure A.3: Zombie's Kernel log

A.2 Ubuntu 11.10

Parameters configuration:

Zombie OS	Ubuntu 11.10
Kernel version	3.0.0-12
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets everything works as expected. With 32 packets, we noticed 7 packets missing in responses, confirmed by Zombie's Kernel logs.

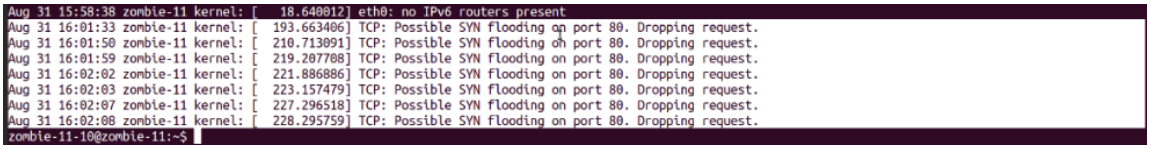


Figure A.4: From Zombie's Kernel log we can see the 7 dropped packets

A.3 Ubuntu 12.04

Parameters configuration:

Zombie OS	Ubuntu 12.04
Kernel version	3.2.0-23
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1 sec
Port	80
Port opening method	Netcat

Observed conduct: the zombie seems to behave in a more expected way: there are SYN-ACKs first and then ACKs. We don't understand that wild SYN-ACK in the middle of nowhere. Also, in kernel logs nothing has been notified.

