Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

LANCASTER UNIVERSITY

Master Thesis

Lightweight Detection of Malicious Traffic in Wireless Mesh Networks

Fabian Hugelshofer

Department of Information Technology and Electrical Engineering

Swiss Federal Institute of Technology Zurich

30 September 2008

ETH Zurich
Communication Systems Group
Prof. Bernhard Plattner
Mario Strasser

Lancaster University
Computing Department
Prof. David Hutchison
Dr. Nicholas Race
Dr. Paul Smith

Acknowledgments

I wish to thank all those who supported me during this thesis.

First of all, I want to thank Prof. Bernhard Plattner who enabled me to write my thesis abroad and Prof. David Hutchison who accepted me at Lancaster University. Without their will to support student exchanges, I would not have had the opportunity to combine writing my thesis with making new social and cultural experiences.

Special thanks goes to Paul Smith. He was always a good help with his critical and constructive statements, guiding the right way and with proofreading. He gave me a lot of freedom and made me feel like a well accepted member of staff. I also want to thank Mario Strasser for his feedback on the final presentation and the documentation.

Pablo Neira Ayuso and Patrick McHardy from the Netfilter mailing list were always open for my input and provided good discussions. I want to thank them for their support.

And finally, I want to thank my friends in Lancaster for contributing a lot to the good time I had there.

Abstract

Wireless mesh networks are becoming increasingly popular as a way of providing Internet access in a cost efficient way. Network traffic generated by malware infected computers can cause significant congestion on these networks, reducing the quality of service both on the wireless mesh network and on Internet uplinks. In this work we present OpenLIDS, a novel anomaly-based lightweight intrusion detection system as a solution to this problem. OpenLIDS detects excessive malicious traffic and automatically applies remedies to reduce the problem of congestion.

In order to be cost-efficient, it is proposed that intrusion detection should be performed on mesh boxes. These devices are typically cheap and have constrained hardware resources. Common intrusion detection systems – Snort and Bro – are shown to perform poorly on a typical mesh device under high traffic volumes. We therefore propose a dynamic load dependent intrusion detection architecture which extends its detection algorithms to more expensive metrics, if sufficient resources are available.

OpenLIDS is lightweight enough to analyse high traffic volumes on resource constrained hardware. It is able to detect a wide range of malicious traffic, has a very low false positive rate and automatically blocks detected malicious flows in a fine-grained way. To the best of our knowledge, no prior systems exist that have these characteristics, and it therefore builds an important missing block in the proposed intrusion detection architecture.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Wireless Mesh Networks

This section gives an introduction to wireless mesh networks and presents two particular examples.

1.1.1 Description

Wireless Mesh Networks (WMNs) are self managing networks in which radio nodes participate in transmitting traffic from others to reach a destination, which they could not reach themselves. This allows to extend the wireless service area. The network is dynamic in a sense that nodes can leave or enter the network at any time. Nodes can be switched on and off or can get out of the radio range of another node because of mobility or a weak signal. To cope with these dynamics ad-hoc routing protocols are used. They provide the information to which neighbouring node a packet has to be sent, such that it reaches its final destination after multiple hops. They respect the network dynamics and do not rely on prior information.

Wireless mesh networks are becoming increasingly popular as a way of providing Internet access in a cost efficient way [73, 55, 3, 77, 46]. A node in this

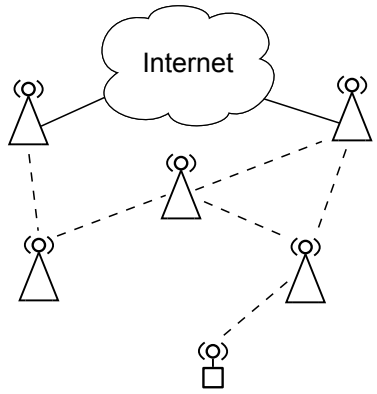


Figure 1.1: Wireless mesh network

case corresponds to a wireless router which participates in the mesh network. Such a router is also referred to as a mesh box. Typically the IEEE 802.11 wireless standard is used to connect the mesh boxes in ad-hoc mode. A client can get access to the network by using a mesh box as a wireless *Access Point (AP)* or by directly connecting to a mesh box with a network cable. The mesh network forwards packets from clients to uplink points from where they get to the Internet. The uplinks can be provided e.g. by people sharing their Internet connection or by an *Internet Service Provider (ISP)*. Figure 1.1 schematically shows such a WMN.

The advantage of a WMN is that it can easily be deployed without large infrastructure costs. No cabling is needed between the mesh boxes and they do not need to have a lot of hardware resources and can therefore be cheap. The WMN projects presented below are all operated in a community manner. This means that people can buy mesh boxes themselves and can become part of the network, even if they are technically unaware. As a consequence there does not have to be an authoritative operator of the network. Its operation is decentralised and unmanaged.

1.1.2 Example: Wray Community Mesh Network

The village of Wray is situated in the North West of England. It is located in a rural area and has less than 500 inhabitants. Because of its remote location and its small size Internet service providers had no economic interest in building up the infrastructure necessary to provide broadband Internet access. Lancaster University made the decision to establish a wireless mesh network in Wray in order to provide broadband Internet access to the village on one hand and to have a real world testbed on the other [25, 94]. Access to the network is free for users. There is a local support group consisting of village members which is backed by researchers from Lancaster University.

The Wray network uses a hardware and software solution from Locustworld [38]. 12 Mesh boxes are located at user's homes and at key locations in the village. For intra-mesh communication IEEE 802.11b (11 Mbps) is used and an IEEE 802.16 (WiMAX) link provides upstream connectivity to the Internet with 10 Mbps. The boxes are equipped with 120 MB RAM, 32 MB Flash memory and a VIA Eden processor running at 533 MHz. The hardware is relatively expensive.

Each mesh box provides a virtual access point and a wired port where clients can connect to. The virtual AP assigns an IP address from its dedicated 192.168.X.0/24 subnetwork whereas for the LAN port addresses from 192.168.1.0/24 are used. After Web-based authentication, the client gets access to the network. It has to be noted that such an authentication system is generally rather unlikely in a WMN. The mesh boxes apply *Network Address Translation (NAT)* to IP addresses in the reserved address block 1.0.0.0/8 which are used for intra-mesh communication. Communication between mesh boxes is secured by pair-wise point-to-point VPN connections. AODV [61] is used as the routing protocol. Three boxes are connected to the uplink router and apply NAT to translate private mesh addresses into public ones. For fairness reasons, traffic to a specific mesh box is capped at 3 Mbps and to a particular user at 2 Mbps.

1.1.3 Example: Openwireless St. Gallen

Openwireless [56] is a non-profit community, operating wireless mesh networks in multiple cities in Switzerland. Their motivation is to provide free Internet access for everyone. The biggest Openwireless network is in the city of St. Gallen in the eastern part of Switzerland. It consists of 243 online nodes at the time of writing. Users can buy their own hardware and install software provided by the community to become part of the mesh. Dedicated shops sell pre-installed boxes. Core uplink points exist at well exposed positions in three different locations. They share a 10 Mbps connection to the Internet which is funded by sponsors. The community notes that this connection might be extended to 100 Mbps in the future. There exist 14 further uplink points due to users sharing their Internet connections at home.

Common wireless routers from Linksys (WRT54GL) and Buffalo (WHR-G54S) serve as mesh boxes. Both devices have 16 MB RAM, 4 MB Flash memory and a Broadcom 5352 Processor running at 200 MHz. Compared to the Wray network this hardware is more typical for a community-operated WMN. The software installed on the boxes is the Freifunk firmware [18] which is based on OpenWRT [57]. OpenWRT is a stripped down Linux distribution for network devices.

The mesh boxes use the IEEE 802.11g standard (54 Mbps) for wireless communication and OLSR [14] as the routing protocol. Users can connect to their mesh boxes' LAN port or can join the wireless network in ad-hoc mode. No authentication is necessary but a Web-based agreement has to be accepted. Clients using the wired connection get an IP address from private range 192.168.1.0/24. Each mesh box gets a subnet with a 28 bit netmask assigned from 10.248.0.0/16. One address is used for intra-mesh communication and 13 addresses can be assigned to wireless clients. Both, wired and wireless client addresses, get masqueraded by NAT. All traffic in the network is unencrypted.

1.2 Problem Statement

This section explains contention and congestion problems in wireless and wireless mesh networks in Section 1.2.1 and Section 1.2.2 gives the motivation for detecting malicious traffic in wireless mesh networks.

1.2.1 Wireless Multihop Networks

The wireless channel is a shared medium. If two stations try to send at the same time, collisions can occur. In IEEE 802.11 a sender has to sense the channel to be free for a certain period of time, known as the *Distributed Inter-Frame Spacing (DIFS)*, before starting to transmit. If the channel is busy, it has to wait for a specific period of time, known as the *Backoff Interval (BO)*, before sensing the channel again. This channel access scheme is called *Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)*. A heavily utilised channel leads to link layer contention. This means that senders have to wait for the channel to become free to get their turn to send. Collisions can still occur in random cases or in the presence of hidden terminals [83]. The optional *Request-to-Send (RTS)* and *Clear-to-Send (CTS)* message mechanism addresses this problem, but it is seldomly used [26].

Wireless card manufacturers often implement an *Auto Rate Fallback (ARF)* mechanism which reduces the transmission rate, if errors occur [30]. With lower sending rates the distance between symbols in the signal space is bigger and the probability of a symbol error therefore smaller. A lower sending rate increases the stability of the communication in case of lossy channels. Under congestion, packets get lost because of collisions or because they cannot be sent at all. Lower sending rates do not help in this case, but drive the channel utilisation even higher. The problem is that ARF does not distinguish losses due to congestion from losses due to bad channel quality. It has been shown that the auto rate fallback drastically reduces the overall throughput in case of high congestion [23, 26]. High congestion was defined as a channel utilisation over 80 %.

Wireless networks have the property that a node can only send or receive at anytime (half-duplex). A mesh node forwarding packets has first to receive a packet and then send it on, whereas no further reception is possible during this time. This results in a throughput which is at least 50 % smaller than the maximum possible throughput of a single-hop transmission. As packets get sent from hop to hop, the channel gets occupied multiple times for the same packet. The result is higher contention and a higher probability for collisions. As long as the radio ranges overlap, the maximum possible throughput decreases with every hop. Koutsonikolas et al. [33] show that the maximum throughput for TCP in an IEEE 802.11b network decreases from 5.5 Mbps for a one-hop flow to 0.7 Mbps for a three-hop flow.

1.2.2 Malicious Traffic in Wireless Mesh Networks

Stone-Gross et al. [80] analysed the effect of malware on wireless networks. They show that single attackers are able to adversely affect other client's performance in the network. Single worm-infected PCs caused worm propagation traffic, which drove channel utilisation up to nearly 100 %. At peak times almost 50 % of all transmissions were retransmissions. The average of TCP *Round-Trip Times (RTT)* increased by more than 50 % and the mean by more than 100 %. Clients have been observed reducing their transmission rates due to increased packet loss which made the problem of congestion more severe. The heavy channel utilisation additionally obstructed the access point's probe responses to clients, which were probing for access points. The authors conclude that providing quality of service in a wireless network is virtually impossible without developing mechanisms to reduce unwanted link layer contention.

