ARP GUARDIAN SYSTEM

Contents

Introduction

Nowadays, the rapid proliferation of network systems has significantly heightened the importance of securing communication infrastructures, particularly within local area networks (LANs), where vulnerabilities such as ARP spoofing represent persistent threats. The Address Resolution Protocol (ARP) is essential in mapping of IP addresses to MAC addresses but it has no inherent authentication, therefore it is subject to exploitation through spoofing attacks. Such attacks enable malicious actors to intercept, manipulate, or disrupt network traffic, compromising data integrity and confidentiality. In response to this challenge, the project "ARP Guardian: ARP Spoof Detection System" presents a robust Network Intrusion Detection System (NIDS) designed to detect and visualize ARP spoofing incidents with precision and clarity. This work merges theoretical notions with practical implementation to address a critical security gap in modern networking.

This document is divided in five chapters, each of which contributes to a comprehensive understanding of the ARP Guardian System (AGS) and its importance. Chapter 1 outlines how ARP spoofing attack work and it explains the ARP protocol's basic concepts, the attack methodology, and the experimental environment setup composed by virtualized Kali Linux machines. Chapter 2 defines the scope and the structure of AGS, clarifying the adoption of the Model-View-Controller (MVC) architecture, the system's functional and non-functional requirements and the UML diagrams that illustrate its design. Chapter 3 focuses on the back-end implementation, its core logic of packet capture and spoofing detection through Python and the Scapy library. Chapter 4 is centered on the front-end, describing the web-based interface that provides real-time network visualization and user interaction. Chapter 5 deals with the limitations and the related solutions of the AGS system within real-world switched networks. In the end, a conclusion synthesizes the project's achievements and limitations.

Chapter 1

ARP Spoofing Attack

1.1 Basic Concepts

The Address Resolution Protocol (ARP) [1] is a network protocol used to map an IP address to a MAC address within a local area network (LAN). Since devices in a LAN communicate using MAC addresses, ARP plays a crucial role in translating an IP address to the corresponding MAC address before data packets can be transmitted. ARP operates at the Data Link Layer (Layer 2) of the OSI model and it is essential for proper network communication. There are two main types of ARP messages:

- **ARP Request**: A device broadcasts an ARP Request packet to the network, asking "Who has this IP address? Tell me your MAC address."
- **ARP Reply**: The device with the requested IP address responds in unicast with an ARP Reply, providing its MAC address. This response is then stored in the sender's ARP cache for future communications.

1.2 ARP Spoofing Attack

ARP spoofing (or ARP poisoning) is a type of Man-in-the-Middle (MitM) attack [2] that exploits ARP's lack of authentication. Attackers send falsified ARP messages to associate their MAC address with the IP address of a legitimate device (such as the network gateway or a victim's device). As a result, network traffic intended for the legitimate device is redirected to the attacker, who can intercept, modify, or drop the packets. The attack typically involves four main entities:

- **Attacker**: A malicious actor who sends fake ARP replies to manipulate the network's ARP tables.

- **Victim**: A user or device that unknowingly communicates with the attacker instead of the intended recipient.
- **Access Point (Network Gateway)**: The legitimate router or switch that connects devices to the Internet.

A third entity part, called **Detector**, can be deployed as a security system that monitors ARP tables and network traffic to detect anomalies and mitigate ARP spoofing attempts.

1.3 Environment setup

In order to execute an ARP spoofing attack and to test the developed **Detector System**, a virtual testing environment can be configured to carry out all activities within a controlled setting. In particular, the VMware software has facilitated the deployment of three Kali Linux virtual machines, which, in the scenario under consideration, act as the attacker, victim, and detector. Through VMware's settings, the virtual network adapters of these machines can be configured in NAT mode, allowing each to be managed independently of the host machine's network adapter. The VMware environment provides a virtual gateway to which each virtual machine is connected via the eth0 interface. This virtual gateway routes the traffic of each machine to the external network for internet access and directly connects the machines as if they were on the same LAN, as shown in Figure 1.1.

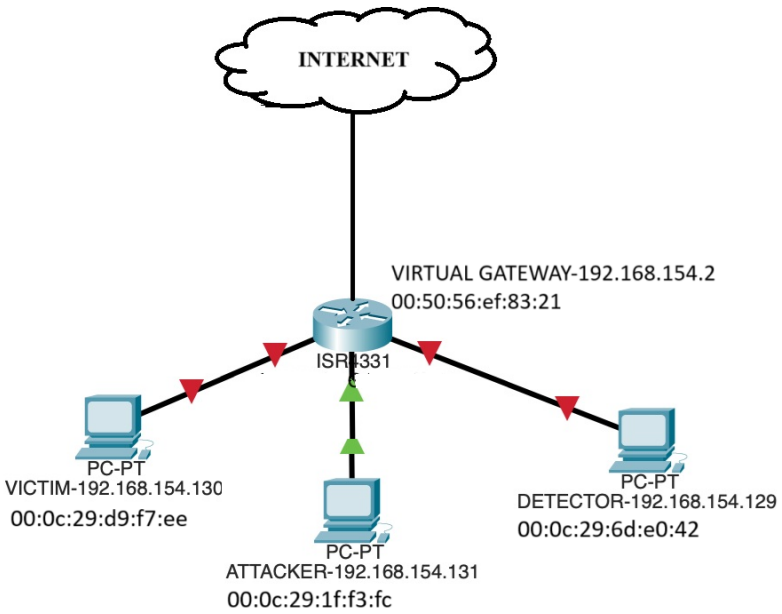

Figure 1.1: Network Topology

1.4 Steps of ARP Spoofing attack

The python script used to perform the ARP spoofing attack uses the Scapy library. It prompts the user to input a target IP and a gateway IP, then continuously sends forged ARP response packets to associate the attacker's MAC address with the gateway's IP in the target's ARP cache, and vice versa. This poisons the ARP tables, enabling a MITM attack. The script also includes a restoration function to revert the ARP tables to their original state upon interruption (via Ctrl+C). As shown in Figure 1.2, the attacker executes the attack via a terminal command. The Python code sends a large number of fake ARP Reply packets to poison both the victim's (the other kali machine) and the gateway's ARP caches.