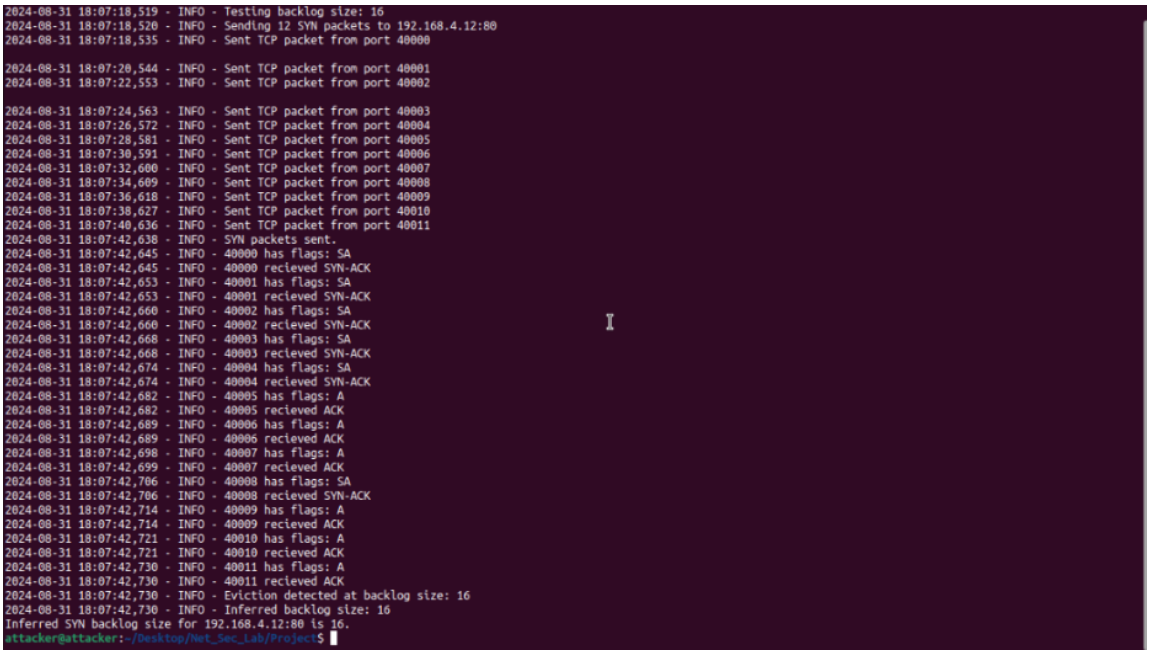


Figure A.5: Wild SYN-ACK

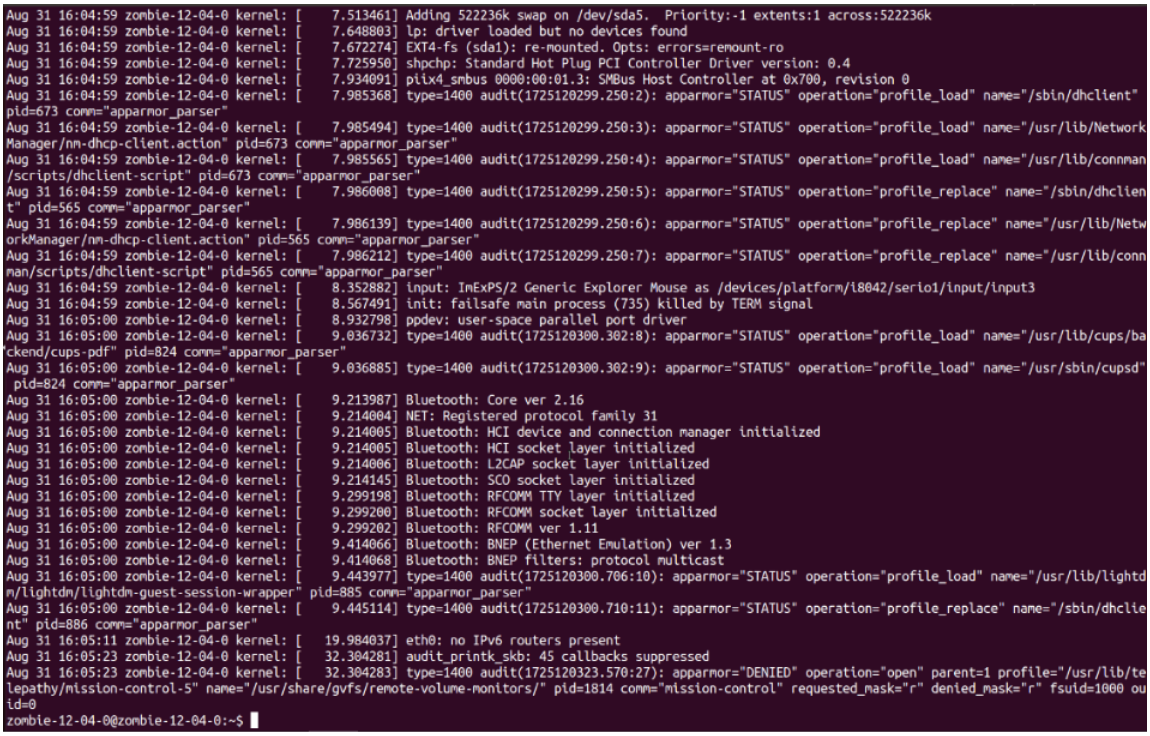


Figure A.6: Nothing in Zombie's Kernel logs

A.3.1 Rebooting

We performed the test after rebooting the zombie. No changes in parameters.
Observed conduct: the packet order seems more messed than before

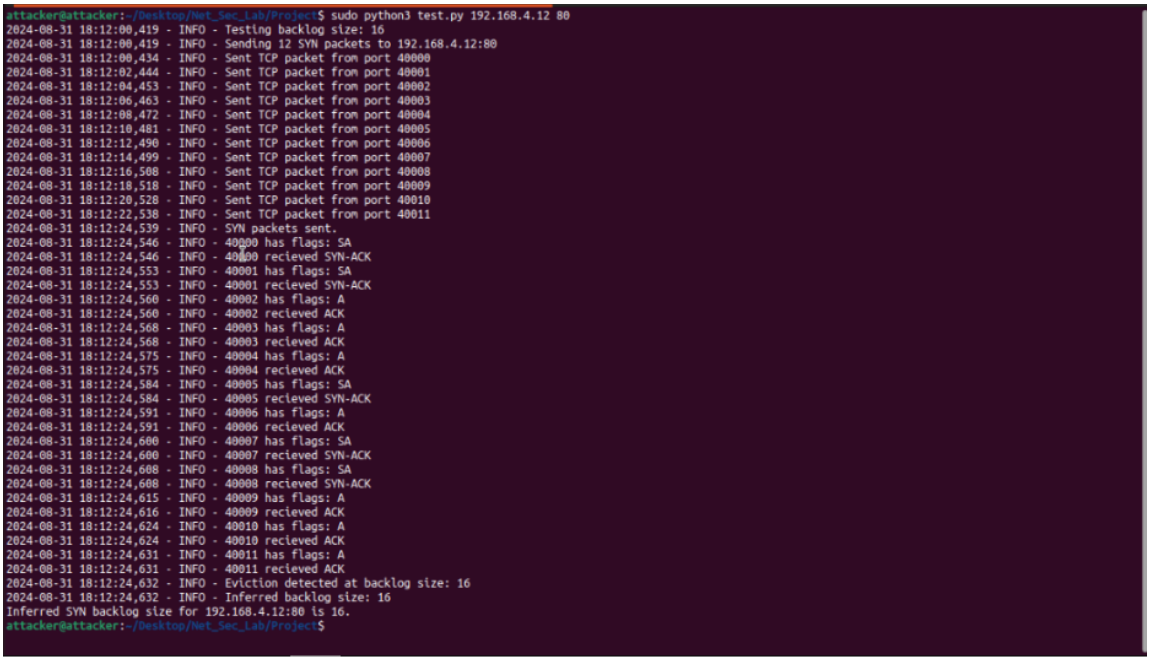


Figure A.7: More wild SYN-ACKs

A.3.2 Variant: lower timeout and fewer packets

Timeout value is now set to 1/5 seconds as suggested in the paper. The number of packet is lowered to 8.
Observed conduct: everything works as expected.

A.3.3 Variant: restoring number of packets

The initial number of packet returns to 16.
Observed conduct: with 16 packets, everything works as expected. With 32 packets, we receive SYN-ACK and an alert on Zombie's kernel log.

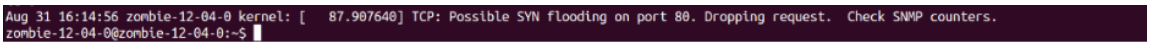


Figure A.8: Kernel alert has now a new form

A.4 Ubuntu 12.04.1

Parameters configuration:

Zombie OS	Ubuntu 12.04.1
Kernel version	3.2.0-29
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets, everything works as expected. With 32 packets, we receive SYN-ACK and an alert on Zombie's kernel log.

A.5 Ubuntu 12.04.2

Parameters configuration:

Zombie OS	Ubuntu 12.04.2
Kernel version	3.2.0-37
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets, everything works as expected. Also with 32 packets seems working, except for the first packet received that is a SYN-ACK.

