Issam Akhtar
30131310
ABDELRAHMAN ABBAS
30110374

CPSC 526
ASSIGNMENT 5

Implementation Overview

We implemented a firewall simulator that processes rules and packets according to the assignment specifications. The simulator reads rules and packets from specified files, applies the rules in order to each packet, and determines whether each packet should be accepted, dropped, denied, or handled by the default action.

IP Address Handling

We implemented `ip2int()` to convert IP addresses to 32-bit integers for easier comparison and manipulation:

```
def ip2int(ip_str) -> int:
    if ip_str == "*":
        return 0

    try:
        oct = ip_str.split('.')
        if len(oct) != 4:
            return None  # not enough octets in ip

        for o in oct:
            value = int(o)
            if value < 0 or value > 255:
                return None  # Invalid

        # convert to int32: a.b.c.d = (a << 24) | (b << 16) | (c << 8) | d
        return (int(oct[0]) << 24) | (int(oct[1]) << 16) | (int(oct[2]) <<
8) | int(oct[3])
    except ValueError:
        return None
```

This function converts IP addresses from string format to integer representation for easier comparison. It handles the wildcard "*" as a special case, returning 0. The function validates that the IP address has exactly four octets and that each octet is between 0 and 255. If validation fails, it returns None. For valid IP addresses, it uses bitwise operations to convert the four octets into a single 32-bit integer, which enables efficient range checking in the CIDR matching function.

CIDR Range Matching

We implemented `cidr_ip_range()` to determine if an IP address falls within a specified CIDR range:

```
def cidr_ip_range(ip_int, cidr_range) -> bool:
    if cidr_range == "*":
        return True

    try:
        ip_str, prefix_len_str = cidr_range.split('/')
        prefix_len = int(prefix_len_str)

        if prefix_len < 0 or prefix_len > 32:
            return None

        base_ip_int = ip2int(ip_str)

        if base_ip_int is None:
            return None

        # mask based on prefix length and check if in range
        return (ip_int & 0xFFFFFFFF << (32 - prefix_len)) == (base_ip_int &
0xFFFFFFFF << (32 - prefix_len))

    except ValueError:
        return None
```

This function determines whether an IP address represented as an integer falls within a specified CIDR range. When the range is "*", it returns True to match any IP address. For standard CIDR notation, it parses the network address and prefix length, validating that the prefix is between 0 and 32. It then converts the base IP to an integer and uses bitwise operations to compare the network portions of both IPs. The function applies a mask based on the prefix length to isolate the network portion of both addresses and checks if they match, effectively determining if the IP is within the specified subnet.

Port Handling

The `get_ports()` function parses port specifications from the rules:

```
def get_ports(ports_str):
    if ports_str == "*":
        return ["*"]

    try:
        ports = []
        for port_str in ports_str.split(','):
            if port_str:  # skip empty
                port = int(port_str)
                if port < 0 or port > 65535:
                    return None
                ports.append(port)
        return ports
    except ValueError:
        return None
```

This function processes the port specifications from firewall rules. It treats the wildcard "*" as a special case that matches any port. For specific port listings, it splits the comma-separated string and converts each value to an integer. The function validates that each port number is within the valid range of 0 to 65535. If any port number is invalid or cannot be parsed as an integer, the function returns None to indicate an error. Otherwise, it returns a list of valid port numbers that can be used for matching against packet port values.

Rule-Packet Comparison

The core matching logic is in `rule_packet_comp()`:

```
def rule_packet_comp(rule, packet, flag_value=None):
    r_dir, r_action, r_range, r_ports, *r_flags = rule
    p_dir, p_ip, p_port, p_flag = packet

    if r_dir != p_dir:
        return False

    packet_ip = ip2int(p_ip)
    result = cidr_ip_range(packet_ip, r_range)

    if result is None:
        return None

    if not result:
        return False

    rule_ports_parsed = get_ports(r_ports)

    if rule_ports_parsed is None:
        return None

    if ("*" not in rule_ports_parsed and int(p_port) not in
rule_ports_parsed):
        return False

    if r_flags and r_flags[0] == "established" and p_flag != "1":
        return False

    return True
```

This function performs the crucial task of determining whether a packet matches a firewall rule. It first unpacks both the rule and packet components, then performs a sequence of checks. Direction matching ensures that rules for incoming traffic only apply to incoming packets, and similarly for outgoing traffic. Next, it converts the packet's IP address to an integer and checks if it falls within the rule's CIDR range. The function also verifies that the packet's port matches one of the ports specified in the rule (or matches any port if the wildcard is used). Finally, it handles the "established" flag by checking if the rule requires an established connection and whether the packet is part of one. The function returns True only if all conditions are met, False if any check fails, and None if an error is encountered during validation.