Figure 1.2: Terminal execution of attack code

The victim's device updates its ARP table with incorrect information, believing that the attacker is the network gateway. Similarly, the access point may update its ARP table, believing the attacker is the victim. As a result, network traffic flows through the attacker, enabling packet interception, modification, or dropping of data.

The ARP cache of the victim, shown in the Image 1.3, has been poisoned: the gateway IP (192.168.154.2) and the attacker IP (192.168.154.131) are both mapped to the same MAC address (00:0c:29:1f:f3:fc), indicating a successful ARP spoofing attack where the attacker has associated their MAC address with multiple IPs.

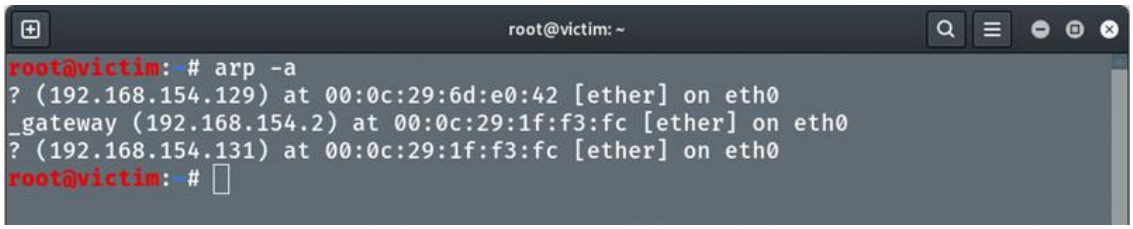

Figure 1.3: Poisoned ARP cache of the victim

Chapter 2

Project Scope and Structure

2.1 Intrusion Detection System (IDS)

An Intrusion Detection System (IDS) is a security tool designed to monitor network traffic and detect suspicious activities or policy violations. Specifically, when applied to ARP spoofing attacks, an IDS can identify unauthorized attempts to manipulate Address Resolution Protocol (ARP) traffic, such as falsified ARP responses, by analyzing packet patterns and deviations from expected network behavior. The IDS described in this project is a Network Intrusion Detection System (NIDS) implemented as an out-of-line, offline device, enabling passive monitoring without direct interference in network operations.

2.1.1 Implementation goals

The primary objective of **ARP Guardian System (AGS)** is to autonomously analyzes and reports entities conducting ARP spoofing attacks on network hosts, while also identifying and marking the affected victims. The findings will be presented visually through an intuitive web-based graphical interface, depicting the network as a graph. This solution is designed to equip IT administrators with a clear and comprehensive insight into their network's security status.

2.2 Design Decision

The architecture of the system has been structured around the Model-View-Controller (MVC) design pattern, which serves as the foundational framework for organizing components and classes. This selection was driven by the specific technical requirements of the project and the well-documented advantages inherent in this architectural approach. The following sections outline the rationale for integrating MVC into the system design and the benefits it delivers.

2.2.1 Rationale for Selection

Design choices play a crucial role in shaping a system's architecture, features, and non-functional attributes, and altering them after they've been established can be both challenging and expensive. These choices pertain to fundamental elements of the design, including:

- **Structure**: MVC ensures that the *Model* manages data and core operations, the *View* handles presentation, and the *Controller* orchestrates interactions between them. Communication among the system's classes is governed by this framework, promoting a modular and transparent approach to code management. Additionally, this structure enhances testability and maintainability, as each component can be independently developed, modified, or validated without affecting the others, thereby supporting long-term system evolution. Communication between the various components of the system occurs directly between the classes, which interact with each other via methods and attributes defined within the MVC context.

- **Technologies**: Python was selected as the primary programming language for its straightforward syntax, versatility, and access to powerful libraries such as Scapy, which enables efficient network scanning and packet analysis. Scapy, in particular, was chosen for its advanced capabilities in network protocol manipulation, critical to the system's analytical objectives.

- **Communication**: Communication between the various components of the system occurs directly between the classes, which interact with each other via methods and attributes defined within the MVC context.

- **Scalability**: AGS is designed to scale efficiently across different network environments, ensuring consistent performance and adaptability regardless of the network size or complexity.

2.3 Programming Languages and Libraries

AGS has been developed leveraging a suite of technologies, specifically Python, Flask, HTML, CSS, and JavaScript, each selected for its distinct contribution to the system's functionality. Python serves as the primary programming language, underpinning the core logic and backend operations due to its robust capabilities and extensive library support. Flask, a lightweight Python web framework, has been employed to facilitate the creation of a responsive web server, enabling efficient handling of HTTP requests and dynamic content generation. On the frontend, HTML provides the structural foundation for the user interface, defining the layout and content presentation. CSS enhances the visual design, ensuring a polished

and user-friendly aesthetic through styling and formatting. JavaScript contributes interactivity, enabling real-time updates and dynamic behavior within the interface to improve user engagement and system responsiveness. Together, these technologies form a cohesive stack that supports the development of a secure, functional, and accessible ARP Spoofing detector.

2.4 Requirements

2.4.1 Functional Requirements

The functional requirements that must be adhered to are:

- **LAN visibility**: Actively identify the various hosts present on the network.
- **ARP packet analysis**: The detector must be capable of capturing and analyzing ARP packets transmitted across the LAN (both malicious and legitimate) to detect potential suspicious behavior.
- **Continuous updating**: The detector must rapidly and synchronously update the network status in accordance with its behavior.
- **Accuracy**: The detector must accurately identify and highlight both the attacker and the victim.
- **Clearness**: The detector GUI must accurately rapresent network topology state.

2.4.2 Non Functional Requirements

Non-functional requirements pertain to the attributes that characterize the overall quality of a system, distinct from its specific operational features. Often termed "quality attributes," these requirements outline the system's performance across different scenarios and its adherence to particular benchmarks. They encompass aspects such as:

- **Performance**:AGS must analyze all ARP reply packets transmitted within the LAN to promptly and efficiently respond to potential replies indicative of an ARP spoofing attack, with a refreshing time of 1 second.
- **Scalability**: AGS must scale across different LANs.
- **Reliability**: AGS must detect the attacker and the victims with a success rate of 99%.