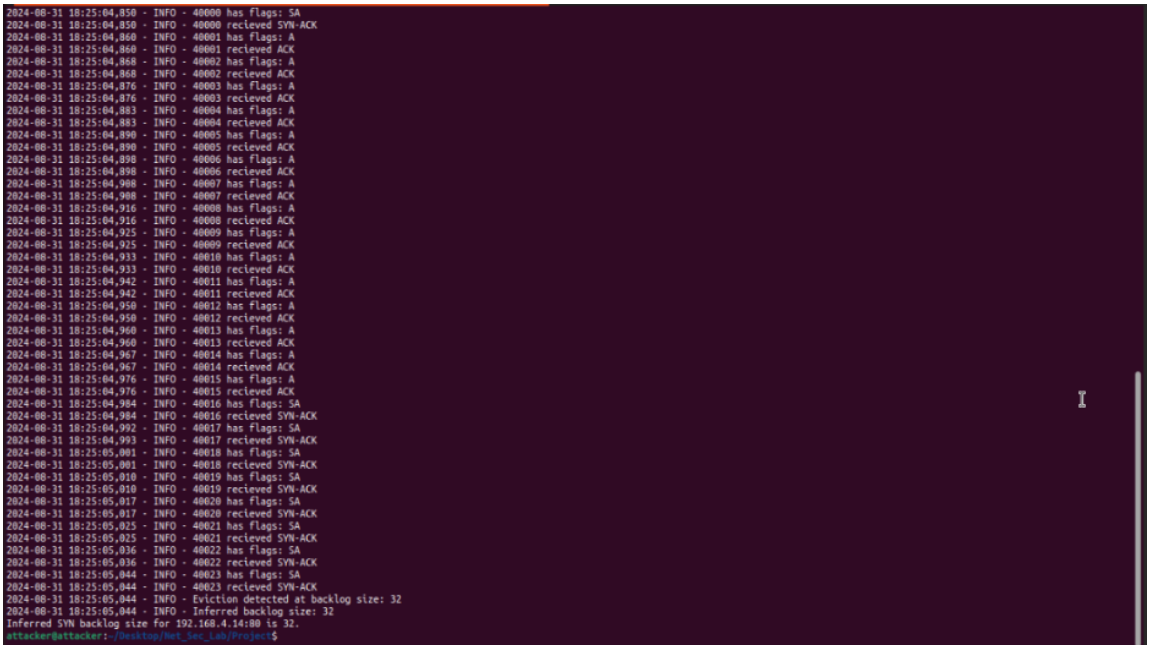


Figure A.9: Only first SYN-ACK is out of place

A.5.1 Attempt 2

We tried to reboot the zombie.
Observed conduct: everything seems work even with 32 packets.

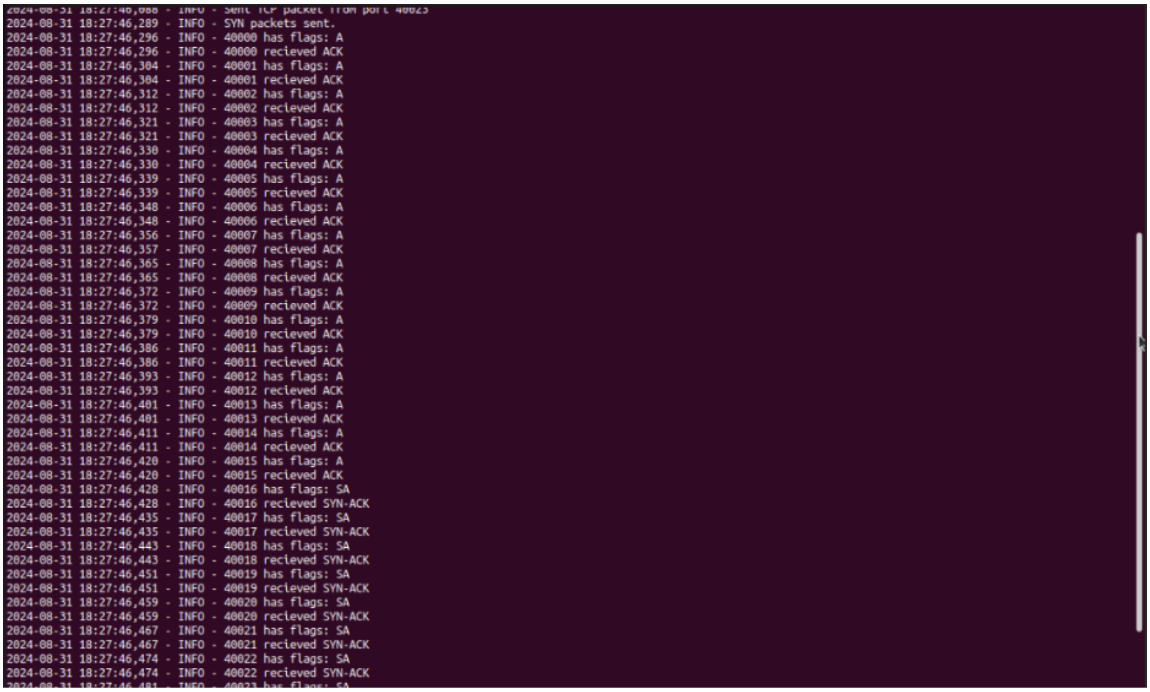


Figure A.10: Everything works as expected

A.6 Ubuntu 12.04.3

Parameters configuration:

Zombie OS	Ubuntu 12.04.3
Kernel version	3.2.0-51
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets, everything works as expected. With 32 packets we obtain a SYN flooding alert on Zombie's kernel log and a bunch of SYN-ACK as responses.

A.7 Ubuntu 12.04.4

Parameters configuration:

Zombie OS	Ubuntu 12.04.4
Kernel version	3.2.0-58
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets, everything works as expected. When testing 32 packets, we report

that during the sending of the 16th packet it crashes, the syn flood in the zombie is notified and the abnormal behaviour begins. In the zombie's logs we notice a new message: 'check SNMP counters'. Since we only see one message despite having multiple dropped packets, we assume that SNMP is the lost packet counter.

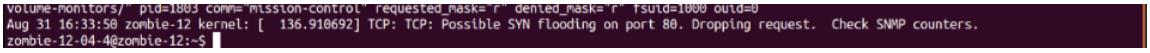


Figure A.11: Check SNMP counters

A.8 Ubuntu 12.04.5

Parameters configuration:

Zombie OS	Ubuntu 12.04.5
Kernel version	3.2.0-67
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets, everything works as expected. When testing 32 packets, after 16th packets it crashes and a SYN flooding is notified.

A.9 Ubuntu 12.10

Parameters configuration:

Zombie OS	Ubuntu 12.10
Kernel version	3.5.0-17
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets, everything works as expected. When testing 32 packets, after 16th packets it crashes and a SYN flooding is notified.

A.10 Ubuntu 13.04

Parameters configuration:

Zombie OS	Ubuntu 13.04
Kernel version	3.8.0-19
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets, everything works as expected. When testing 32 packets, after 16th packets it crashes and a SYN flooding is notified.

A.11 Ubuntu 13.10

Parameters configuration:

Zombie OS	Ubuntu 13.10
Kernel version	3.11.0-12
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets, everything works as expected. When testing 32 packets, after 16th packets it crashes and a SYN flooding is notified.