Input Validation

We implemented thorough validation for both rules and packets:

For rules:

```
def get_rules(line, i, filename):
    fields = line.split()

    if len(fields) not in [4, 5]:
        raise Warning(f"{filename}:{i}: rule must have 4 or 5 fields")

    if fields[0] not in ["in", "out"]:
        raise Warning(
            f"{filename}:{i}: invalid direction '{fields[0]}', must be 'in'
or 'out'")

    if fields[1] not in ["accept", "drop", "deny"]:
        raise Warning(
            f"{filename}:{i}: invalid action '{fields[1]}', must be
'accept','drop', or 'deny'")
...
```

The `get_rules` function parses and validates firewall rules from input lines. It first splits the line into fields and checks that the rule has either 4 or 5 fields as specified in the assignment. It then validates that the direction is either "in" or "out" and that the action is one of the allowed values: "accept", "drop", or "deny". The function performs additional validation on the IP range, ensuring that CIDR notation is correct with a valid prefix between 0 and 32. It also validates port specifications and checks that the optional flag field, if present, is set to "established". If any validation fails, the function raises a Warning with a descriptive error message that includes the filename and line number for easier debugging.

For packets:

```
def get_packet(line, i, filename):
    fields = line.split()

    if len(fields) != 4:
        raise Warning(f"{filename}:{i}: packet needs 4 fields")

    if fields[0] not in ["in", "out"]:
        raise Warning(
            f"{filename}:{i}: invalid direction '{fields[0]}', must be 'in'
```

```
or 'out'")
...
```

The `get_packet` function handles the parsing and validation of packet definitions. It ensures that each packet has exactly 4 fields and that the direction is either "in" or "out". The function validates that the IP address is properly formatted and that the port number is an integer between 0 and 65535. It also checks that the flag value is either "0" or "1", representing a new or established connection respectively. Similar to the rule validation, this function raises a Warning with a descriptive error message that includes the filename and line number when it encounters any invalid data, making it easier to identify and fix issues in the input files.

Main Simulator Function

The `fwsim()` function orchestrates the entire simulation:

```
def fwsim(rules_fname: str, packets_fname: str) -> list[list[str]]:
    results = []
    rules = []
    line_num = []

    try:
        # Read and process rules...

        # Read and process packets...
        for i, line in enumerate(packet_lines, 1):
            # Process comments and empty lines...

            packet = get_packet(l, i, packets_fname)
            match = False

            for i, rule in enumerate(rules):
                result = rule_packet_comp(rule, packet)

                if result:
                    results.append(
                        (rule[1], str(line_num[i]), packet[0], packet[1],
packet[2], packet[3]))
                    match = True
                    break

            # No match then use default
            if not match:
```

```
                results.append(
                    ("default", "", packet[0], packet[1], packet[2],
packet[3]))

    except (IOError, FileNotFoundError) as e:
        raise Warning(f"Error opening file: {str(e)}")

    return results
```

The `fwsim` function is the main entry point for the firewall simulator. It takes filenames for rules and packets as inputs and returns a list of matching results. The function first reads and processes the rules from the rules file, storing them along with their line numbers for later reference. It handles comments by removing anything after a '#' character and skips empty lines. For each packet in the packets file, it attempts to find a matching rule by iterating through the rules in order. When a match is found, it adds a tuple to the results containing the rule's action, line number, and the packet's details. If no matching rule is found, it applies the default action. The function includes comprehensive error handling for file operations and other exceptions, raising informative Warning messages when problems occur.

Error Handling

Our implementation includes comprehensive error handling for various scenarios. We validate all input data rigorously, checking for valid IP addresses, port numbers, CIDR notations, directions, and actions. When an error is detected, we raise a Warning exception with a descriptive message that includes the filename and line number, making it easy to identify the source of the problem. We also handle file opening errors and user interruptions gracefully. All of this ensures that the simulator behaves predictably even when given problematic input.