- **Usability**: The user interface must be designed to be intuitive and accessible for users without specialized expertise, featuring straightforward instructions and editable default settings.
- **Compatibility**: The software shall ensure compatibility with leading operating systems, including Windows, macOS, and Linux.
- **Maintainability**: The codebase must be structured in a modular fashion to support seamless future updates and the integration of additional features.

2.5 UML Diagram

The UML Class Diagram, shown in Figure 2.1 provides a comprehensive structural representation of AGS, a Flask-based web application designed to detect ARP spoofing attacks within a local network. The diagram adheres to the Model-View-Controller (MVC) architectural pattern, delineating the system's components into three distinct layers: Model, Controller, and View.

Each class and component is annotated with attributes and methods, with relationships depicted to illustrate dependencies and aggregations, rendered in a color-coded format for enhanced clarity. The **Model** layer, responsible for data management and core logic, comprises three classes:

- **ARPCache**, depicted in lavender, serves as the central data repository and detection engine. It maintains a baseline of trusted IP-to-MAC mappings (baseline_cache), tracks live devices (devices), and logs spoofing events (events). Key methods include update(ip: str, mac: str) for detecting spoofing attempts and get_devices() for retrieving the current network state. A background thread (scan_thread) ensures periodic network scans via _periodic_scan().
- **NetworkDevice**, shown in lavender, encapsulates individual device properties such as ip, mac, and status flags (attacked, is_attacker, is_gateway). It provides a to_dict() method for serialization.
- **ARPEvent**, rendered in lavender, records details of detected spoofing incidents, including victim_ip, spoofed_mac, and timestamp, with a to_dict() method for data export.

The **Controller** layer, facilitating interaction between the Model and View, includes:

- **ARPSpooferDetector**, in light blue, manages real-time packet sniffing. It holds a reference to **ARPCache**(cache) and uses a thread (thread) to execute _sniff_thread(iface: str) for capturing ARP packets, processed via process_packet(packet: Packet). Public methods start_sniffing() and stop_sniffing() control its operation.

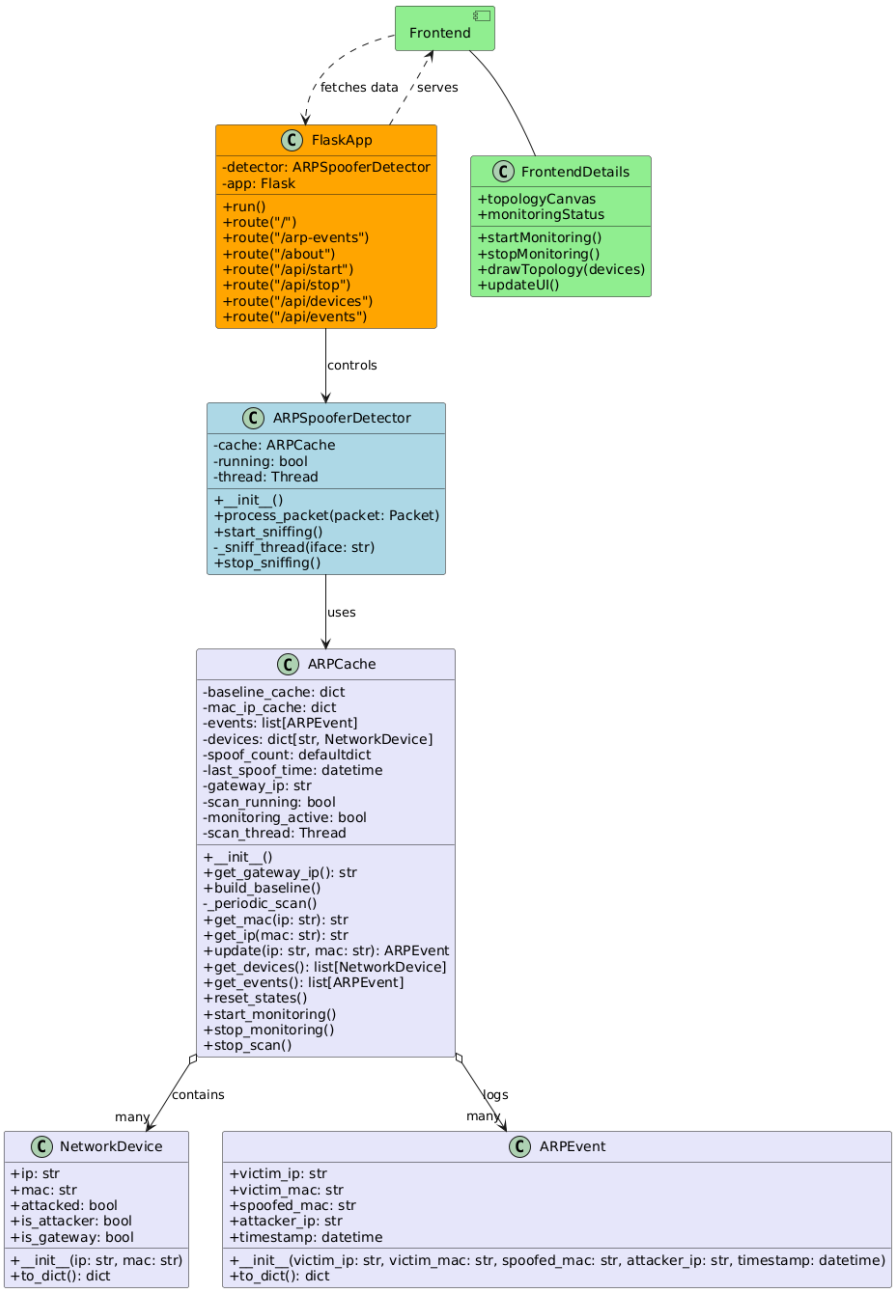

Figure 2.1: UML graph

- **FlaskApp**, in orange, represents the Flask web application (app) and orchestrates system coordination through an instance of ARPSpooferDetector (detector). It exposes routes such as /api/start and /api/devices to handle user requests and data retrieval.