A.12 Ubuntu 14.04

Parameters configuration:

Zombie OS	Ubuntu 14.04
Kernel version	3.13.0-23
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets, everything works as expected. When testing 32 packets, after 16th packets it crashes and a SYN flooding is notified.

A.13 Ubuntu 14.04.1

Parameters configuration:

Zombie OS	Ubuntu 14.04
Kernel version	3.13.0-32
Syn cookies	Disabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: with 16 packets, everything works as expected. When testing 32 packets, after 16th packets it crashes and a SYN flooding is notified.

A.14 Ubuntu 14.04.2

Parameters configuration:

Zombie OS	Ubuntu 14.04.2
Kernel version	3.16.0-30
Syn cookies	N/A
Number of initial packets	16
Sending timeout	1/5 sec
Port	N/A
Port opening method	N/A

Observed conduct: login loop issue. Test aborted.

A.15 Ubuntu 14.04.3

Parameters configuration:

Zombie OS	Ubuntu 14.04.2
Kernel version	3.19.0-25
Syn cookies	N/A
Number of initial packets	16
Sending timeout	1/5 sec
Port	N/A
Port opening method	N/A

Observed conduct: login loop issue. Test aborted.

A.16 Ubuntu 14.04.4

Parameters configuration:

Zombie OS	Ubuntu 14.04.4
Kernel version	N/A
Syn cookies	N/A
Number of initial packets	16
Sending timeout	1/5 sec
Port	N/A
Port opening method	N/A

Observed conduct: recovery shell unavailable, login loop issue. Test aborted.

A.17 Ubuntu 14.04.5

Parameters configuration:

Zombie OS	Ubuntu 14.04.5
Kernel version	4.4.0-31
Syn cookies	disabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: SYN flooding since 1st packet. Is here where the kernel changes?