The work presented by Stone-Gross et al. [80] analyses a scenario of access points with wired backlinks. No similar work exists on wireless mesh networks. Previously we have seen, that congestion in multi-hop networks causes more contention and collisions due to multihop forwarding. It has to be expected that the effects of malware in a mesh network have a bigger impact on the network quality than in a single hop case. As the traffic propagates through a wider area, its impact on the network quality is not limited to a local scope. This implies that it is particularly important to reduce the amount of malicious traffic in a wireless mesh network to maintain the quality of service in the presence of attacks.

In addition to the impact on the wireless medium, it is further important to prevent malicious traffic from using uplink bandwith. Broadband Internet connections are typically asymmetric: they have a much lower upstream than

downstream bandwith. The available uplink capacity should not be wasted for malicious traffic. Another aspect of users sharing their Internet connection is, that law enforcement could identify them as the origin of an attack, even if the attack was actually carried out by a client in the wireless mesh network.

The problem in a typical WMN is that there is no central authority deploying *Intrusion Detection Systems (IDS)* to identify infected or misbehaving clients, as this might be the case in enterprise networks. In an unmanaged and community-organised mesh network it is unlikely that anyone is willed to pay for additional hardware to perform intrusion detection and prevention. Host-based solutions are not feasible, as the wireless mesh community has no influence on client PCs. In an open network everybody can connect and there is no guarantee that clients are running any security software. The most attractive way to perform intrusion detection in a community-based wireless mesh network is therefore to use the existing infrastructure: the mesh boxes. The community does not necessarily operate them directly, but has a big influence by providing their software.

1.3 Goal and Requirements

1.3.1 Goal

The main goal of this work is to **protect wireless mesh networks and their Internet uplinks against congestion caused by malicious traffic**. Congestion is mainly caused by excessive traffic. Malicious traffic with low rates does not cause much harm to the network and its detection is therefore not of high importance.

Mesh boxes provide the most convenient way to perform intrusion detection in a community-based WMN. As they have constrained hardware resources, it is important to understand their capabilities and limitations. An important aspect of this work is to analyse these resource constraints. This will build up knowledge for future projects in wireless mesh networks.

To detect malicious traffic, a set of lightweight detection metrics has to be identified. It has to allow reliable detection of malicious traffic with resource constrained hardware. The detection metrics have to be implemented in an intrusion detection system, which will run on wireless mesh boxes. It has to take automated remediation to respond to detected events. More detailed requirements to this system are specified in Section 1.3.2.

1.3.2 Requirements

The following list specifies the requirements for a malicious traffic detection system in a WMN:

R1 Malicious Traffic Recognition: Excessive malicious traffic harming the network by causing congestion has to be detected. Malicious traffic with low rates needs not necessarily to be detected, as it does not contribute much to congestion. However, detection is desired as it makes the Internet safer and reduces legal risks for users sharing their Internet connection.

R2 Automated Response: As no operator is examining intrusion alerts to decide about mitigative actions, the IDS has to be built to automatically

respond to malicious traffic. The response has to abolish congestion caused by malicious traffic within a reasonable time.

R3 Very Low False Positive Rate: Identifying benign traffic as malicious in combination with automated response is dangerous, as it can lead to denial-of-service conditions or reduced quality of service. False positives have therefore to be avoided.

R4 Fine-grained Identification: Remediation should only affect malicious traffic. Benign traffic originating from the same source should not be affected. This allows a user of a malware infected machine to still browse the Web, even if his computer is the origin of malicious traffic. More important, the user should still be able to download software updates and update his Anti-Virus signatures. To achieve this, malicious traffic has to be identified as fine-grained as possible.

R5 No Interference with Router Operation: Operational tasks of the router shall not be hindered by the IDS. This includes packet forwarding, calculating routing tables, gathering statistics and others. The IDS has to be resource efficient regarding CPU and memory consumption.

R6 Security: The IDS shall not give additional attack surface. It has to be designed in a secure way to prevent being exploited e.g. to get access to the router or to prevent it from working correctly. Further, it shall not be possible for an attacker to cause denial of service conditions for other users in the network by tricking the IDS.

R7 Maximum Traffic Load: The system has to be designed to be able to cope with maximum traffic loads. A high traffic load shall not increase the probability of false negatives.

R8 Evasion Robustness: It shall not be possible to evade the IDS e.g. by fragmentation or other obfuscation techniques.

1.4 Work Organisation

Chapter 2 starts with presenting background information. It discusses related work, intrusion detection systems and different types of malicious traffic, as well as its characteristics.

In Chapter 3 a preliminary performance analysis of a common wireless router, as it could be used in a wireless mesh network, is performed. It analyses the router's capabilities and limitations with respect to forwarding traffic, capturing packets and running existing intrusion detection systems. This analysis demonstrates the need for a lightweight intrusion detection system and draws important conclusions regarding the design of such a system.

Anomaly-based detection metrics for a novel IDS are described in Chapter 4 and the implementation of this system is presented in Chapter 5. Chapter 6 then evaluates the implementation and verifies if the requirements are met.

Chapter 7 summarises the achievements and gives an outlook on future work.

Chapter 2

Background

This chapter provides background information related to this work. Targeted attacks against wireless mesh networks are presented in Section 2.1. A classification of intrusion detection systems, related work to detect malicious traffic and two popular intrusion detection systems are presented in Section 2.2. Section 2.3 discusses different types of malicious traffic, its characteristics and similarities to benign traffic.

2.1 Attacks on Wireless Mesh Networks

Targeted attacks against wireless mesh networks can occur on various layers. They attack the wireless medium or the routing infrastructure and both their origin as well as their impact are local. This corresponds to the close left lower space in the attack space diagram shown in Figure 2.1.

[95, 19] study the feasibility and effectiveness of launching jamming attacks against wireless networks, analyse different jamming detection metrics [95] and propose fast channel hopping to prevent such kind of attacks [19]. Bellardo and Savage [9] present denial-of-service attacks on the 802.11 link layer and propose countermeasures against these attacks. Radosavac et al. [67] focus on denial-of-service attacks which propagate from the MAC layer to the routing layer. They propose a scheme for attack detection based on modelling of MAC protocols using extended finite state machines.

Routing is an important aspect of a wireless mesh network's security. An attacker could try to disrupt routing or could try to attract more traffic to be routed through herself. This would allow her to read traffic from others or to drop packets to cause denial-of-service conditions. Karlof and Wagner [31] describe different attacks on routing protocols in wireless ad-hoc networks and identify countermeasures. The *Jellyfish* attack has been analysed in detail by [1]. In this attack a node participates in the routing process, but periodically drops, delays or reorders a small amount of TCP packets which triggers the congestion avoidance algorithm. This attack can have a devastating effect on throughput. To address this issue, secure routing protocols have been implemented [59, 10]. They rely on monitoring the traffic from a node's neighbours and are able to exclude misbehaving nodes from the routing process. Observing neighbours might not be possible in all cases or could be tricked [1]. The Jellyfish attack

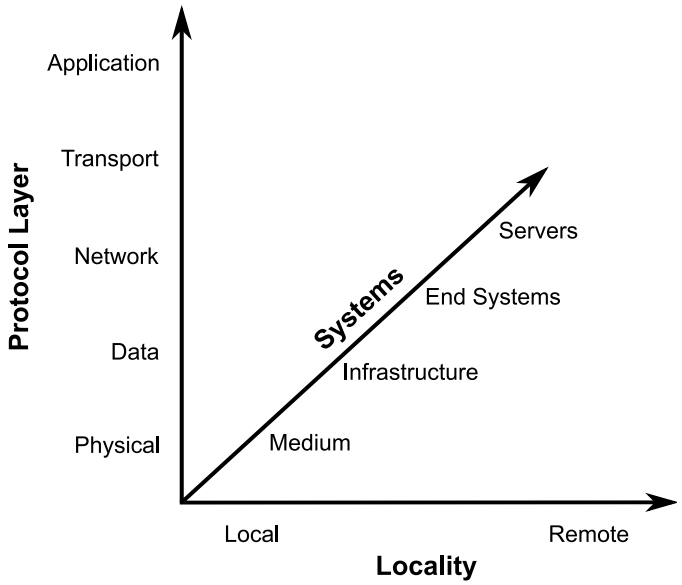


Figure 2.1: Attack Space

has therefore still to be considered as an unresolved threat.

Chen et al. [13] give an overview about intrusion detection systems in wireless mesh networks. They conclude that attacks on the wireless mesh infrastructure are conjectured and theoretical today because of their low deployment. The effect of malicious traffic in a wireless mesh network, which is addressed in this work, is already a problem today. It is different from the research presented in this section because the malicious traffic does not target the mesh network itself. It attacks remote, and potentially local, end systems and servers and is targeted at the network up to the application layer. The local impact on the physical up to the transport layer is rather a side effect than intended.

2.2 Intrusion Detection Systems

Deploying firewalls is not enough to protect a network against attacks. Some services must be reachable, and benign traffic has to be allowed to traverse the network. In an open wireless mesh network without any filters at all, even all traffic is generally allowed. Intrusion detection systems help to detect malicious traffic or the compromise of systems, such that it is possible to react to security incidents in a proper way and in a reasonable time. Section 2.2.1 classifies intrusion detection systems, Section 2.2.2 presents previous work on malicious traffic detection and Section 2.2.3 gives an overview about Snort and Bro.

2.2.1 Classification

Intrusion Detection Systems (IDS) in general are employed to detect attacks on confidentiality, integrity and availability of assets. In computer science there are two major types of IDS:

Host-based IDS (HIDS) monitor the state of a computer system and its transitions to detect attacks or even a compromise. This can include monitoring filesystems, operating system structures, behaviour of processes or network traffic entering or leaving the system. A HIDS has a detailed view about what is going on at a computer and can be used to detect malicious activity which might not be detectable externally. In a WMN there is no guarantee that client computers run a HIDS, which prevents them from emitting malicious traffic. Host-based approaches are therefore not a reliable solution to protect the network against malicious traffic.

Network-based IDS (NIDS) are strategically placed in a network to monitor traffic for signs of malicious activity. Typically, dedicated hardware is used to analyse traffic at key locations such as network borders. Because of the distributed manner of the mesh network, intrusion detection must be distributed, as well. There are e.g. no central Internet uplink points where a single detection system could be placed. To analyse all traffic in the network, multiple sensor nodes are required. In the case of a community-based WMN it is unlikely that somebody is willing to pay for dedicated hardware to perform intrusion detection. In this work, the existing wireless routers are used to perform network-based intrusion detection.

Network-based intrusion detection systems can perform traffic analysis on different levels:

Pattern Matching inspects packet contents for a match against a pattern from signatures of known attacks. Attack variations or new attacks require new signatures, which potentially leads to large signature databases. New attacks and permutation of old attacks cannot be detected this way. Content analysis of packets in general requires measures like IP defragmentation and TCP stream reassembly to prevent evasion.

Protocol Analysis can be used to check if communication protocol specifications are strictly obeyed. For clearly specified protocols a packet stream can be decoded and checked for deviations from the specification. In case of violations (e.g. overlong field lengths or incorrect state transitions) an alert is triggered. Protocol analysis does not help for obscure protocols or against attacks which are possible within the specification of the protocol for example.