The **View** layer, responsible for user interaction, is modeled as:

- **Frontend**, a light green-colored component, representing the web interface (index.html, app.js, dashboard.css). Its internal structure is detailed in a nested class, FrontendDetails, which includes attributes like topologyCanvas and methods such as drawTopology(devices) for rendering the network topology. The composition relationship (Frontend – FrontendDetails) indicates that FrontendDetails encapsulates the functional specifics of the View.

Relationships within the diagram highlight system interactions:

- ARPSpooferDetector uses ARPCache (solid arrow), reflecting its reliance on the model for spoofing detection.
- ARPCache aggregates multiple NetworkDevice and ARPEvent instances (open diamond arrows), managing collections of devices and events.
- FlaskApp controls ARPSpooferDetector (solid arrow), initiating its operations.
- FlaskApp serves Frontend (dotted arrow), delivering HTML and API responses, while Frontend fetches data from FlaskApp (dotted arrow), polling for updates.

This Class Diagram effectively encapsulates the static architecture of AGS, providing a clear blueprint of its modular design and the interplay between its MVC components.

2.6 UML Sequence Diagram

The UML Sequence Diagram in Figure 2.2, illustrates the dynamic behavior of AGS during the process of initiating network monitoring, detecting an ARP spoofing attack, and updating the user interface accordingly. This diagram traces the interactions among key system participants over time, from user input to backend processing and frontend visualization, employing a color-coded scheme to distinguish roles within the Model-View-Controller (MVC) framework.

The participants involved are:

- **User**: in red, representing the human operator interacting with the system via a web browser.
- **Frontend (Fe)**: in light green, encompassing the client-side logic (app.js) and interface (index.html).
- **FlaskApp (FA)**: in orange, the Flask application (app.py) serving as the central controller.

- **ARPSpooferDetector (DET)**: in light blue, the packet-sniffing controller (detector.py).
- **ARPCache (CACHE)**: in lavender, the primary model component (arp_cache.py) managing network state.
- **NetworkDevice (ND)**: in lavender, representing device objects updated during detection.
- **ARPEvent (EVENT)**: in lavender, logging spoofing incidents.

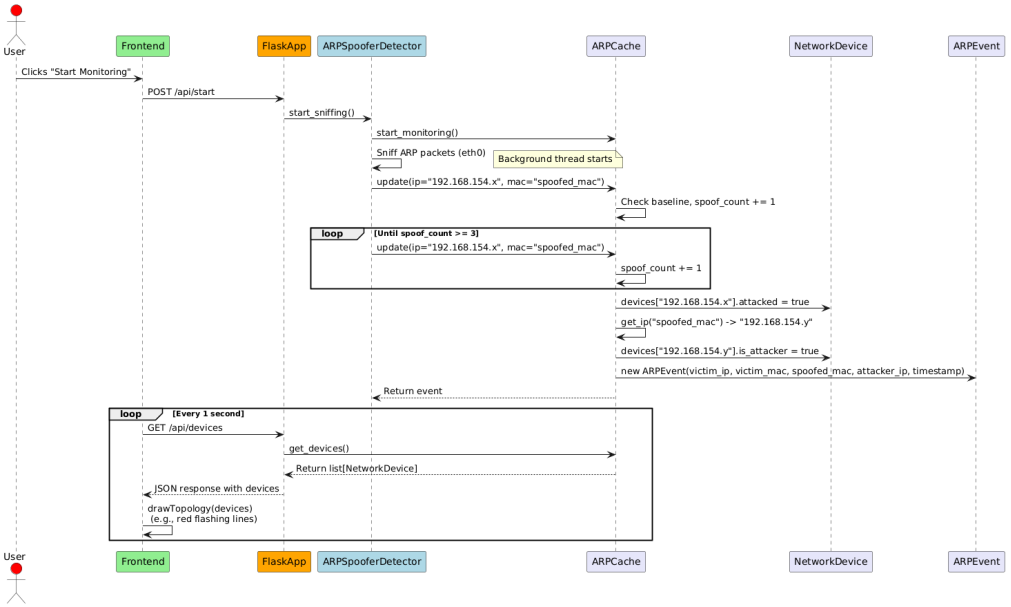

Figure 2.2: UML Sequence Diagram

The sequence unfolds as follows:

1. *Initiation of Monitoring*: The User triggers the process by clicking "Start Monitoring" on the Frontend. This action prompts Frontend to send a POST /api/start request to FlaskApp.

   FlaskApp invokes ARPSpooferDetector.start_sniffing(), which, in turn, calls ARPCache.start_monitoring() to enable spoofing detection. ARPSpooferDetector then begins sniffing ARP packets on the eth0 interface in a background thread, as noted in the diagram

2. *Detection of ARP Spoofing*: As ARPSpooferDetector captures ARP packets, it repeatedly calls ARPCache.update with packet data. ARPCache checks the baseline IP-to-MAC mapping and increments spoof_count for each mismatch. A loop (Until spoof_count >= 3) simulates three spoofed packets, ensuring

detection reliability. Upon reaching the threshold, ARPCache updates the victim device, resolves the attacker's IP via get_ip("spoofed_mac"), marks it as the attacker, and instantiates an ARPEvent to log the incident. The event is returned to ARPSpooferDetector.

3. *UI Update*: Concurrently, Frontend polls FlaskApp with GET /api/devices every second (modeled as a loop). FlaskApp retrieves the updated device list from ARPCache.get_devices() and returns it as a JSON response. Frontend processes this data in drawTopology(devices), rendering visual indicators such as red flashing lines to denote the attack.

Chapter 3

Back-end Implementation

This section outlines the features of the ARP Spoof Detector System and explains how these features will be implemented in the back-end. The primary challenge in implementing the AGS system undoubtedly lies in adopting an effective and robust technique for capturing ARP packets and distinguishing legitimate replies from malicious ones. The Python library Scapy provides powerful tools for analyzing and crafting packets across all layers of the network stack, thereby enabling the development of a technique that, by comparing ARP replies in real-time, identifies requests that deviate from a baseline stored on the detector machine. This baseline contains the authentic associations between hosts and their corresponding MAC addresses.