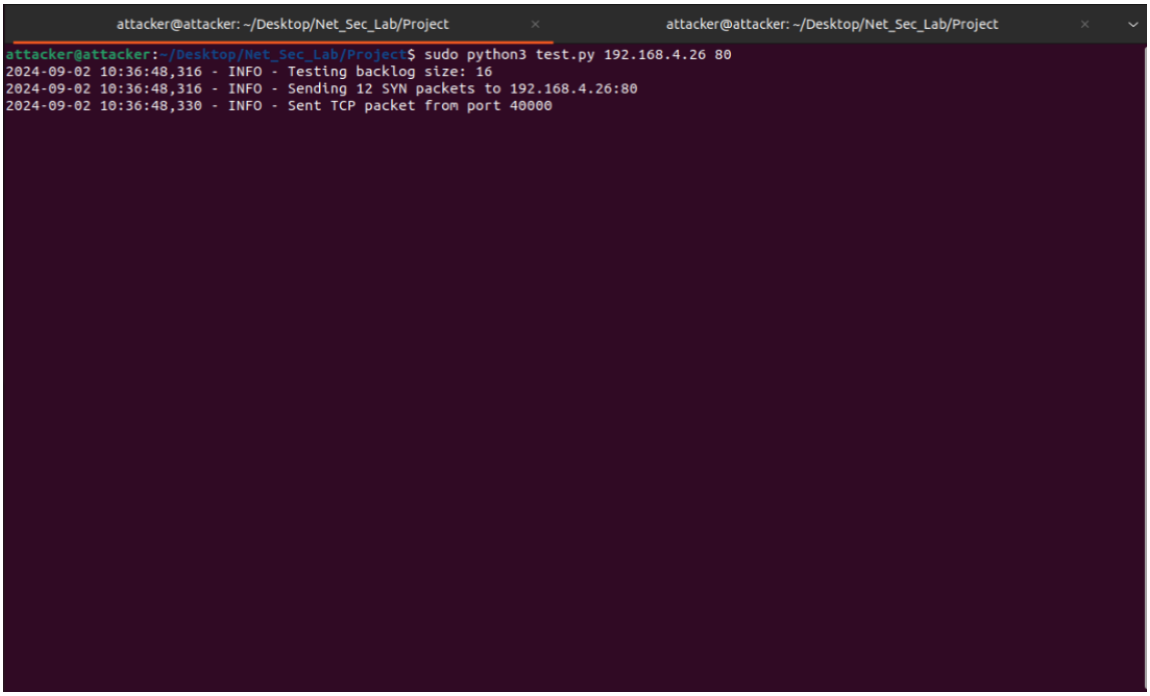


Figure A.12: Behaviour of the attacker

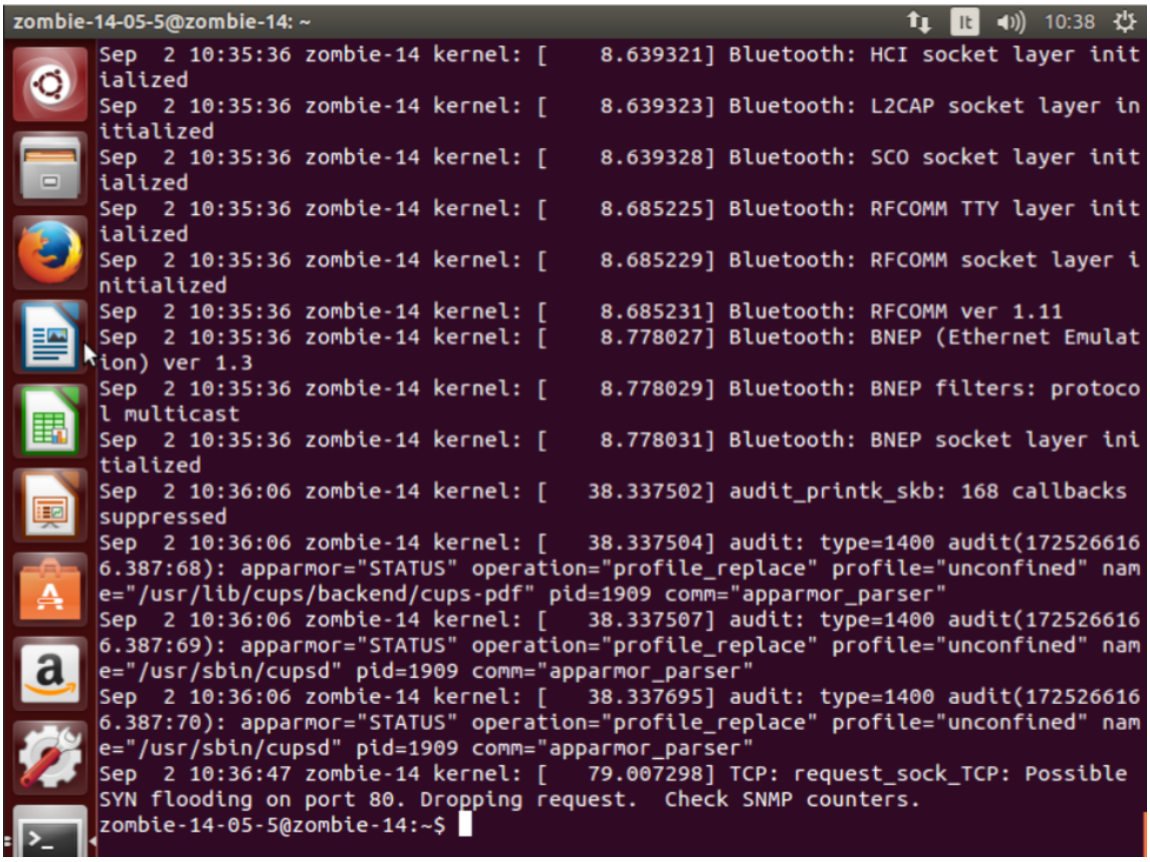


Figure A.13: Behaviour of the zombie

A.17.1 Variant: custom socket

Instead of using Netcat, we tried to implement our own socket with backlog size = 16.
Observed conduct: Some SYN-ack at the beginning.

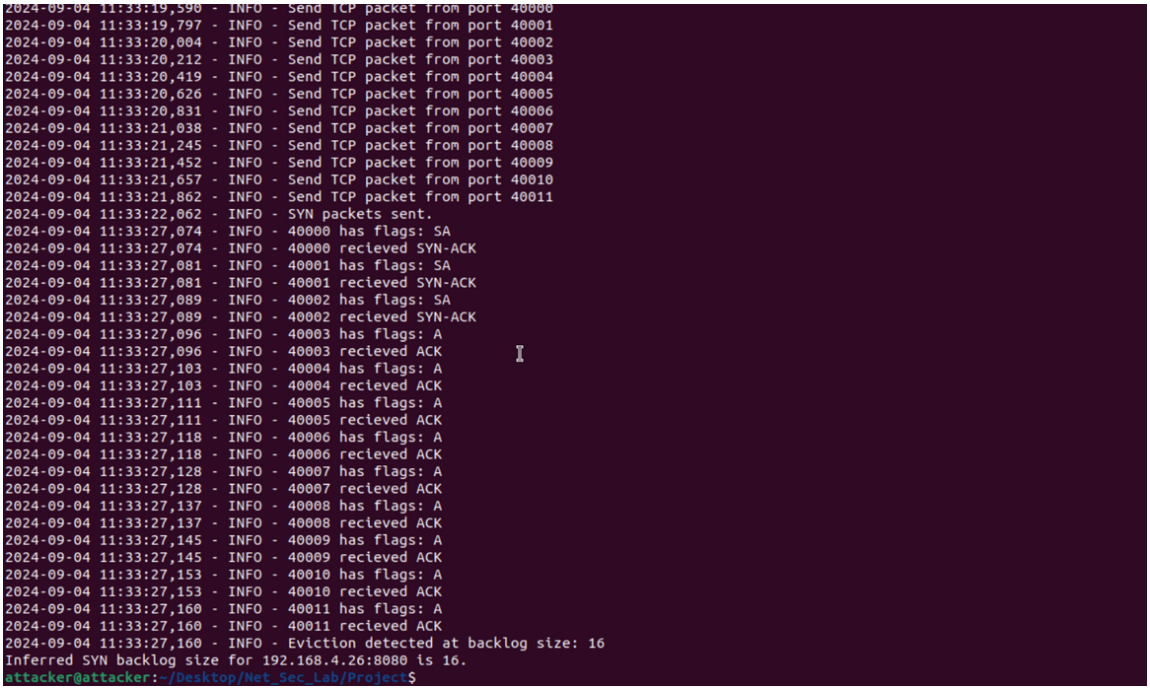


Figure A.14: Behaviour with the C socket with backlog size 16

A.17.2 Variant: lowered initial number of packets + bigger backlog

Starts sending 8 packets and proceed with 16 and 32. Zombie's custom socket is now set with backlog size = 32.

Observed conduct: The first 8 packets are SYN-ACK. The listen backlog is 32 (passed as a parameter to the socket). So we send 24 test packets (¾). Half of 32 is 16 so according to the paper after 16 packets it should start doing evictions. The first 8 packets are evicted. 24-8 = 16... Seems reasonable. The problem was Netcat?