Anomaly Detecion compares characteristics of network traffic to a baseline of normal behaviour. Deviations to the baseline are flagged as anomalies. The network observation can be based on abstract metrics. This can make it difficult to identify the exact cause of an anomaly. If e.g. the number of flows[1] rapidly increases, it is not clear whether this is because of malicious activity or just because of an unexpected change in user behaviour. Further analysis is required to understand the cause of the anomaly. Abstract statistics to be gathered can e.g. contain the number of packets or bytes per time or flow, the number of concurrent flows or the number of successful or failed connection attempts. Collecting these abstract statistics

[1] a flow is a unidirectional sequence of packets with identical layer-3 and layer-4 protocols and source and destination addresses

is resource efficient and is even possible, if application layer encryption is used. A problem of anomaly detection is that not every anomaly is caused by malicious traffic. Benign activities could cause an alert as well. Thresholds have to be carefully adjusted to avoid false positives. On the other side it is not possible to detect attacks, which do not show highly anomalous behaviour. This would constitute a false negative.

Once an attack is detected, an IDS triggers an alert by writing to a logfile or monitoring console or sending a warning to an administrator or a central correlation system. Additionally, it might capture a trace of the attack for further analysis. The attack itself is not prevented and another entity has to investigate further and potentionally take concrete actions. In an unmanaged wireless mesh network no such entity exists.

Intrusion Prevention Systems (IPS) automatically respond to detected attacks. One possible reaction is to block certain traffic completely. It is obvious that such a system has to be designed carefully. No benign traffic should be blocked and it should not be possible to misuse the IPS to deny service to legitimate users. A less strict mitigation strategy is to throttle certain traffic to an acceptable rate. Malicious traffic in this case would still be allowed to pass the network, but its harm to the network would be reduced. Throttling also reduces the impact of false positives as benign traffic which is wrongly flagged as malicious, is slowed down instead of being completely blocked. Which response to take has to be decided on the type of an event, the security requirements and the confidence in a malicious cause of that event.

2.2.2 Malicious Traffic Detection

The problem of detecting malicious traffic has already been addressed by previous work. None of it has analysed the feasibility of detecting malicious traffic in a wireless mesh network by the use of the existing resource constrained hardware.

In 1990 Heberlein et al. [22] implemented a connection counter to detect worms. It simply counts the number of connection attempts a client is performing. A machine infected with a network-spreading worm will have a high number of connection attempts as the worm tries to find vulnerable targets to propagate. This metric will also flag benign machines if they are very active.

Many connection attempts of a worm scanning for targets will not be successful because the target does not exist or does not run a particular service. For TCP, a connection attempt is made by sending a SYN packet to the target. The connection attempt is unsuccessful, if either no reply comes in at all, the target reports a closed port with a RST packet or an ICMP destination unreachable message is returned. Paxson [60] counts the number of unsuccessful connection attempts to identify infected machines.

A mixed approach presented in [29] is called *credit-based connection rate limiting*. With this approach a host gets credit for successful connection attempts, while its credit gets reduced for failed connection attempts. Connection attempts from a client with a low credit are rate limited.

Wagner and Plattner [87] use the entropy of IP addresses to detect worms. Entropy is a measure for randomness or disorder. Destination IP addresses of

packets sent by a worm-infected computer are distinct and have therefore a high entropy.

To detect network anomalies such as worm outbreaks the number of packets or flows can be used [11]. These metrics, like the connection counter, can result in false alerts and provide only coarse-grained information. Alerts have to be analysed by a network operator and are therefore not suitable in a WMN. But they have the nice property that they are also effective with traffic sampling, where only a fraction of packets are captured [11]. Traffic sampling might make sense in a wireless mesh network context.

Göldi and Hiestand developed an algorithm to identify worm-infected computers in an office environment to prevent the VPN uplink of branch offices from congestion resulting from malicious traffic [20, 88]. To reduce false alerts, they use a multi stage process to identify infected machines. First the number of failed connection attempts is counted for each host in the network. If it is higher than a fixed threshold, this host is monitored in more detail. Depending on how many different targets it tries to connect to and on which port, the host is classified as infected or the closer monitoring is terminated. Besides the detection algorithm, a good overview about various worms and their scanning and propagation methods is given in their work.

Their approach is interesting and is investigated further in this thesis. But there are also some limitations. First, there was a restrictive firewall in place during their tests. This increases the number of failed connection attempts compared to a setting without firewall, as this is the case in a WMN. The second problem is performance. The detection system was running on a comparably powerful computer (Intel Pentium 4, 2.4 GHz) and had only to analyse low traffic rates. In case of a high number of infected machines the CPU usage rose to 100 %. A more light-weight solution is necessary to be deployed in a WMN.

Malicious software can spread by email or can be used to send SPAM. If it uses a built-in SMTP engine to send the emails directly to its targets, it has to query the IP addresses of the mail servers accepting the mail for a particular domain. For this, the *Domain Name System (DNS)* is queried for the *Mail Exchange (MX)* record of that domain. Musashi et al. [49] use this to identify mass mailing worms by monitoring DNS MX queries.

Unused IP address, so called darknets or greynets, can be used to detect network scanning as no legitimate packets will be sent to these addresses [21]. This approach is only able to detect scanning of local address blocks, which does not cover non-local egress scanning traffic. This is an important class of malicious traffic.

The detection metrics presented so far are all anomaly based. Existing intrusion detection systems such as Snort [68] or Bro [60] inspect the content of packets and match it against signatures of known attacks. These systems will be presented in Section 2.2.3 and their performance will be analysed in Section 3.4.

Another approach based on deep content inspection is presented by [89]. They identify identical payload of epidemic worm propagation traffic based on histograms of the appearance frequencies of bytes. Their technique allows to detect high numbers of similar flows.

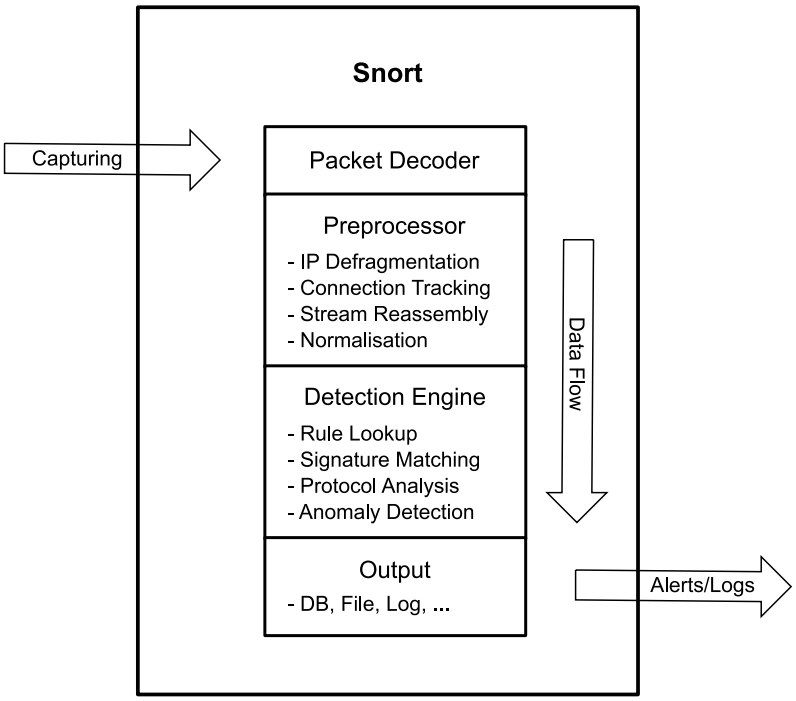


Figure 2.2: Overview Snort

2.2.3 Existing Intrusion Detection Systems

Snort and Bro are popular open source intrusion detection systems. Their performance will be analysed in Section 3.4. It will be shown that they have performance problems and are not a perfect solution for wireless mesh networks. This section presents how they work to later be able to understand the results of the performance measurements.

Snort

Snort [68] is a widely used network intrusion detection system. It is referred to as a lightweight intrusion detection system. Snort provides mainly signature, but also protocol and anomaly based inspection methods. Figure 2.2 gives an overview about Snort's operation.

Snort uses *libpcap* [36] to capture full packets from the network, typically via a network tap or a mirror port on a switch. The packet decoder then parses the packets from the link layer up to the application layer and fills the protocol information into data structures.

The preprocessor plug-ins are used to examine, manipulate and normalise the arriving packets. At this stage, IP defragmentation, connection tracking and stream reassembly are performed. Defragmentation, stream reassembly and data normalisation are important to prevent detection evasion. The computational and memory overhead for preprocessing depends on the particular plug-ins. They can be stateful.

After packets have been preprocessed they get forwarded to the detection

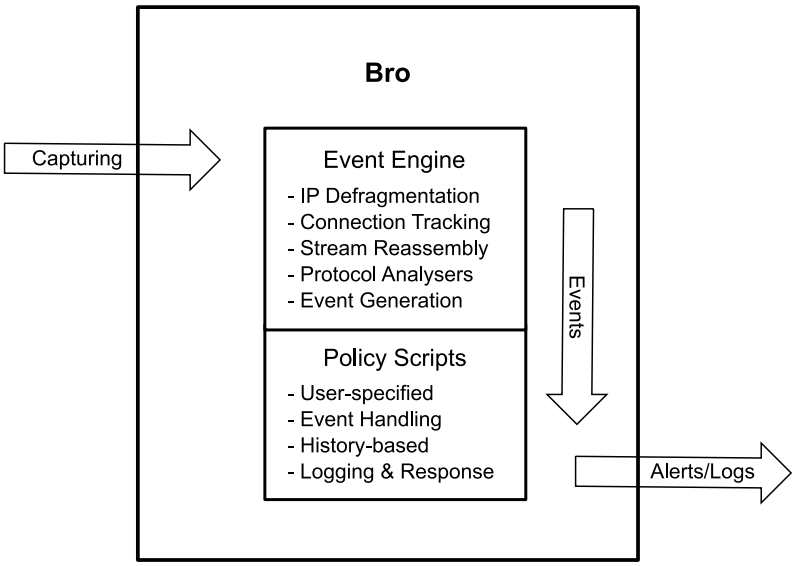


Figure 2.3: Overview Bro

engine. Based on layer-3 and layer-4 addresses, a rule database is searched for rules which have to be checked. Rules can easily be specified by the user. Commercial rule definitions as well as community rules are available. Snort organises the rules in a tree: similar rules are stored in the same branch and the search for rules only evaluates the branch, if it is relevant for a particular packet. If active rules for a packet are found, their referenced detection plug-in is called. Typically the content of the packet is tested for matching patterns. Pattern matching suffers from continuously growing signature databases and requires a high number of active signatures to achieve a good coverage. Protocol analysis and anomaly detection are possible with Snort as well.

On detected violations Snort can generate log messages, execute commands, log to databases or save a packet trace for forensic analysis.

Bro

The network intrusion detection system Bro [60] was built as a research platform for intrusion detection and traffic analysis. It is mainly anomaly based, but also supports pattern matching and protocol analysis. Figure 2.3 gives an overview about Bro's operation.

Similar to Snort, Bro uses libpcap to capture full packets. To prevent detection evasion it defragments fragmented packets, validates network- and transport-layer checksums and reassembles TCP streams.

TCP packets are stored in memory until they are acknowledged by the receiving host. An attacker could first send a packet with benign content and a low time to live, such that the destination does not receive the packet but the IDS does. If the IDS already continues reassembling the TCP stream, it might discard a second packet with equal sequence numbers but malicious content and now a high time to live. To defeat this kind of evasion, Bro stores packets in memory until they get acknowledged by the destination.