3.1 Model functions

The model is structured around three main classes: **NetworkDevice**, **ARPEvent**, and **ARPCache**, each serving a distinct purpose in tracking devices, logging events, and managing the detection process.

The NetworkDevice class represents individual devices on the network, encapsulating attributes such as IP address, MAC address, and flags indicating whether the device has been attacked, is an attacker, or serves as the network gateway.

The ARPEvent class is responsible for logging instances of detected ARP spoofing. It records critical details, including the target IP, the original and spoofed MAC addresses, the attacker's IP, and a timestamp. This class ensures that each spoofing incident is documented with sufficient context for later analysis or reporting.

The core functionality resides in the ARPCache class, which leverages the Scapy library to perform network scanning and ARP monitoring. It establishes a baseline of legitimate IP-MAC mappings by broadcasting ARP requests across the subnet and storing the responses. The self.baseline_cache maps IP to MAC and serves as a trusted reference to detect ARP spoofing; self.mac_ip_cache maps MAC to IP

to quickly identify attackers after spoofing is detected. Both enable O(1) lookups in opposite directions, ensuring fast, real-time monitoring. Separating them improves performance, accuracy, and baseline integrity.

It also identifies the network gateway using system routing information. The class continuously scans the network in a background thread to maintain an up-to-date list of active devices with the *_periodic_scan* function, while a separate monitoring mode can be activated to detect anomalies. When a mismatch between a device's baseline (Code 3.1) MAC address and a newly observed MAC address is detected multiple times, the system flags it as a spoofing attempt as shown in Code 3.2, identifies the attacker's IP, and logs the event. A counter (*spoof_count*) ensures that the sniffed packet is part of an attack and not an error. Additional methods retrieve the list of devices and events, resetting states after a timeout, and controlling the monitoring and scanning processes.

Listing 3.1: Baseline function

```
1
2  def build_baseline(self):
3          arp_request = scapy.ARP(pdst="192.168.154.0/24")
4          broadcast = scapy.Ether(dst="ff:ff:ff:ff:ff:ff")
5          arp_request_broadcast = broadcast / arp_request
6          answered_list = scapy.srp(arp_request_broadcast, timeout=5,
               verbose=False)[0]
7          for sent, received in answered_list:
8              self.baseline_cache[received.psrc] = received.hwsrc
9              self.mac_ip_cache[received.hwsrc] = received.psrc
10             self.devices[received.hwsrc] = NetworkDevice(received.
                   psrc, received.hwsrc)
11         self.baseline_cache[detector_ip] = detector_mac
12         self.mac_ip_cache[detector_mac] = detector_ip
13         self.devices[detector_mac] = NetworkDevice(detector_ip,
               detector_mac)
```

Listing 3.2: Core part of the detector logic (ARPCache: Model class)

```
1                   if real_mac != mac:
2                   self.spoof_count[(ip, mac)] += 1
3                   self.last_spoof_time = now
4                   # If 3+ spoof attempts, confirm attack
5                   if self.spoof_count[(ip, mac)] >= 3:
6                       attacker_ip = self.get_ip(mac)
7                       if attacker_ip and attacker_ip != ip:
8                           event = ARPEvent(ip, real_mac, mac,
                                attacker_ip, now)
9                           self.events.append(event)
10                          # Mark the victim as attacked
11                          self.devices[ip].attacked = True
12                          # Mark the attacker if already known
13                          if attacker_ip in self.devices:
```

```
14                                  self.devices[attacker_ip].is_attacker =
                                        True
15                              else:
16                                  # Add attacker as a new device
17                                  self.devices[attacker_ip] =
                                        NetworkDevice(attacker_ip, mac)
18                                  self.devices[attacker_ip].is_attacker =
                                        True
19                              # Reset spoof counter after confirmed
                                    attack
20                              self.spoof_count[(ip, mac)] = 0
21                              return event
22                  else:
23                      # Reset spoof counter if MAC matches baseline
24                      self.spoof_count[(ip, mac)] = 0
25                      # Reset states if no spoofs for 5.2 seconds
26                      if now - self.last_spoof_time > timedelta(seconds
                            =5.2):
27                          self.reset_states()
28              else:
29                  # New device: add to baseline and caches
30                  self.baseline_cache[ip] = mac
31                  self.mac_ip_cache[mac] = ip
32                  if ip not in self.devices:
33                      self.devices[ip] = NetworkDevice(ip, mac)
34                      if ip == self.gateway_ip:
35                          self.devices[ip].is_gateway = True
36          return None
```

3.2 Controller functions

The ARPSpooferDetector class is responsible for the management of the detection process by capturing ARP packets from the network and also the delegation of the analysis to the ARPCache model. In the inizialization, the constructor initializes an **ARPCache** object to store IP-MAC mappings and detect inconsistencies (potential spoofing). In the Code 3.3 each ARP packet is captured to extract the source IP (`psrc`) and the MAC (`hwsrc`) from the ARP header.

Listing 3.3: process_packet method

```
1 def process_packet(self, packet):
2     if packet.haslayer(ARP):
3         ip = packet[ARP].psrc
4         mac = packet[ARP].hwsrc
5         self.cache.update(ip, mac)
```

The *start_sniffing()*, reported in the Code 3.4 method initializes the monitoring process by first verifying that no duplicate thread is active (if not self.running).

Upon confirmation, it enables attack detection in the ARPCache model via *start_monitoring()* and launches a daemon thread (*sniff_thread*) to capture ARP packets on the specified network interface (eth0). This threaded approach prevents blocking the main application during packet processing. The *_sniff_thread()* method in Code 3.5 operates in a loop, utilizing *Scapy's sniff()* function to filter and process ARP packets exclusively (filter="arp"). Each captured packet is passed to the *process_packet* callback for analysis. A one second timeout allows periodic checks of the *self.running* flag, ensuring responsiveness to termination requests.