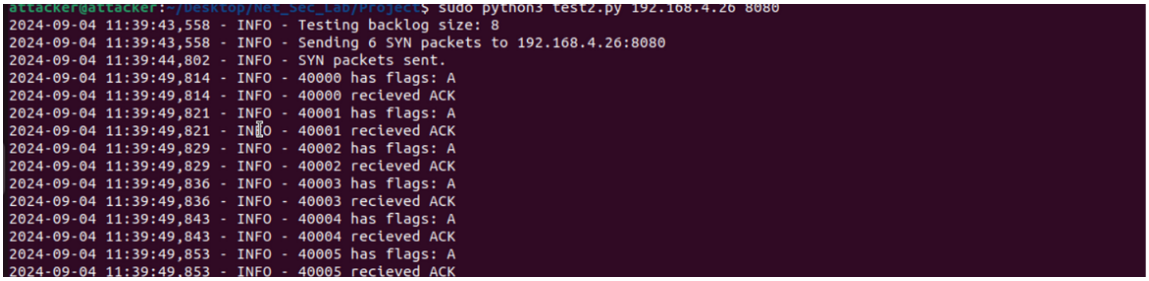


Figure A.15: All 6 packets still in the backlog

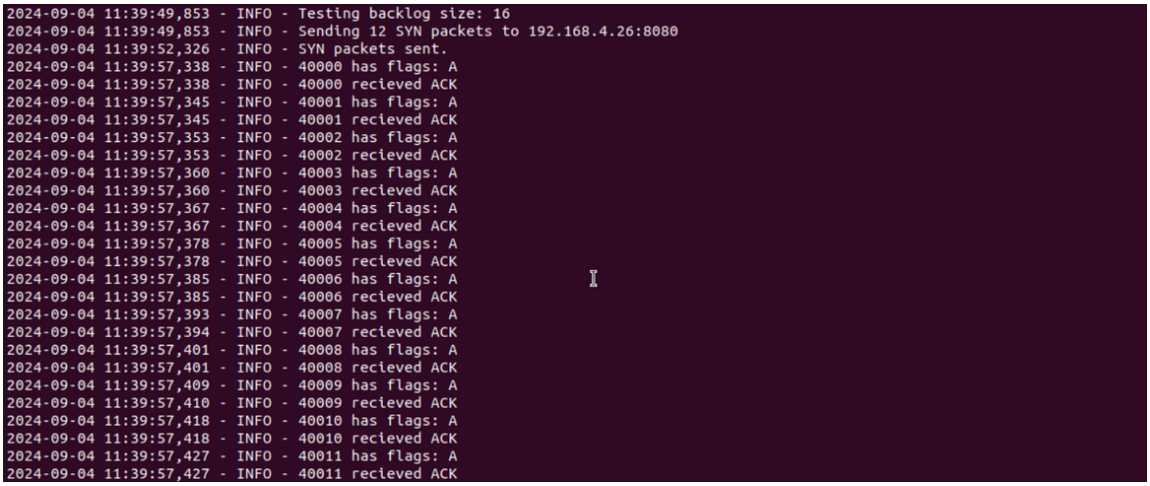


Figure A.16: All 12 packets still in the backlog

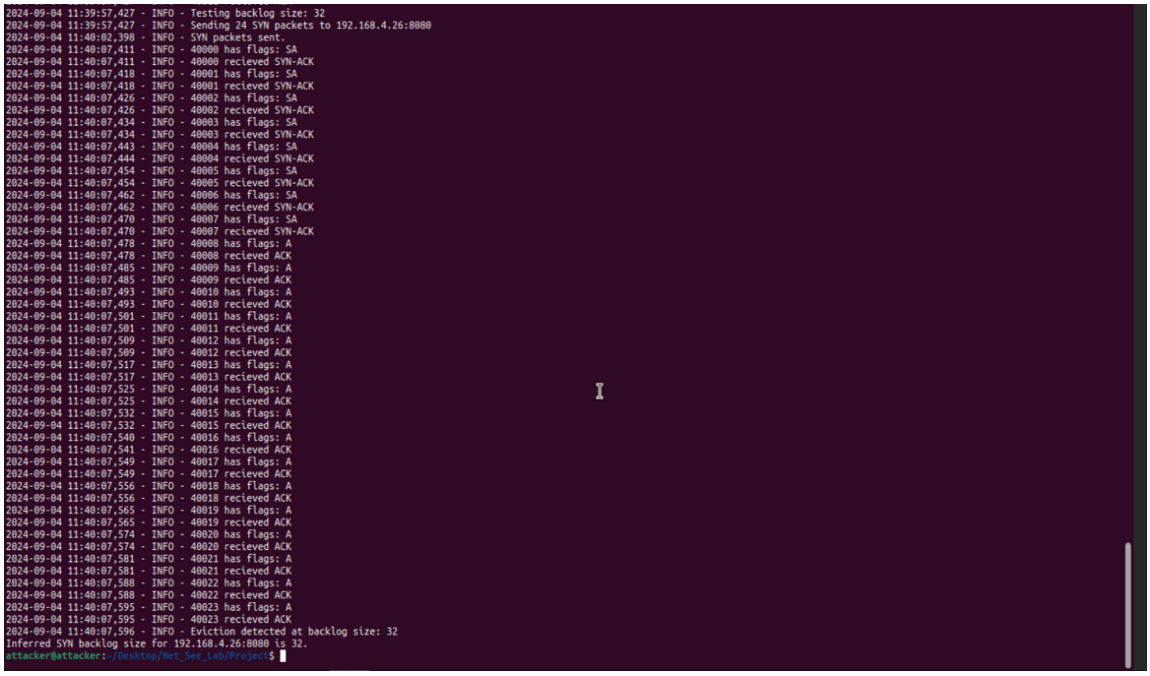


Figure A.17: The first 8 packets have been evicted

B Kernel Recompilation

B.1 How to recompile a Linux Kernel

There are some packages needed before starting building our own kernel, you can install it with the following command:

```
sudo apt build-dep linux linux-image-unsigned-$(uname -r)
```

Also be aware to have the correct *deb-src* lines in your /etc/apt/sources.list files.
You can get the source code for the current running Linux version there

```
apt source linux-image-unsigned-$(uname -r)
```

After doing all the required modification, follows those steps to build the kernel:

```
chmod a+x debian/rules
chmod a+x debian/scripts/*
chmod a+x debian/scripts/misc/*
fakeroot debian/rules clean
fakeroot debian/rules editconfigs
```

The `chmod` is needed only if you obtain the source by apt rather than git, because the way the source package is created, it loses the executable bits on the scripts.

Now change your working directory to the root of the kernel source tree

```
fakeroot debian/rules clean
# quicker build:
fakeroot debian/rules binary-headers binary-generic binary-perarch

# if you need linux-tools or lowlatency kernel, run instead:
fakeroot debian/rules binary
```

If the build is successful, several .deb binary package files will be produced in the directory above the build root directory. Install the kernel with the following command.

```
sudo dpkg -i *.deb
```

Reboot the system and the custom kernel should work.