The core of Bro is the event engine. It performs policy-neutral analysis of network traffic at different semantic levels and generates events, which can be analysed by user-specified policy scripts. Events are e.g. generated for attempted, established, terminated or rejected connections, HTTP requests or replies, or failed or successful FTP authentication attempts. To generate these events Bro tracks connections and executes application layer protocol analysers, which can be stateful as well.

With Bro's own scripting language the generated events can be processed arbitrarily. It is possible to perform pattern matching with regular expressions, store information in hash tables or increment counters. The scripts can define highly stateful behaviour. This gives a lot of freedom to perform anomaly based intrusion detection. It is possible to generate log messages and execute programs. The latter can be used to send email notifications, start capturing network traces or introduce active response.

2.3 Malicious Traffic

This section discusses the origin and different types of malicious traffic, as well as its properties. The focus is on abstract characteristics which could be used for anomaly-based detection. Section 2.3.7 shows benign traffic which can have similar characteristics what could lead to false positives.

2.3.1 Origin of Malicious Traffic

In this work malicious traffic typically means traffic being generated by malware on an infected computer in the the mesh network. Malware is defined as software which has a purpose the user, on whose computer it is running on, would not agree with. An infection can e.g. occur by:

- Exploitation of network-exposed vulnerable services (e.g. ISS or Samba),
- Propagation at the application layer (e.g. by email or instant messaging),
- Drive-by-downloads while visiting a compromised Website,
- Hidden malware in allegedly benign software (Trojans) or
- Virus infected executables, documents or removeable medias.

In our scenario it is unlikely that malware is aware of the wireless mesh network or a specific intrusion detection system in place. General IDS evasion techniques might however still be applied. In most cases, the harm which malicious traffic causes to the network is not intended by the malware's creator, it is a side effect.

Malicious traffic can, besides malware infections, be generated by a user, who intentionally uses the network to launch attacks against targets in the Internet or in the network itself. Such an attacker can have detailed knowledge about the network and the IDS.

2.3.2 Worm Propagation

A computer worm is a self-contained and self-propagating program. It can propagate with or without human interaction. When propagating or preparing to, excessive traffic can be generated. To be able to infect new machines, a vulnerability is exploited. In case of human interaction this can be a user executing a file. Propagation can be described in three phases:

- Find a potentially vulnerable target
- Try to exploit the vulnerability
- Install and run the worm

Worms which exploit vulnerabilities in network-exposed services can retrieve information about potential targets locally or from a remote location, or they can scan the network. Scanning means that many hosts are contacted to check if they are potentially vulnerable. During this process many non-existing hosts or hosts which do not run a certain service are contacted. This leads to specific scanning traffic characteristics as shown below. Non-scanning worms show do not show the same typical behaviour. This will be discussed as well. Worm propagation and scanning techniques are further explained in other work [20, 78, 92], here this discussion is reduced to the aspects relevant for this work.

TCP Scanning

TCP [64] uses a three-way handshake to establish a connection. While scanning for a particular TCP service, a scanning host sends TCP SYN packets to the corresponding TCP port on many destinations to try to establish a connection. Other scanning techniques such as FIN scanning are not discussed here because they can easily be filtered with *Stateful Packet Inspection (SPI)*. Destinations to scan are chosen at random or from certain subnets such as the local one, or mixed approaches are used.

After the SYN packet has been sent, it has to be waited for a reply or a timeout. Typically several hosts are scanned in parallel to be more time efficient. Opening sockets consumes CPU and memory resources on the scanning host, which limits the maximum number of open connections. This limits the maximum scanning rate and therefore the propagation speed.

If root privileges are available, it is possible to speedup scanning by sending a SYN packet without allocating a connection structure. Incoming packets can be monitored and if a SYN/ACK packet is returned, it is known that a connection is possible. In a second step a socket is opened and the connection is established in the usual way.

A successful connection attempt is indicated by a SYN/ACK packet being returned by the target. Failed connection attempts are indicated in different ways: If the port is closed, a RST packet is returned. The same can happen, if the connection is filtered by a firewall. According to the TCP specification these reset packets have their sequence number set to zero, which helps to distinguish them from reset packets sent due to a loss of synchronisation. If the host is not reachable, a router may return an ICMP unreachable message (type 3) [63, 7]. Reasons for sending unreachable messages are e.g. if no route to the destination exists, the host cannot be found in the local network because no ARP response

has been received, or if a packet filter denies the packet to be forwarded. If the packet is lost in transmission, routers are not configured to send unreachable messages on missing ARP replies, or a firewall is silently dropping packets, the connection attempt fails with a timeout. RST packets with sequence number zero, ICMP unreachable messages and connection timeouts can therefore be used as indicators for failed connection attempts.

Because a scanning host is trying to contact potential targets which might not exist or might not run a particular service, it has an unusually high number of failed connection attempts [20]. As many distinct hosts are contacted, destination IP address entropy is high i.e. the addresses have a high distribution [87]. To reduce false positives Göldi and Hiestand [20] count the number of distinct IP addresses a host fails to establish connections to on the same port. A worm is built to exploit a particular set of vulnerabilities. It scans therefore only a limited set of ports. Connection attempts to ports which are not being scanned, are less likely to fail.

The characteristics of a TCP scanning host are summarised in the following list:

- High number of connection attempts to a high number of destinations on a limited set of destination ports
- Out of which a high number fail with:
  - TCP resets with sequence number zero
  - ICMP unreachable messages
  - Connection timeouts without reply

UDP Scanning

An infected host trying to exploit a vulnerability in a UDP [62] service, shows similar behaviour to the TCP case. Compared to the latter, UDP is not a connection-oriented transport protocol. Even if UDP itself does not have the concept of a connection, the application layer protocol may do and it is possible to use a flow-based definition of a UDP connection. A connection can be understood as one or several UDP packets with identical source and destination IP addresses and ports in one direction and one or several UDP packets with inverse address information into the opposite direction. Packets in the original direction can be referred to as requests and packets in the reply direction as responses. Here a failed connection attempt for UDP is defined as a UDP connection, which times out without a reply or gets rejected with an ICMP unreachable message.

A worm spreading with a single UDP packet is bandwith limited only. No timeouts have to be awaited and no resources have to be allocated on the infected machine. An example of such a worm is the SQL Slammer worm. It caused massive congestion in computer networks, crashed network equipment and even lead to backbone disruption [48]. Because the scanning, exploitation and installation phases of this worm are combined to a single step, its propagation speed was very fast. It infected more than 90 % of the vulnerable hosts within 10 minutes, which affected at least 75'000 hosts [48].

As in the case of TCP, a UDP scanning machine will have a high number of failed connection attempts, as many targets do not exist, do not run the

desired service or because the traffic is filtered by a firewall. For a single packet spreading worm, timeouts also occur on successful compromises.

Another characteristic of Slammer is that all packets it sends have a UDP payload size of 376 B [48]. Using packet size to identify worms is not very reliable. Depending on the vulnerability, it is easy use a padding of random size. One worm using this evasion technique is the Witty worm [34].

Witty spread in 2004 and exploited a vulnerability in several *Internet Security Service (ISS)* products. Interestingly, Witty sent packets with random destination ports. This contradicts the characteristic of a worm to address a limited set of destination ports. This was possible because ISS products analyse traffic for security reasons, even if the machine is not listening on the targeted port or is not even addressed by the packet. This property is considered to be rather an exception.

For worms which do not require a reply from the targeted host to exploit the vulnerability it is possible to spoof the source address to make identification of the source more difficult. Slammer and Witty use the real host address, but this has no implication for other worms.

The characteristics of a UDP scanning host are summarised in the following list:

- High number of flows to a high number of distinct destinations on a limited set of destination ports
- Out of which a high number:
  - Get rejected with ICMP unreachable messages
  - Stay unreplied
- Potentially high number of UDP packets or flows of same or similar size
- Potentially spoofed source address

ICMP Scanning

Instead of directly checking the reachability of a certain service on a specific target, a scanning machine can use ICMP echo request messages (ping) to check if the target generally reachable. One of the largest attacks observed by Stone-Gross et al. [80] during the 67$^{\text{th}}$ IETF meeting was an ICMP sweep, which was used to prepare for a subsequent NetBIOS worm exploit. This traffic drove the wireless channel utilisation to nearly 100 %, increased the number of retransmissions and the RTT, and caused clients to reduce their transmission speed.

As a host scans for online targets with a ping sweep, again a high number of echo requests will not be replied with an echo response: they result in an ICMP destination unreachable message being returned or they will not be answered at all.

The characteristics of a ICMP scanning host are summarised in the following list:

- High number of ICMP echo requests to a high number of distinct destinations
- Out of which a high number fail with:

– ICMP unreachable messages
– Timeout without reply

Non-Scanning Propagation

Before the characteristics of scanning traffic have been discussed. An important aspect of this is that scanning worms cause a lot of failed connection attempts while searching for possibly vulnerable targets. While scanning a lot of resources are wasted trying to contact non-existent hosts or hosts that do not run the desired vulnerable services. Staniford et al. [78] introduce *hit-list scanning*, which leads to faster propagation. Before releasing the worm, its author collects a list of potentially vulnerable systems. This list is used to kick-off the worm and gets distributed to infected children in an overlapping or non-overlapping manner. Using a hit-list results in higher propagations speed and fewer failed connection attempts.

No hit-list-only based worms have been observed [5, 20]. Hit-lists do not need to be the only propagation technique of a single worm, as stated in [78]. Shannon and Moore [75] analysed the spreading of the Witty worm and observed that it infected an initial population with exceptional speed. They concluded that this population was already under the control of the attacker or that a hit-list has been used. Further evidence in [34] showed that a single host (*patient zero*) running a different code base started the spread with the help of a hit-list. The normal worm version then, used scanning to propagate.

Precomputed hit-lists are therefore already partly being used for fast propagation. In a wider sense, a hit-list can be understood as a list of potentially vulnerable targets. It can be built dynamically and target identifiers do not necessarily have to be IP addresses. Precomputed as well as dynamic hit-lists reduce the number of failed propagation attempts. Various remote or local information sources can be used to build such a list of targets:

- Email worms use the address book of local mail user agents to find addresses to send the worm code to. The same technique could be applied by an instant messaging (IM) worm.
- Webcrawlers or Web search engines, such as Google, can be used to find vulnerable Web servers or Web applications.
- Peer-to-Peer (P2P) applications maintain connections to several neighbours or peers. A worm listening to P2P traffic on an infected machine or compromising the local P2P application itself can gain access to a highly interconnected network.

Detecting non-scanning worm propagation is considered to be difficult. Primarily because it causes less or no failed connection attempts at all. Secondarily, this is because the propagation can be hard to distinguish from benign traffic. Sending instant messages for example is benign. It is probably feasible to detect excessive message sending based on the amount of traffic or the number of flows. However, this requires application-specific definitions of normal behaviour. For a P2P multimedia application, for example, it is usual to have a high number of flows and high data rates.

A possible detection metric could be the distribution of number of packets or bytes per flow. If the same packets are always sent to exploit a given vulnerability and the malicious payload is propagated without varying padding, the flows will have similar size. "Similar size" because of potential retransmissions [16]. Worm propagation would lead to a high number of similarly sized flows. To make detection less error prone, the number of destinations per flow size could be counted. An alert could be triggered if the number of destinations for a particular flow size reaches a certain threshold. Even with this improvement a high number of false alerts is expected. An alert could e.g. be caused by a peer-to-peer application sending an identical request to a high number of peers. Additionally, depending on the number of clients and their flows, the memory requirements to maintain this state could be considerably high for a wireless router.