Listing 3.4: start_sniffing method

```
def start_sniffing(self):
    if not self.running:
        self.running = True
        self.cache.start_monitoring()
        self.thread = threading.Thread(target=self._sniff_thread,
            args=("eth0",))
        self.thread.start()
```

Listing 3.5: _sniff_thread method

```
def _sniff_thread(self, iface):
    while self.running:
        sniff(iface=iface, filter="arp", prn=self.process_packet,
            store=0, timeout=1)
```

To halt monitoring, *stop_sniffing()* in Code 3.6 sets *self.running* to False, disabling spoofing checks in the **ARPCache** but preserving device tracking. The method then attempts to join the sniffing thread with a two seconds timeout, forcibly releasing resources if the thread fails to terminate cleanly and issuing a warning. This design guarantees robust thread management and prevents resource leaks during prolonged operation.

Listing 3.6: stop_sniffing method

```
def stop_sniffing(self):
    if self.running:
        self.running = False
        self.cache.stop_monitoring()
        if self.thread:
            self.thread.join(timeout=2)
            if self.thread.is_alive():
            self.thread = None
```

Chapter 4

Front-end Implementation

In this last section it is descripted the Frontend of the AGS which constitutes the View layer within its MVC architecture, implemented through a combination of static web assets: index.html, app.js, and dashboard.css and others, located in the static directory. This component provides an interactive web interface, hosted by the Flask application at http://127.0.0.1:5000, enabling users to initiate network monitoring, visualize the network topology, and receive real-time attack notifications, recorded in a Table, as reported in Figure 4.3. Furthermore it is possible to read the aims and functionalities of AGS in the Figure 4.4. Structured as a single-page application, the Frontend is anchored by index.html, which defines the HTML layout, including a navigation bar, a canvas element (topologyCanvas) for network visualization, and control elements such as "Start Monitoring" and "Stop Monitoring" buttons, alongside a monitoringStatus indicator. The visual styling, governed by dashboard.css, employs a dark theme with a gradient background and color-coded alerts (e.g., pink for normal states, red for attacks), ensuring clarity and aesthetic coherence. The dynamic behavior is orchestrated by app.js, a JavaScript file leveraging the HTML5 Canvas API to render the network topology and handle user interactions. Key functionalities include:

- **Network Visualization**: The drawTopology(devices) function renders a radial layout on the canvas, positioning the gateway (denoted by gateway.svg) at the center and connected devices (using normal-host.svg) around it via semi-transparent white lines.
- **Monitoring Control**: The startMonitoring() and stopMonitoring() functions issue HTTP POST requests to /api/start and /api/stop, respectively, toggling the monitoringStatus display between "Monitoring: Active" (green) and "Monitoring: Stopped" (pink).
- **Real-Time Updates**: A periodic polling mechanism (setInterval(updateUI, 1000)) fetches device data from /api/devices, refreshing the topology every second.

4.1 Behavioral Changes Upon Attack Detection

When an ARP spoofing attack is detected by the backend (ARPCache), the Frontend dynamically adapts to reflect the threat. The updateUI() function retrieves updated device states, identifying devices marked as attacked or is_attacker. The drawTopology() function then modifies the visualization:

- **Icon Changes**: Affected devices switch from normal-host.svg to attacked-host.svg (for victims) or attacker-host.svg (for attackers), visually distinguishing their roles.
- **Line Animation**: Connections involving attacked or attacking devices transition from steady white to flashing red lines, achieved through a setInterval toggle (flashState) every 500 milliseconds, maintaining the same 0.4 transparency for consistency.
- **Alert Notification**: The alert div updates from "Network Safe", represented in Figure 4.1, to "Attack Detected!" in red, visualised in Figure 4.2, with a blinking animation enhancing visibility.

These changes ensure immediate, intuitive feedback, enabling users to identify and respond to ARP spoofing incidents effectively. The Frontend's responsive design and real-time updates underscore its role as a critical interface between the system's detection capabilities and the end user.