B.2 Changes to the Linux Kernel

In our investigation we explored various logging methods provided by the Linux Kernel in order to try and debug our issue.
We modified the following Linux functions:

- `reqsk_timer_handle` placed in `net/ipv4/inet_connection_sock.c`;
- `tcp_syn_flood_action` placed in `net/ipv4/tcp_input.c`

This last file was debugged because the Linux kernel sends a warning as soon as it detects a possible SYN flooding and saves it inside the `/var/log/kern.log` file. For this reason placing some `printk` before that warning would ensure that if we saw new logs before the syn flooding, our new Linux kernel would be actually working.

Initially we employed `printk`, the primary function for printing messages form the kernel; it supports different log levels, including KERNEL_INFO and KERNEL_WARNING, which control the severity and visibility of the message. Despite our attempts to use the function with different log levels we encountered no success.
We then tried using several `pr_*` variants, which are macros designed to simplify and standardize logging across the kernel, such as `pr_info`, `pr_warn`, `pr_err`, `pr_alert`, `pr_emerg`. Despite utilizing these macros, we did not observe any changes in the output compared to our initial `printk` statements. To ensure thoroughness we also examined all relevant log files and kernel message files but found no new information or variation in the output.

B.3 The Failure :(

We undertook multiple attempts to build the kernel using both fast and complete build options. The fast build successfully completed and allowed the VM to boot without issues, however the complete build process consistently led to severe problems, causing the VM to become unbootable and forcing us to repeatedly set up a new VM from scratch.
The issue stems form complications with `initramfs`, a `cpio` archive that the kernel unpacks into RAM during boot, which it mounts and uses as the initial root filesystem. This initial filesystem allows the kernel to set up the true root filesystem in the user space. However, after the kernel recompilation, the system could no longer locate the Ubuntu partition, preventing it from booting.

Additionally, while following the kernel rebuild guide, which indicated that multiple `.deb` files should be generated, we encountered inconsistent results. In some instances we obtained several `.deb` files as expected, in others we received only one and in some cases no `.deb` files were produced at all.