2.3.3 Denial-of-Service Attacks

A malware compromised machine can be used to launch a *Denial-of-Service (DoS)* attack against an arbitrary target, which prevents legitimate users from accessing a certain service. If several agents participate in the same attack, it is called a *Distributed Denial-of-Service (DDoS)* attack. Worms which infect a huge number of machines are powerful means to launch DDoS attacks. One example is the Storm worm, which launched SYN- and ICMP-flood attacks against various anti-spam Websites [79].

It is possible to cause DoS conditions by exploiting vulnerable services. For such an attack sending only a few or just a single packet can be sufficient. This can have a big impact on the target system, but does not lead to a lot of traffic in the mesh network. Detecting this kind of attack with anomaly-based detection metrics is very probably impossible, because it does not deviate from the baseline, especially not if it is targeted at common services. The same applies to privilege escalation attacks e.g. to exploiting a buffer overflow in a network exposed service.

Flooding attacks using the TCP, ICMP and UDP protocols as well as DoS attacks on the application layer are presented below.

SYN Floods

As mentioned earlier, TCP uses a three-way handshake to establish a connection:

- The client requests a connection by sending a SYN packet to the server.
- The server indicates to accept the connection by acknowledging the connection request with a SYN/ACK packet.
- The client fully establishes the connection by returning an ACK packet.

When the server receives a SYN packet it typically initialises a connection structure to handle further packets of this currently *half-open* connection. Every listening socket has a backlog queue to store half-open connections, which has a maximum size. If the queue is full, further connection request are silently dropped. During a SYN flood attack, the attacker sends a high number of SYN packets to the target to fill up its backlog queue and deny access to the service to legitimate users. The attacker does not respond to the returned SYN/ACK

packet to keep the connection half-open. The connection is never fully established and has to be kept in the backlog queue until a timeout occurs. Because the attacker does not have to store any connection state, she is able to send SYN packets at high rates, limited only by the available bandwidth. As the attack does not rely on receiving the reply from the server, the source address of the SYN packets can be forged.

It is possible to defend against this effect of a SYN flood attack [12] (e.g. with SYN cookies) but it can still be used to consume CPU resources or to congest the targets network. In the latter case, it is even possible to use random destination ports. The attack can also be extended to a connection flood attack. Here the connection is fully established by the client, but nothing is sent to the server later. Again the server has to keep a connection structure allocated until a timeout occurs. This can lead to memory exhaustion. Because an attacking host has to keep state for every SYN packet sent until it is acknowledged by the server, the maximum rate is limited. This disadvantage of the connection flood attack diminishes, if a high number of agents participate in the same attack. Even if the effects of the basic SYN flood attack can be mitigated, SYN flood attacks are still a threat today [79, 80].

If a SYN flood attack is successfully overwhelming the target or if source addresses are spoofed, one can observe a high number of unreplied SYN packets for an attacking host. Assume the source address is not spoofed and the attack is used to congest the target's network. In this case random destination ports can be used. Depending on the target's firewall configuration a high number of RST packets or ICMP unreachable messages may be returned to an attacking host.

The characteristics of a SYN flooding host are summarised in the following list:

- High number of connection attempts to a limited set of destinations
- Out of which a high number fail with:
  - Connection timeouts without reply
  - TCP resets
  - ICMP unreachable messages
- Likely attacks a limited set of destination ports
- Potentally spoofed source addresses

ICMP Floods

ICMP floods aim to saturate a target's network connection, such that legitimate traffic is hindered. The target is simply flooded with a large amount of ICMP echo request messages. ICMP floods have e.g. targeted Georgian Websites in August 2008 and had a devastating effect on their targets [2]. If the target gets congested, limits the rate of ICMP echo replies, or if the source address is spoofed, a high number of echo requests will stay unreplied.

A special type of a ICMP flood attack is the *Smurf* attack. In this attack the source IP address of an echo request is set to the target's IP address. Then this packet is sent to some networks broadcast address. If not configured to

do otherwise, all hosts in the addressed subnet will send an echo reply to the target. This leads to an amplification: the attacker has only to send one packet whereas the target receives a multiple of it.

The characteristics of an ICMP flooding host are summarised in the following list:

- High number of echo requests to a limited set of destinations
- Out of which a high number fail with:
  - Timeout without reply
  - Potential ICMP unreachable messages (e.g. administratively prohibited)
- Potentially spoofed source address
- Potentially large packet size
- Likely fixed packet size

UDP Floods

Heavy UDP traffic can be used to exhaust the target's available bandwith or to consume CPU resources, because the target has to receive and analyse the arriving packets, even if they do not have any meaningful content. In the first case random destination ports can be used, in the second the destination ports are from a limited set. Again, it is possible to use spoofed source addresses. ICMP unreachable messages may be returned, if random destination ports are addressed or if the attacker is non-silently filtered. Because UDP, unlike TCP, does not have a handshake, a UDP flood not targeting a responsive service will not cause any UDP messages to be returned.

The characteristics of a UDP flooding host are summarised in the following list:

- High number of UDP packets sent to a limited set of destinations on a random or fixed destination port
- Out of which a high number:
  - Stay unreplied
  - Potentially get rejected with an ICMP unreachable message
- Potentially spoofed source address
- Potentially large packet size
- Potentially fixed packet size

Application Layer Attacks

Before denial-of-service attacks on the network and transport layer have been discussed. Additionally, it is possible to attack the application layer to cause a huge load on the target machine. This can be achieved by forcing it to answer a large number of requests or to perform complex computations, such as intensive searches or cryptographic computations. Depending on the type of attack and the underlying transport protocol this can, but does not necessarily, have to cause congestion in the wireless mesh network.

It is difficult to distinguish application layer DoS attacks from benign requests. One indication of an attack can be a high number of flows to a single destination. This could also be caused by an intensive benign application. Furthermore, a single connection can be used to constantly cause load on the targeted server. For UDP services it is not necessarily true that no response is received as the server might send an answer to a request. Under heavy load of the target, it could however be possible that an application which normally would send a reply, is failing to do so. Again, if being filtered packets from the attacker could be rejected or its requests could be silently dropped.

The characteristics of application layer DoS attacks are summarised in the following list:

- Likely high number of short-lived flows to a limited set of destinations on a limited set of destination ports
- Out of which a high number:
  - Potentially stay unreplied
  - Potentially get rejected with ICMP unreachable messages
- Potentially spoofed source address
- Potentially fixed flow size

2.3.4 Spamming

Malware infected hosts can be used to send *Unsolicited Bulk Email (UBE)*. This includes *Unsolicited Commercial Email (UCE)* and mass mailing worm propagation. Target addresses can be specified by the spammer, can be collected locally, e.g. by analysing documents or the the victim's address book, or can be guessed. To send out emails, the mail server of the user's ISP can be used. Access to that server can be gained by directly instructing installed mail user agents to send the emails or by extracting the necessary configuration settings from them, which might also include authentication information.

Worms like SoBig or MyDoom use built in SMTP engines to send mails directly to the target's mail server [92]. This evades detection on the user's ISP's mailservers. To send emails to a specific domain, it must be known which mail servers accept emails for that domain. These servers are called *Mail Exchange (MX)* servers and can be found by querying the *Domain Name System (DNS)* with an MX query. Because these queries are not necessary to send mails over an ISP's mail server, observing them can be used to identify spam sending hosts [49].

To avoid being detected, it is possible to retrieve the IP addresses of MX servers from a non-DNS online location or to have them hardcoded, e.g. the addresses of MX servers of popular mail services. In anycase, the SMTP engine contacts a high number of distinct IP addresses to send email to [92]. Even if SMTP [32] allows to send multiple emails within a single connection and allows to address several recipients within a single message, Wong et al. [92] observe that infected hosts initiate multiple flows to the same IP address. They observed on average 2000 successful SMTP flows to 200 distinct IP addresses for a SoBig and 900 successful SMTP flows to 115 distinct IPs for a MyDoom infected client. Dübendorfer et al. [16] show that SoBig transmits only one copy per flow. The number of SMTP flows can therefore be used as an anomaly metric. For the case that multiple mails are sent within the same connection, comparably long-lived flows with a high number of bytes sent could be observed.

The characteristics of spam sending hosts are summarised in the following list:

- High number of SMTP flows potentionally to a high number of distinct IP addresses
- High amount of SMTP traffic potentionally in long-lived flows
- Likely high number of DNS MX queries (SMTP engine)

2.3.5 Password Cracking

To get access to a password protected service, a malware infected machine can be used to guess passwords. This can frequently be observed for SSH and FTP [35]. Getting access to an SSH account is particularly interesting for an attacker, because it might let him use the system to launch further attacks.

If random hosts are scanned for a running SSH or FTP service, this leads to similar characteristics than with worm propagation. The behaviour is less typical, if a specific host is attacked. Because each wrong guess is likely to lead to a connection termination, a high number of short-lived flows could be observed.

2.3.6 Malicious Ingress Traffic

The focus until now was on malicious egress traffic. This section discusses the potential impact of malicious ingress traffic.

In both wireless mesh networks presented in Section 1.1 border routers translate mesh IP addresses into public ones in a many-to-one manner. This prevents connection attempts from the outside of the mesh network. The same is the case, if connection attempts are prohibited with a packet filter. This does not prevent malicious ingress traffic which is associated with connections that have been established from the inside. A malware infected host could, for example, download additional software components or address lists, or could connect to a botnet to receive further instructions. Such flows are expected to be of short duration or low throughput and not to cause massive congestion. In terms of congestion more dangerous would be the situation, where a client connects to a malicious UDP service. Such a connection would allow the service to launch a UDP DoS flood which can traverse the NAT and cause congestion in the wireless

mesh network, as long as it sends packets with matching source and destination ports. Such an attack is difficult to distinguish from benign behaviour as it could e.g. be the case, that the client requests some kind of video stream.

In the environment Stone-Gross et al. [80] analysed, clients have been fully exposed to the Internet and neither NAT nor packet filters were deployed. They noticed that malicious ingress flows consumed less bandwith and caused fewer collisions than malicious egress flows. Ingress scanning traffic e.g. does not necessarily cover the full network range and if it does, it gets spread over the network which distributes potential congestion. The paths near the uplink node, where the traffic comes in, however will suffer from higher congestion than others. Typically not all addresses from the WMN's subnet are routable. The impact of a full network scan in terms of congestion will therefore be limited, but could pose a thread to the clients. The effect of a targeted DoS attack against a client in the WMN on the network would be much more severe, but also less likely.

It can be concluded, that it is more important to protect a wireless mesh network from the impact of malicious egress traffic than protecting it from malicious ingress traffic.

2.3.7 Confusion with Benign Traffic

In this section the characteristics of malicious traffic have been described. Benign traffic can have similar characteristics. The detection metrics, thresholds and algorithms have to be carefully chosen not to classify benign traffic as malicious. Examples of benign traffic, which could reveal similar behaviour to malicious traffic, are given here.

Peer-to-Peer Traffic

A traditional client-server architecture has a low number of highly-connected nodes and a high number of lowly-connected nodes. In contrast to this, *Peer-to-Peer (P2P)* applications form highly interconnected networks. Depending on the application, a single peer might contact a high number of peers. A high number of connection attempts might not be successful, because the peer is behind a firewall or NAT. P2P traffic could therefore be wrongly identified as scanning traffic.