Figure 4.1: AGS GUI: Network Safe

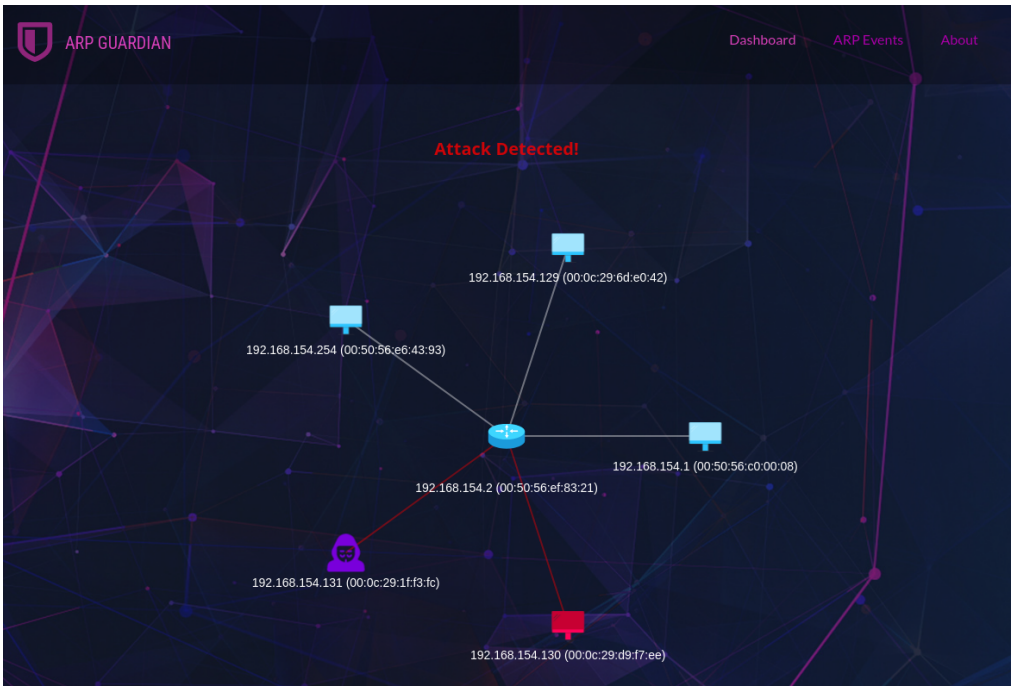

Figure 4.2: AGS GUI: Network under attack

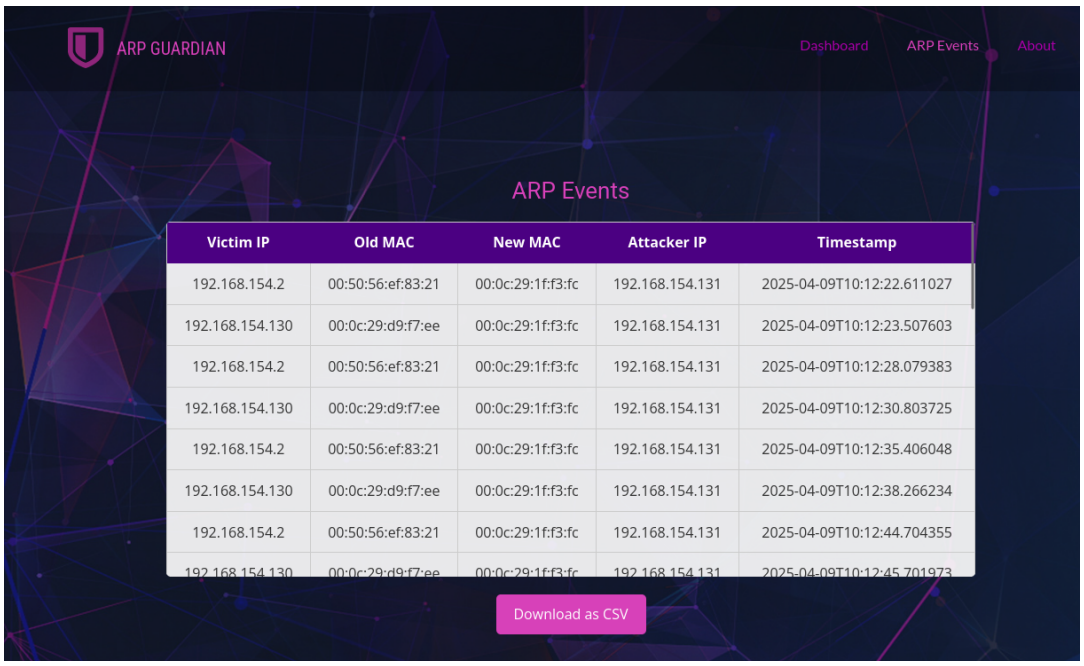

Figure 4.3: AGS GUI: ARP Events

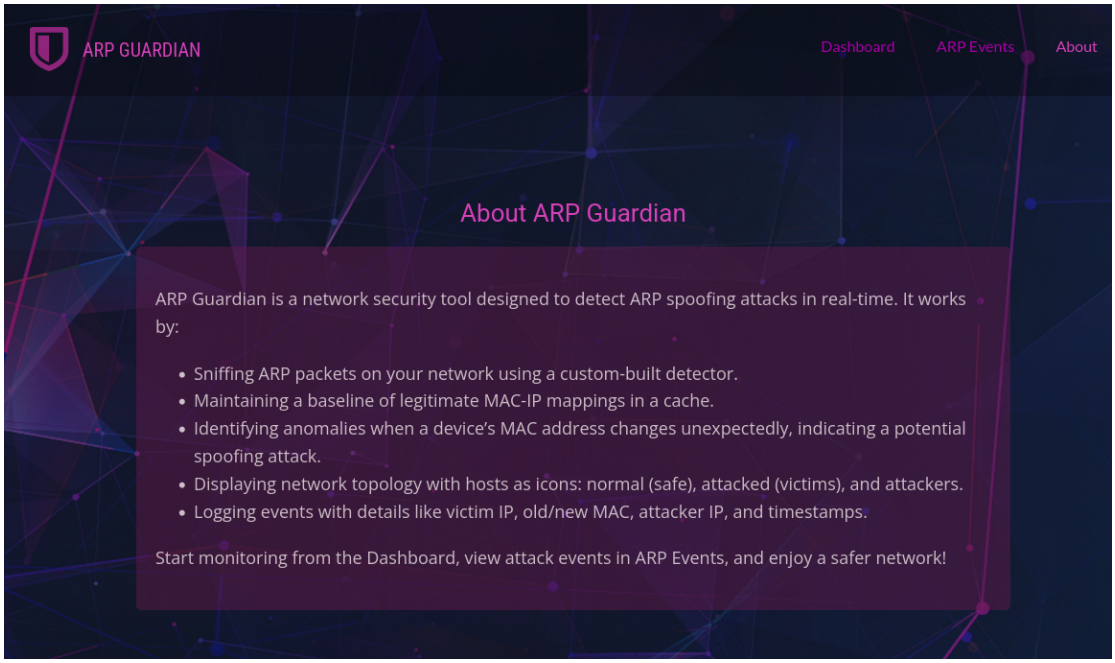

Figure 4.4: AGS GUI: About ARP Guardian

Chapter 5

Challenges and limitations

The detection logic of AGS, encapsulated within the *ARPCache.update()* method, relies on the continuous observation of ARP packets to compare their source IP and MAC addresses against a trusted baseline stored in *self.baseline_ cache*. In a virtualized environment such as VMware, this mechanism operated effectively because the virtual network switch emulated a shared medium, delivering all subnet traffic—including unicast ARP replies—to the Detector's interface. This ensured that attacks targeting any virtual machine within the subnet were visible and detectable. However, when deployed in a **real-world switched network**, the system's detection capabilities were significantly constrained, failing to identify ARP spoofing attacks directed at devices other than the Detector itself.

This limitation arises from the inherent behavior of Ethernet switches in physical networks. Modern switches maintain a MAC address table to forward unicast packets—such as the ARP replies employed in spoofing attacks—exclusively to the port associated with the destination MAC address.