Figure B.1: Unbootable kernel after complete build

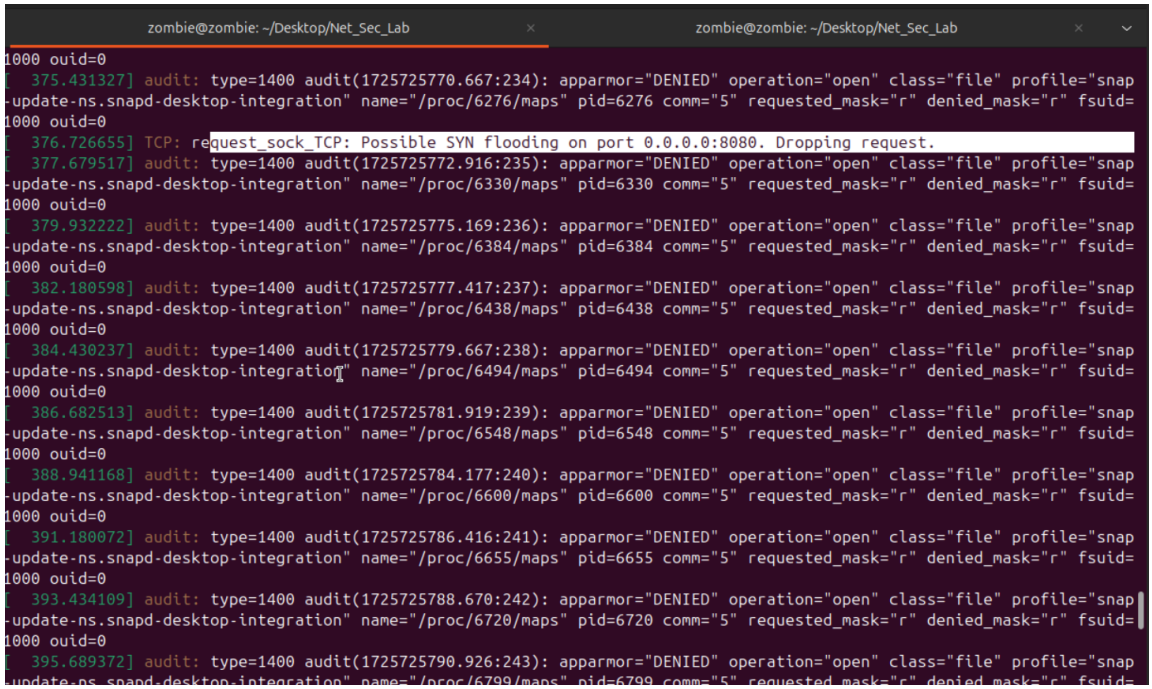


Figure B.2: Screen showing there is no printk log

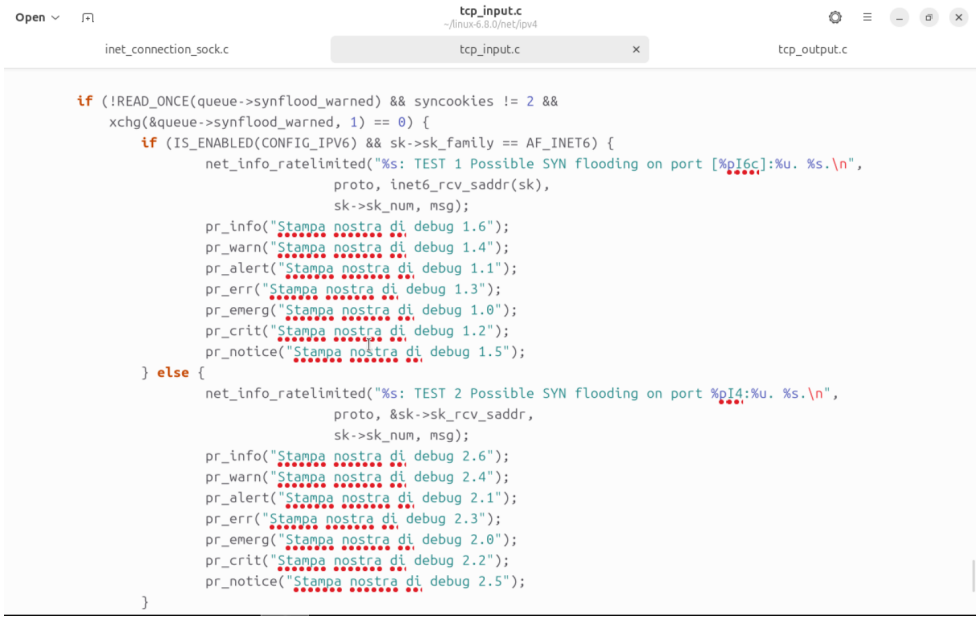


Figure B.3: Screen showing all the PR_* used

We couldn't explore further the compilation and building of the kernel due to time constraints: each compilation required hours to be completed and we had not enough time to fully understand and repeat the procedure enough times to debug successfully the kernel sections that were of interest to us.

C Other useful tests

In this appendix we compiled some of the tests that did not yield useful discoveries. Despite this, we chose to include them in the report as they provide insight into our workflow and may help others avoid the same pitfalls when implementing this scanning methodology.

C.1 Enable Syn cookies

Parameters configuration:

Zombie OS	Ubuntu 13.04
Kernel version	3.8.0-19
Syn cookies	enabled
Number of initial packets	16
Sending timeout	1/5 sec
Port	80
Port opening method	Netcat

Observed conduct: everything works as expected. Instead of dropping requests, the Zombie replies with cookies.

Figure C.1: Sending cookies instead of dropping request

C.2 Using ephemeral ports + Syn cookies

Parameters configuration:

Zombie OS	Ubuntu 11.10
Kernel version	3.11.0
Syn cookies	enabled
Number of initial packets	16
Sending timeout	1 sec
Port	50000
Port opening method	N/A

Observed conduct: same behavior as host with open port 80. Initially we were wondering if the SYN flooding control was limited to non-ephemeral ports.

C.3 Trying to find an non-existent target

While attempting to get the code to work for identifying the target, we decided to test what would happen if we entered a nonexistent target IP. Although this might seem useless, it reveals how our implementation behaves when faced with an invalid target.
In scenarios where we scan ranges of IPs rather than a single IP, being able to recognize a negative response is crucial. This test was done having a zombie with backlog size equal to 64

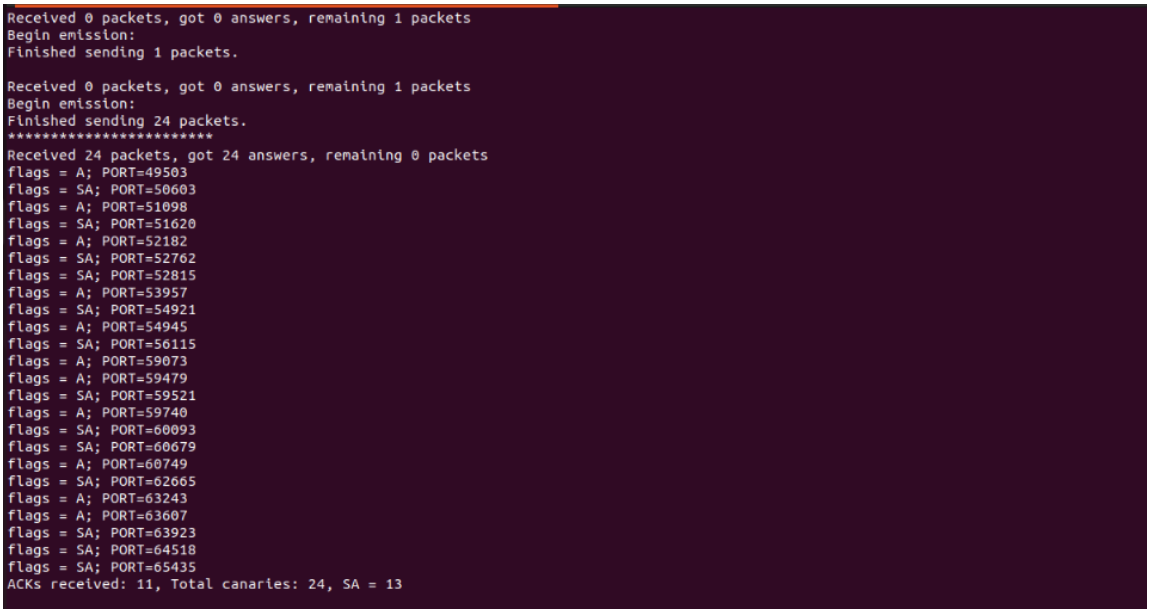


Figure C.2: Test with an invalid target IP

D Contribution of Team Members

This appendix outlines the responsibilities for the various sections that compose the final work.

The following division is not meant to highlight 'who did what' but 'who is responsible for what'.
The chosen approach allows each member to freely contribute or integrate any section of the project, rather than working in isolation and assembling everyone's work only at the end.
This cooperative effort was essential in achieving the project's goals and overcoming any challenges that arose during development.

- **Suitable Zombie machines searching** - Marco Chinellato
- **TCP/SYN Backlog Size code implementation and analysis** - Elisa Rizzo
- **Backlog scan implementation** - Martino Pistellato, Marco Chinellato
- **Statistical analysis** - Martino Pistellato, Marco Chinellato, Elisa Rizzo
- **Testing environment design and management:** Gabriele Pilotto, Thomas Vego Scocco

All tests were conducted collaboratively, as well as the process of Kernel recompilation and Kernel code analysis.
The thoughts and observations included in our report were thoroughly discussed until a consensus was reached. Major changes to the project, such as redefining objectives or assumptions, were discussed within the group members and then reviewed with Professor Leonardo Maccari, our supervisor.