Peer-to-peer applications are considered as benign here, even if they possibly use a lot of bandwith and can have legal issues. Detecting P2P traffic as malicious would therefore constitute a false positive. Openwireless St. Gallen [56] does not tolerate P2P file sharing in its network, but does not block it generally, but Operators of upstream routers are free to implement QoS measures or block unsolicited traffic. The Wray community WMN tolerates P2P traffic, but generally limits the maximum bandwith per host.

Video Streaming

For video streaming short delays and low jitter are important. Packet loss is often tolerated, because a retransmitted packet becomes useless, if it arrives after its actual playback time. UDP is therefore often used as transport protocol, e.g. in RTSP [71] and RTP [72]. For video streaming data is sent at high data and

packet rate to the receiver of the stream. This could lead to confusion with a DoS attack. Checking if a stream is unidirectional only, does not necessarily lead to more clarity. An RTP stream is unidirectional and no packets will be returned from the receiver of the stream. An application example of such a video stream is a surveillance or Webcamera streaming over RTP.

Webcrawler

Webcrawlers can be used to create a local copy of a Webpage or to check links on a Webpage to be correct. HTTP/1.1 [17] allows to issue multiple commands within the same persistent connection. If a crawler does not use persistent connections or contacts a large number of distinct IP addresses, a high number of short-lived flows can be observed. Depending on the detection metrics, this could lead to confusion with application layer worm propagation or a host-specific password cracking attack.

Chapter 3

Preliminary Performance Analysis

This chapter analyses the performance of a typical wireless router which could be used in a wireless mesh network. Section 3.1 describes the platform, its software and the test setup. In Section 3.2 the CPU usage for forwarding different types of high load traffic is analysed, Section 3.3 evaluates methods to capture packets and Section 3.4 shows how existing intrusion detection systems perform. Section 3.5 draws some conclusions of important aspects for the design of a lightweight intrusion detection system.

3.1 Test Environment

3.1.1 Hardware

A common Netgear WG302 wireless router is used to represent a mesh box in this work. This router has an Intel XScale IXP422 network processor, which has a big-endian ARMv5 architecture and runs at 266 MHz. The device has 32 MB of RAM and 8 MB of flash memory. It has an IEEE 802.11g wireless card based on an Atheros 5212 chipset. The router is available for about $ 200, which is rather expensive compared to the Linksys WRT54G which is used in the Openwireless network and costs about $ 50. The use of the more expensive WG302 is justified, because its hardware resources are comparable to the ones of the WRT54G. It just has more memory and a slightly faster CPU. The price difference can mainly be explained by the more enhanced software features of the WG302.

3.1.2 Software

A solution for a community operated WMN has to be cost effective. Besides that no money is available for overprovisioned hardware, the software solution cannot cost a lot either. OpenWRT [57] has been chosen as operating system for the router. OpenWRT is an extensible Linux distribution for network devices and is distributed under GPL. Its first versions released in 2004 were based on Linksys' GPL sources for their WRT54G. At the beginning Linksys ran Linux

on that device without releasing the source code. This was in violation to the GPL. The full source code was released to the public after the GPL violation has been reported in 2003 [45].

OpenWRT supports a wide range of devices with various architectures. It includes a toolchain which supports cross-compilation for specific architectures. A package manager makes it easy to extend the functionality which gives a lot of freedom and breaks with the static approach of conventional firmware.

In this work development revision 10848 of the newest release Kamikaze 8.08 is used. The use of a development version was necessary, as the WG302 was not fully supported in the previous release Kamikaze 7.09 and 8.08 has not been released yet. In the revision 10848 the ethernet port is still unusable. The kernel version is 2.6.24.2.

After the firmware of the WG302 has been replaced by OpenWRT, almost 4 MB of flash memory and 12 MB of RAM are being used by the operating system.

3.1.3 Test Setup

To test the performance of the router, a two-hop scenario was used. A sender emits traffic to the router, which forwards it to a receiver. The corresponding routes have been set statically and the stations were less than 2 m apart. All wireless cards were based on the Atheros 5212 chipset and every machine used the MadWifi [39] wireless drivers. IEEE 802.11g was the standard used in all experiments. Every card was operating in ad-hoc mode on channel 8 (2.447 GHz). The experiments have been carried out in the computer systems laboratory at Lancaster University. The environment was not completely free from interference. The directly neighboring channels were free, the campus wireless network was on channel 1 and other networks with low traffic volume were on channels 6 and 11.

Different sources have been used to generate traffic. They are described in Section 3.2. Each test was repeated three times and lasted for 60 seconds. The results from the three series have been averaged. The CPU usage on the router was measured with *Top* [66]. Top was run in batch mode and wrote statistics to a file every 10 seconds. To exclude initial effects until saturation, the values of the last 5 periods have been averaged to a run's CPU usage. During the test runs no packets have been filtered on the router, if not indicated otherwise.

3.1.4 Connection Tracking

Connection tracking is an optional functionality provided by the kernel. It associates single packets with a connection. This is required for dynamic NAT and stateful inspection. In OpenWRT connection tracking is enabled by default. During the measurements an overhead for tracking connections has been observed. To understand these effects, the connection tracking mechanism is explained here.

Netfilter [53] is the packet filtering framework of the Linux kernel. It inserts several hooks into the Linux networking stack for packet handling at different stages. One of its subsystem is the connection tracking. The Netfilter architecture is described in [69, 70], and [6] explains the connection tracking.

The connection tracking system is implemented with a hash table. Each bucket contains a linked list of hash tuples. Each tuple includes the relevant layer-3 and layer-4 protocol information to represent a unidirectional flow. Two hash tuples are embedded in a *conntrack* which represents a bidirectional connection.

Packets coming in from the network first enter the PREROUTE hook and get delivered to the connection tracking module. First, it is checked, if the packet belongs to an already existing connection. For this, the packet is converted into a tuple, which is hashed to obtain its bucket in the hash table. The linked list in this bucket is followed and entries are checked to match the tuple. If none matches, a new conntrack structure is allocated, else the packet is associated with the existing one. Later during packet filtering, the connection state of the packet can be checked. If the filter allows the packet to traverse and the packet starts a new connection, the conntrack is stored in the hash table. Both tuples are hashed and added to the linked list of the corresponding buckets. The maximum number of conntracks is limited, in the default configuration of OpenWRT to 4096 entries. If the table is full, the least recently created unassured connection of a bucket is replaced. A TCP connection is assured, if it was successfully established. General UDP connections do not get assured, but they can be assured if a helper module is used. If no unassured connection is found in the current bucket, the linked lists of other buckets are traversed. If after traversing the linked list of a bucket, the number of inspected entries is greater than 8 and still no unassured connection was found, the packet is dropped.

3.2 Packet Forwarding

This section analyses the CPU utilisation on the router while it is forwarding packets at high data rates. No other tasks are performed during this test. Both normal and attack conditions are evaluated. Figure 3.1 summarises the CPU usage for the test cases presented below.

3.2.1 Netperf

Netperf [28] is a client and server application to perform network benchmarks. The sender establishes a TCP connection to the receiver and sends as fast as possible. Netperf reports the payload throughput at the end of the measurement. In a single-hop scenario, where the sender sends directly to the receiver without involvement of the router, the maximum throughput was 25312 kbps. If the traffic is forwarded by the router, the channel is utilised twice per packet and the throughput dropped to 12547 kbps. The CPU utilisation on the router to forward the packets was 8 %.

3.2.2 Wray Traffic

Lancaster University has access to the Wray community wireless mesh network. To have a set of real world network traces, traffic in Wray has been captured from 10 April to 28 April 2008. On every mesh box *Tcpdump* [81] was used to capture the first 68 B of every packet entering or leaving the ethernet port or the

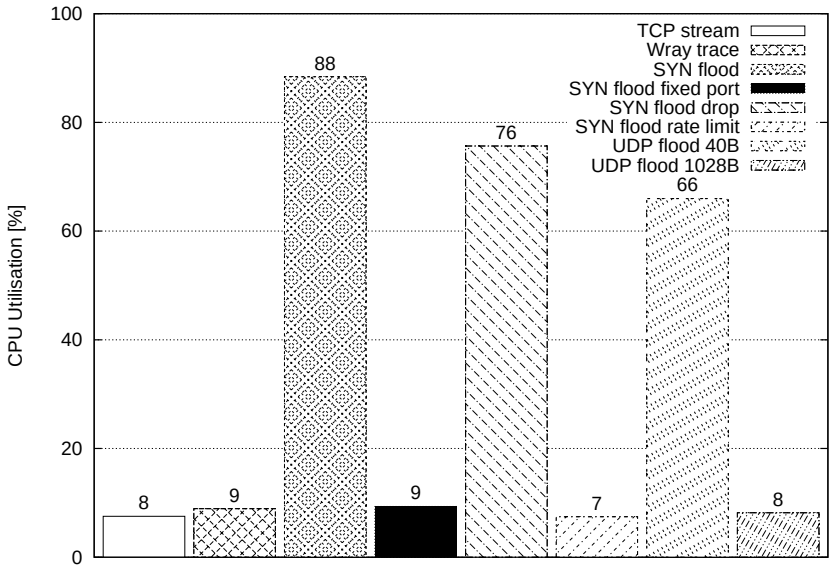


Figure 3.1: CPU utilisation to forward packets. Traffic with a high rate of new connections consumes significant CPU resources due to connection tracking overhead. To reduce the CPU load it is important to prevent packets from reaching the connection tracking system.

client wireless interface. This allowed to identify single clients, as no addresses have been translated at this stage. The data was written to a file in a RAM drive. Tcpdump was restarted hourly and the capture file was renamed. During this process some packets got lost. The capture file of the previous period was compressed and uploaded to a statistics server at Lancaster University. During postprocessing all broadcast traffic was removed from the traces and it was assured that IP addresses assigned to hosts connected to ethernet ports on different boxes were unique. All traces were merged into a single file and were split into clean hourly chunks.

In the performance analysis here, the traffic from 18:00 until 19:00 on 23 April 2008 was replayed. During this hour the largest number of packets has been transmitted. The traffic was replayed with *Tcpreplay* [82]. The packets have been padded to their original length and were sent to the router. The packet flow sent contained both ingress and egress traffic. The router had a default route set to the receiver, which simply dropped everything. The captured traffic was replayed at the maximum possible speed, rather than the speed it was actually captured.

During this experiment, the CPU usage on the router was 9 %. This slight difference to the Netperf experiment can be explained by smaller packet sizes on average and a higher number of flows.

3.2.3 SYN Flood

A SYN flood attack has been simulated. The source IP address was not forged and the source port was randomly chosen. The receiver silently dropped all packets. SYN packets were sent at maximum rate. On average 2737 packets per second were sent, which corresponds to a data rate of 876 kbps. During the attack the CPU usage on the router was 88 %. This high CPU load is caused by every SYN packet corresponding to a new connection. The connection tracking subsystem performs a hash table lookup, has to allocate a new conntrack structure for every packet, and has to calculate two hash values to store it in the table.