Consequently, the Detector, situated on a distinct port, does not receive these packets unless it is the intended receiver. In testing, this restricted the system's scope to self-directed attacks, while attacks against other devices, went undetected despite the Detector's interface operating in promiscuous mode. This discrepancy highlights a fundamental challenge in ARP-based monitoring tools: their dependency on network-level visibility, which is readily available in virtual setups but obstructed in switched LANs.

5.1 Requirements for effective detection

For the detection logic of AGS, to function universally in a real-world environment—capable of identifying ARP spoofing against any device within the subnet the following conditions must be satisfied:

1. **Comprehensive Traffic Visibility**: The Detector's network interface must have access to all ARP traffic within the subnet, encompassing both broadcast requests and unicast replies directed to other devices. Overcoming the switch's port isolation is critical and can be achieved through:
    - **Traffic Mirroring**: Configuring a Switched Port Analyzer (SPAN) port or equivalent on a managed switch to duplicate all subnet packets to the Detector's port.
    - **Shared Medium**: eploying a hub or network tap to broadcast traffic to all connected ports, though this is less practical due to the obsolescence of hubs and the specialized nature of taps.
    - **Strategic Positioning**: Repositioning the Detector within the network topology to intercept all traffic, such as designating it as the default gateway or integrating it into the router/switch infrastructure.
2. **Promiscuous Mode Activation**: The Detector's interface (e.g. eth0) must operate in promiscuous mode to process all packets it receives, not solely those addressed to its own MAC address. While this was successfully implemented in the system, its effectiveness hinges on the prior condition of traffic visibility being met.
3. **Network Configuration Control**: Achieving comprehensive visibility requires either administrative access to the network infrastructure (e.g., a managed switch for SPAN configuration) or the ability to reconfigure device routing and hardware deployment. Consumer-grade routers typically lack advanced features like SPAN, necessitating alternative strategies or specialized equipment.

In the absence of these conditions, the Detector's interface is limited to capturing broadcast ARP requests and packets explicitly directed to its MAC address. Since ARP spoofing attacks predominantly utilize unicast replies to manipulate a victim's ARP cache, these packets bypass the Detector in a standard switched network unless explicitly redirected to its port. This behavior explains the system's success in VMware, where the virtual switch flooded all traffic to the Detector, contrasted with its failure in the physical environment without additional configuration. The reliance on specific network conditions thus underscores a practical constraint of ARP-based monitoring tools in modern switched LANs, distinguishing the ease of virtualized testing from the complexities of real-world deployment.

5.2 Solution: Detector as Default Gateway with Traffic Forwarding

To address this visibility challenge and enable universal detection, a possible solution could be implemented: configuring the Detector machine as the default gateway for all devices within the subnet. In this approach, the Detector supplants the router's role as the gateway, with devices such as the Attacker and Victim manually reconfigured to route their traffic through the Detector. The Detector inspects all outbound ARP packets for spoofing attempts before forwarding them to the actual router, effectively positioning itself as a central traffic hub.

By acting as the default gateway, the Detector receives all ARP requests and replies from subnet devices, including unsolicited replies from an attacker attempting to spoof the gateway's IP to target the Victim.

The *ARPCache.update()* method processes these packets, comparing them against the baseline to detect mismatches indicative of spoofing. Once identified, the system logs the event and updates the topology to reflect the attack, marking the Attacker and Victim accordingly.

The **Detector-as-gateway solution** resolves the visibility issue by ensuring all ARP traffic—broadcast requests and unicast replies—passes through the Detector's interface. Unlike the original real-world setup, where switch isolation limited detection to self-targeted attacks, this approach guarantees that **spoofed ARP replies targeting any device (e.g., Victim) are intercepted and analyzed.** By forwarding traffic post-inspection, it maintains network functionality while fulfilling the detection logic's need for comprehensive packet access, aligning with the system's design goals in both virtual and physical contexts.

Conclusion

This project represents a significant contribution to the field of network security, particularly in addressing the pervasive threat of ARP spoofing attacks. This work successfully demonstrates the design, implementation, and testing of a Network Intrusion Detection System (NIDS) capable of detecting ARP spoofing within a local area network (LAN) with high accuracy and efficiency. By leveraging the Model-View-Controller (MVC) architectural pattern, the ARP Guardian System (AGS) integrates a robust back-end, powered by Python and the Scapy library, with an intuitive front-end interface built using Flask, HTML, CSS, and JavaScript. This combination ensures both technical precision in packet analysis and user accessibility through real-time visualization of network states. The system operates by establishing a baseline of legitimate IP-to-MAC address mappings and continuously monitoring ARP traffic to identify anomalies indicative of spoofing attempts. Upon detection, AGS flags attackers and victims, logs events, and updates a graphical user interface (GUI) to provide immediate, actionable insights to network administrators. The testing phase, conducted in a controlled virtual environment using Kali Linux machines, validated the system's reliability in identifying spoofing incidents. AGS stands out for its modular design, which enhances maintainability and testability, and its use of open-source technologies, making it an accessible solution for educational and small-scale enterprise environments. However, the project acknowledges limitations, such as its passive, out-of-line nature as an NIDS, which restricts it to detection rather than active prevention.

The ARP Guardian System, as currently implemented, functions effectively as an Intrusion Detection System (IDS), excelling in the passive monitoring and reporting of ARP spoofing attacks.

However, in real world networks, traffic visibility is restricted compared to virtual environments like VMware. It can be achieved through methods like SPAN ports or configuring the detector as the default gateway. This last allows it to inspect all ARP traffic and to identify spoofing attempts while maintaining normal network operation.

A promising direction for future development involves evolving AGS from an IDS into an Intrusion Prevention System (IPS), enabling it to not only detect but also actively mitigate such threats in real time. This transition would require several

enhancements to the system's architecture and functionality.

Bibliography

[1] Mahendra Data. The defense against arp spoofing attack using semi-static arp cache table. In *2018 International Conference on Sustainable Information Engineering and Technology (SIET)*, pages 206–210, 2018.

[2] Akhil P and Bijoy Antony Jose. A profiling based approach to detect arp poisioning attacks. In *2021 International Conference on Green Energy, Computing and Sustainable Technology (GECOST)*, pages 1–5, 2021.