Testing

We've tested our implementation with all the provided test cases and it correctly produces the expected outputs for each. Our simulator successfully handles the various rule types, packet scenarios, and edge cases described in the assignment. The results match the expected output files, confirming that our implementation meets all the requirements. We wrote a shell script that runs test cases. You provide the desired suffix for the test file and the script runs the firewall with that input.

```
[UC issam.akhtar@csx2 a5] ./test.sh
Enter file suffix number
0
Running 'python3 fw.py rules0.txt packets0.txt | diff -y results0.txt - '
accept(1)    in  136.159.5.22    22    1                       accept(1)    in  136.159.5.22    22    1
deny(3)      in  10.0.0.44       80    0                       deny(3)      in  10.0.0.44       80    0
default()    out 10.0.1.1        80    0                       default()    out 10.0.1.1        80    0
results match
[UC issam.akhtar@csx2 a5] ./test.sh
Enter file suffix number
1
Running 'python3 fw.py rules1.txt packets1.txt | diff -y results1.txt - '
accept(1)    in  136.159.5.5     22    0                       accept(1)    in  136.159.5.5     22    0
default()    in  136.159.5.5     23    0                       default()    in  136.159.5.5     23    0
accept(6)    in  136.159.5.6     22    1                       accept(6)    in  136.159.5.6     22    1
accept(2)    in  136.159.255.5   80    0                       accept(2)    in  136.159.255.5   80    0
accept(2)    in  136.159.0.5     8080  1                       accept(2)    in  136.159.0.5     8080  1
accept(2)    in  136.159.255.0   8080  0                       accept(2)    in  136.159.255.0   8080  0
default()    in  136.158.5.5     8080  0                       default()    in  136.158.5.5     8080  0
accept(3)    in  24.25.26.27     443   0                       accept(3)    in  24.25.26.27     443   0
default()    in  24.25.26.27     444   0                       default()    in  24.25.26.27     444   0
accept(6)    in  24.25.26.27     444   1                       accept(6)    in  24.25.26.27     444   1
deny(5)      in  24.25.26.27     21    1                       deny(5)      in  24.25.26.27     21    1
accept(4)    in  10.0.0.0        1000  0                       accept(4)    in  10.0.0.0        1000  0
accept(4)    in  10.0.0.1        1000  0                       accept(4)    in  10.0.0.1        1000  0
default()    in  10.0.0.2        1000  0                       default()    in  10.0.0.2        1000  0
accept(7)    out 137.255.0.255   33    0                       accept(7)    out 137.255.0.255   33    0
deny(8)      out 10.0.0.133      1     0                       deny(8)      out 10.0.0.133      1     0
deny(9)      out 5.5.5.5         22    0                       deny(9)      out 5.5.5.5         22    0
deny(9)      out 5.5.5.5         22    1                       deny(9)      out 5.5.5.5         22    1
accept(10)   out 5.5.5.5         23    1                       accept(10)   out 5.5.5.5         23    1
results match
[UC issam.akhtar@csx2 a5] ./test.sh
Enter file suffix number
2
Running 'python3 fw.py rules2.txt packets2.txt | diff -y results2.txt - '
deny(1)      in  1.2.3.4         55    0                       deny(1)      in  1.2.3.4         55    0
drop(6)      in  1.2.3.4         56    0                       drop(6)      in  1.2.3.4         56    0
default()    in  1.2.3.4         57    0                       default()    in  1.2.3.4         57    0
default()    out 200.200.200.127 2     1                       default()    out 200.200.200.127 2     1
accept(8)    out 200.200.200.128 2     1                       accept(8)    out 200.200.200.128 2     1
default()    out 200.200.200.128 2     0                       default()    out 200.200.200.128 2     0
drop(9)      out 127.20.3.5      10    0                       drop(9)      out 127.20.3.5      10    0
default()    out 128.20.3.5      10    0                       default()    out 128.20.3.5      10    0
results match
[UC issam.akhtar@csx2 a5]
```

Here is the test script output for each possible test case. It provides a side by side comparison of the fwsim output with the result.txt file. We end the test script with a message telling us if the results match or not.