In a variation of the attack, SYN packets have been generated with a fixed source port. Because all packets have now the same source and destination IP addresses and ports, they all relate to the same connection. In this case, the CPU usage was only 9 % and packets could be sent with a rate of 3943 pps. The higher packet rate shows, that while the kernel is very busy during a random source port attack, it is not able to forward the same number of packets. A higher CPU load also results in higher round-trip-times (RTT). Measurements with ping showed a RTT of 30 ms from a third host to the attack destination during an attack with fixed source ports and 54 ms with random source ports. The delay is much higher in the second case, even if there are less packets in transmission which could cause contention. The bitrate of the fixed port SYN flood attack is 1262 kbps, much lower than the 12547 kbps which were achieved with Netperf. This is because the channel has to be sensed free for at least the inter frame spacing before the next packet can be sent. With short frames such as SYN packets, the channel can be used less to actually transmit data.

The SYN flood with randomised source ports has been rerun, while this time, all packets have been dropped at the router. The CPU usage in this experiment was 76 % and the packet rate 7771 pps. The packet rate is twice as high as the rate in the last case with the fixed source port, because the wireless channel is only utilised once per packet. The difference to the unfiltered SYN flood with randomised source ports is, that no conntrack entry is stored in the hash table, because the filter denies the packet to be forwarded. The conntrack has still to be allocated, but the initial lookup is fast, because the linked lists are empty, and no hash values have to be calculated to store the conntrack in the table.

We have seen that dropping the packets in the filter table is not enough to reduce the CPU load drastically. Processing of single packets becomes cheaper, but higher packet rates are possible. To bring the CPU load to an acceptable level it is important to prevent the packets from being analysed by the connection tracking subsystem. Netfilter provides the raw table, which is traversed before the packet is passed to connection tracking. A feasible approach is to limit the rate at which SYN packets are accepted. The following Iptables commands show, how this is done:

```
iptables -t raw -A PREROUTING -p tcp --syn -m limit \
  --limit 512/s --limit-burst 1024 -j ACCEPT
iptables -t raw -A PREROUTING -p tcp --syn -j DROP
```

The first rule allows 512 SYN packets per second with an initial burst of 1024 packets to traverse the stack further. To understand the rate limiting one can imagine a bucket with an initial capacity of 1024 tokens. For every arriving

SYN packet a token is put into the bucket. If the bucket overflows while adding a new token, the packet does not match the rule. Every second 512 tokens are removed from the bucket if possible. The second rule then drops all SYN packets, which caused the bucket to overflow. The values chosen here should be high enough for most environments. With this rate limitation, a SYN flood causes only 7 % CPU usage.

3.2.4 UDP Flood

Preventing excessive CPU consumption during a SYN flood attack was easy, because new TCP connections can easily be identified by examination of the TCP flags. But also a UDP flood can use random source or destination ports. In this case, every packet belongs to a new connection as well. Generally rate limiting UDP packets before connection tracking is not practical, because new connections cannot be identified and limiting the UDP rate would affect benign traffic. For connection tracking it does not matter if the source or the destination port is random. In either case or in combination the packets will have different hash values.

To compare the effects of a UDP flood to a SYN flood, the UDP payload length has been chosen as 12 B, which including UDP and IP headers is 40 B per packet. Only the source port has been chosen at random. It was possible to send packets at a rate of 3293 pps. This caused a CPU load of 66 %. For the SYN flood the load was 88 % at 2737 pps. For TCP packets more checks are performed. The TCP state is checked, sequence numbers and window lengths are validated and possible TCP options are examined. The connection tracking kernel code for TCP has 1410 lines, whereas the code for UDP has 240 lines. Even with additional checks for TCP, the difference seems still to be quite big and it remains unclear if there are additional effects.

In a UDP flood attack, it is likely that the packet size is bigger than 40 B, because the objective might be to congest the target's network links. The UDP flood has been rerun with a UDP payload length of 1000 B or an IP length of 1028 B. The throughput was 13721 kbps and packets have been sent at 1668 pps. The CPU load was only 8 %. Because of the bigger packet size, the packet rate and therefore the overhead for connection tracking is lower. A non-linearity in the CPU usage can be observed. The CPU usage with 1668 pps is 8 %, whereas it is 66 % for 2737 pps with smaller packet lengths. Very high packet rates have an disproportionate effect on the CPU consumption. They push the router to its capacity limits.

3.3 Packet Capturing

During experiments with existing intrusion detection systems, packet capturing was causing substantial load under high traffic volume. To be able to understand the isolated performance impact of packet capturing this section first evaluates different capturing methods.

Packet capturing is not only a problem on platforms with very limited resources such as the wireless router used here, but also for powerful systems capturing traffic from high speed links (1–10 Gbps). The focus for the wireless

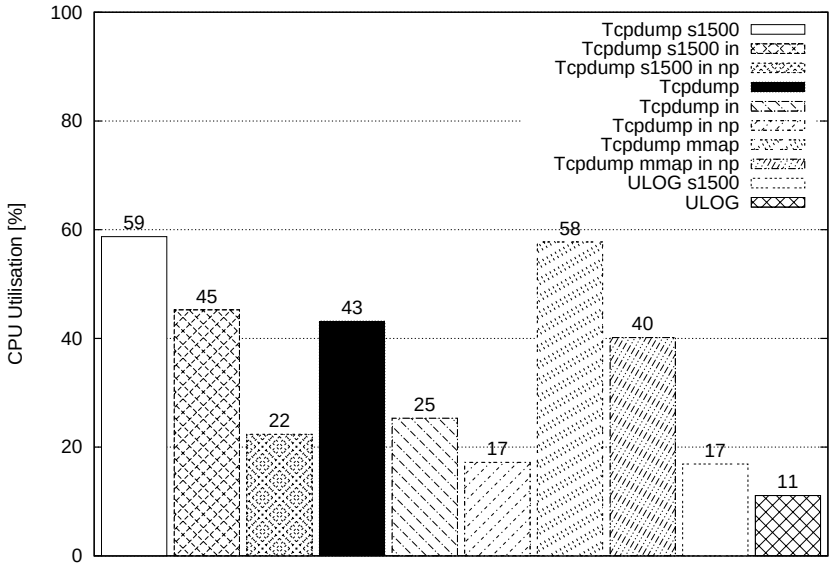


Figure 3.2: CPU utilisation for packet capturing. *s1500* denotes capturing of full packets, *in* capturing of incoming packets only, *np* non-promiscuous capturing and *mmap* capturing with the memory mapped version of libpcap. Capturing packets at their full length is more expensive than capturing packet headers. Libpcap requires a filter to capture incoming packets only. The memory mapped version performs poorly and best performance can be achieved with ULOG.

router stays more on reducing the CPU load, whereas for high speed capturing reducing high packet drop rates is the main issue [15].

For all experiments Netperf has been used to generate a single TCP stream at full throughput. The resulting CPU usage for different capturing methods are summarised in Figure 3.2.

3.3.1 Libpcap

The most popular way to capture packets is with the help of the high level capturing library *libpcap* [36]. Programs using it include Tcpdump, Snort and Bro. Libpcap allows the network interface card to be set into *promiscuous* mode. This lets the card forward packets to the kernel, which are not addressed to the capturing machine on the link layer. Packets can be filtered in kernel space with the *BSD Packet Filter (BPF)* [43] language. Solicited packets then get forwarded to user space through a raw PF_PACKET socket. By bypassing the network stack, this type of socket gives access to all packets including link layer information without having to bind to a specific port. Libpcap also provides a standardised packet format, which can be used to store packets in files. To test libpcap, Tcpdump was used with libpcap 0.9.4. All packets received from the wireless card were simply written to `/dev/null`.

During the first tests, full packets were captured: the Tcpdump *snaplen* option was set to 1500. While capturing the Netperf generated TCP stream the CPU usage was 59 %. Most of the time was spent to handle interrupts, especially software interrupts. The costs of moving packets from the network card to the kernel and then to user space are high. Even if the CPU was not completely utilised, 15.5 % of the packets were dropped by the kernel before Tcpdump could read them. A possible explanation for this is, that while packets arrive at high rates, the low priority user space application does not get enough runtime to process packets before the socket buffer overruns. Increasing the socket buffer would reduce the problem of overruns, as long as enough CPU time is available.

Libpcap reads packets at the link layer. The consequence of this is that packets which get routed by the capturing machine, will be received twice. Once when it comes in from the network card and once when it leaves it again. This is the case if either the packet enters and leaves at the same interface, as in a wireless multihop network, or if libpcap is not bound to a specific network interface card. If the duplicated packets are forwarded to user space this causes an unnecessary overhead and has to be addressed by the application. With BPF it is possible to prevent packets from being captured in the kernel. To capture every packet only once, a BPF rule was set to accept only incoming packets. The CPU usage was reduced to 45 %. The packet drop rate was below one percent.

By default Tcpdump sets the network card into promiscuous mode, but it is also possible run it without doing so. Measurements have shown that not using the promiscuous mode reduces the computation time to handle hardware and software interrupts. While capturing only incoming packets in non-promiscuous mode the CPU usage was only 22 %. The packet drop rate was again below one percent. Why disabling the promiscuous mode has such an impact on performance remains unclear. Promiscuous mode should just set a bit on the network card, which leads it to accept all packets, to whatever MAC address they might be addressed to. There was no other traffic on the network, than the generated

TCP stream of which every packet was either addressed to the router or sent by it. Enabling or disabling promiscuous mode would therefore be expected to behave identically. To investigate this behaviour further, a Netperf stream has been sent to the routers ethernet interface on the wired network. Capturing this traffic had identical performance, independent if promiscuous mode has been enabled or not. This suggests, that the unexpected behaviour is caused by the MadWifi wireless driver. This driver uses a proprietary *Hardware Abstraction Layer (HAL)*, which is released in binary format only. Other people using the MadWifi drivers observed performance drawbacks in promiscuous mode as well[1]. A request for explanation to the MadWifi IRC channel remained unreplied.

To be able to perform content inspection, packets have to be captured at their full length. For anomaly-based detection it is sufficient to capture only packet headers. The same measurements as presented above were repeated with Tcpdump's default snaplen. In this way, only up to 96 B of each packet were captured. This reduces the overhead for copying packets and increases the number of packets which can be stored in a buffer of the same size. Capturing only the first bytes of the packets reduced the CPU load to 43 % if all packets, 25 % if only incoming packets and 17 % if the card was in non-promiscuous mode and only incoming packets were captured. For the case that all packets were captured, the packet drop rate was at 7 % and for the other cases below one percent. Capturing only packet headers therefore leads to substantial performance gains, unless for capturing only incoming packets in non-promiscuous mode, there the difference of 5 % is not very big.

3.3.2 Memory Mapped Libpcap

Libpcap MMAP [93] uses a memory mapped ring buffer instead of the packet socket to forward packets from kernel to user space. Its current version is based on libpcap 0.9.8 and has been reported to reduce packet drops during high traffic loads [15]. During the tests the size of the ring buffer was set to hold 625 packets. Only the first 96 B of the packets were captured.

While capturing all packets from the TCP stream including duplicates due to forwarding, the CPU usage was 58 % (43 % with standard libpcap). All packets could be captured, whereas without memory mapping 6.9 % were dropped. Having no packet drops at all comes at the cost of a higher CPU utilisation. For the case that only incoming packets are recorded and the network device is not put into promiscuous mode, the CPU usage was 40 % (17 % without MMAP). Again no packet drops occurred. Capturing only incoming packets in non-promiscuous mode lead to a big performance gain with the standard version of libpcap. Here the resource saving effect is much smaller and the CPU usage of the memory mapped version is considerably higher.

3.3.3 Libpcap PF_RING

PF_RING is a new family of sockets introduced by [15]. It is designed for high speed traffic capturing and increases the maximum speed at which packets can be recorded. The improvement is achieved by storing packets in a ring buffer right when they come in from the network card. It is possible to drop packets

[1] `http://madwifi.org/ticket/507`

after they have been copied to the buffer to reduce the overhead of traversing the networking stack. This is not possible in the wireless mesh case, because it would render the network unusable for other purposes. Further it is possible to apply complex kernel level filtering and packet sampling. To avoid system calls the ring buffer is exported to user space via memory mapping. A kernel patch and a modified version of libpcap are required.

PF_RING has been installed on the WG302. On a first view everything seemed to work fine. The module was loaded, PF_RING statistics were available and the test tool was running. During the performance test however hardly any packets were captured. Further wrong statistics were reported and Tcpdump hung occasionally. This behaviour together with two minor bugs have been reported to the PF_RING mailing list[2], but the posting remained unreplied. PF_RING was supported in OpenWRT until January 2006. Then it got removed upon a report of similar problems to those experienced here[3]. PF_RING seems to suffer from unknown portability issues. A possible explanation is given in Section 3.3.5. Due to other possibilities to capture traffic no detailed debugging has been performed.

3.3.4 Netfilter ULOG

Linux not only provides access to packets over the PF_PACKET socket, but also over the PF_NETLINK socket. Netlink is used to transfer information between kernel and user space processes. The netlink protocol NETLINK_NFLOG gives access to Netfilter ULOG. ULOG is used for user space logging of packets for firewalling applications. To export packets to user space, they have to match a Netfilter rule with the ULOG target. An example Iptables command to log all forwarded packets at their full length would be:

```
iptables -A -FORWARD -j ULOG
```

The problem of duplicated packets does not exist, because packets captured at the network layer. Placing different rules allows to specifically select which packets have to be captured. Each filter rule can specify a different number of bytes which should be copied from a matching packet. With the *qthreshold* option, up to 50 packets can be aggregated in a Netlink multipart message. This avoids the need of a system calls for each single packet, as multiple packets can be obtained with a single system call. Fewer system calls mean fewer context switches which increases performance. The cost of multipart messages is a small delay as packets first get queued. If the multipart message is not full until a timeout occurs (10 ms by default), the message is sent to user space anyway. An interface for user space access is provided by the *libipulog* library from Ulogd [90], a user space packet logging daemon.

ULOG cannot be used to capture packets in promiscuous mode. It is possible to set the network card into promiscuous mode manually, but packets not addressed to the capturing machine on the link layer will be discarded by `ip_rcv()` before they reach the PREROUTE hook (see `net/ipv4/ip_input.c`). The PREROUTE chain would be the first occasion where they could be selected to be captured with ULOG.

[2] `http://listgateway.unipi.it/pipermail/ntop-misc/2008-April/001080.html`
[3] `https://dev.openwrt.org/ticket/262`

Using the Netfilter capabilities to capture packets also brings the advantage of other kernel services. IP checksums are validated before packets are handed to Netfilter [70]. The connection tracking subsystem then performs IP defragmentation before letting packets traverse further [6]. Frames with an invalid transport layer checksum do can be filtered, as they will have state INVALID. It is therefore not necessary to perform the same operations in the application, if connection tracking is enabled and Netfilter facilities are used to get access to packets. IP defragmentation defeats detection evasion and checksum validation protects from analysing corrupted packets.

To analyse the performance a test program has been written to simply decode multipart messages and count the number of packets it receives. ULOG does not provide detailed statistics about dropped packets. It only reports that the socket buffer had overflown and an unknown number of packets have been dropped by returning an error (`ENOBUFS`) during the `read()` call. To get the exact number of dropped packets the number of received packets has to be compared against the number of packets matching the Netfilter rule.

Capturing packets at their full length caused a CPU usage of 17 % and capturing only the first 96 B 11 %. No packets were dropped. Regarding that 8 % CPU usage was caused just by routing the Netperf traffic ULOG is very efficient.

During the tests it was observed that the MAC header information in the ULOG messages was empty. Some debugging showed, that the MadWifi wireless drivers set an invalid length for the MAC header in kernel structure `sk_buff`. Insted of setting it to 14 B, it was set to 90 B. With this length the invalid MAC header included even parts of the IP packet. Netfilter copies the MAC header only to the ULOG message, if it is at most 80 B long. A bug report against MadWifi has been filed[4]. The ticket is still open and the problem still exists at the time of writing. To be able to obtain the MAC header already before the problem is fixed, I wrote a kernel patch which copies as much as possible instead of discarding the whole header, if the length is too long.

3.3.5 Netfilter NFLOG

Netfilter NFLOG is the successor of Netfilter ULOG. Everything has been rewritten and integrated into the Netlink Netfilter subsystem. The socket family used is still PF_NETLINK but the protocol is now NETLINK_NETFILTER. NFLOG can aggregate more than 50 packets in multipart messages and provides support for IPv6. The default timeout to flush packets to user space is one second. Packets are selected for logging by matching a rule with the NFLOG target. A user space interface is provided by *libnetfilter_log* [51]. This library builds on *libnfnetlink* [52] which provides a generic messaging infrastructure for the in-kernel Netfilter subsystem. NFLOG requires kernel version 2.6.14 or newer.

Libnetfilter_log comes with a test utility called *nfulnl_test* which prints some information about received packets. Running this tool on the router failed already at the first library call. The cause was that the developers of the used libraries did not ensure that all data structures were properly byte aligned. The processor of the WG302 is an ARMv5 processor. It can only access words in

[4] `http://madwifi.org/ticket/1943`

memory whose start address is a multiple of 4. If this is not the case, the result of the operation is incorrect. It has e.g. been observed that the assignment of a small value to an integer resulted in a halfword swap. Architectures such as x86 can access words at any location. If the word it is not properly aligned, multiple load operations are performed and the result is combined in a register. This decreases performance but the result is correct. To illustrate the problem, assume the following C code snippet:

```
char buf[256];
int *value_p = (int*)buf;
```

As `buf` is only supposed to be accessed byte-wise, there is no guarantee that it is aligned to a 4 byte address. To dereference `value_p` word instructions are used and the access is therefore not safe on all platforms. To fix this problem, the buffer can be allocated dynamically. Dynamically allocated memory always fullfills the alignment requirements of all data types. Using a union helps as well, as the union will be aligned to the lowest common multiple of the alignments of all members:

```
union {
  char buf[256];
  int value;
} u;
```

Alternatively the *gcc* attribute "aligned" can be used to force an alignment to the maximum useful alignment of the target machine:

```
char buf[256] __attribute__ ((aligned));
```

If instead of the start address of the array an address at an arbitrary offset is casted, it must still be made sure, that the address is legal. While copying data structures to the array, data structures with variable length should be padded, such that the next data structure starts at a legal address. Data structures with fixed lengths are automatically padded by the compiler. If proper byte alignment cannot be assured, a cast is not safe and the data has to be copied byte-wise to an aligned location first.

I have fixed several byte alignment problems in the Netfilter libraries[5] and the patches have been integrated in their main releases (libnfnetlink 0.0.39, libnetfilter_log 0.0.14, libnetfilter_conntrack 0.0.95 and libnetfilter_queue 0.0.16). With the patches *nfulnl_test* is able to set up properly but crashes with a segmentation fault as soon as the first packet is received and a pointer to the payload is retrieved. There is still a portability problem left, but it was not possible to locate it. Byte alignment errors lead to seemingly non-deterministic behaviour.

[5]Netfilter libraries byte alignment patches:
https://git.netfilter.org/cgi-bin/gitweb.cgi?p=libnfnetlink.git;a=commit;h=6ea730b9d9d1d066c4cb028879131e26d90479d1
https://git.netfilter.org/cgi-bin/gitweb.cgi?p=libnetfilter_log.git;a=commit;h=aa38b2ffd4d4d738a7ebb6981afc778174d56753
https://git.netfilter.org/cgi-bin/gitweb.cgi?p=libnetfilter_conntrack.git;a=commit;h=a47403d558f98368366116bf3aad8a6932671fc6
https://git.netfilter.org/cgi-bin/gitweb.cgi?p=libnetfilter_queue.git;a=commit;h=d6bf05a9974ea3bd46c69fe437291b561ce994b4

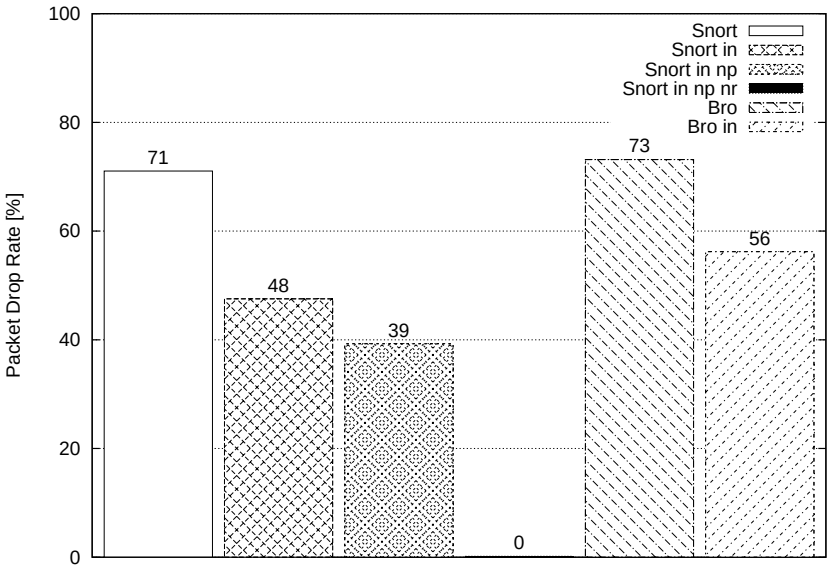


Figure 3.3: Packet Drop Rate for Snort and Bro. *in* denotes capturing of incoming packets only, *np* non-promiscuous capturing and *nr* disabling of all detection rules. Both systems suffer from high packet drop rates because of CPU exhaustion unless Snort is operated without or just with a small number of active signatures.

They are hard to locate in complex designs, where the problems are not obvious. Additionally, it seems that many developers are not familiar with byte alignment, because it is not an issue on common platforms such as x86. Taking care of byte ordering is more known. It is possible that the problems with PF_RING were caused by improper byte alignment as well.

It is not expected that NFLOG gives much better performance than ULOG, because they are both very similar and the performance overhead for ULOG is already very low. NFLOG is therefore not analysed here.

3.4 Existing Intrusion Detection Systems

This section analyses the performance of two popular open source intrusion detection systems under load: Snort and Bro. A single TCP stream has been generated with Netperf at maximum speed. The packet drop rates for Snort and Bro are summarised in Figure 3.3.

3.4.1 Snort

For this test Snort version 2.4.4 with the latest official and freely available detection rules dated back to 2005 has been installed on the router. The default configuration has been used and a set of 958 rules out of 3573 has been activated (bad-traffic, exploit, scan, dos, ddos, dns, tftp, icmp, netbios, smtp, virus).

None of the rules active in the tests matches the Netperf traffic and therefore no signatures are evaluated.

During the first test no BPF packet filters where in place: all packets have been captured twice. The CPU usage in this case was 99 % and Snort was not able to inspect all the traffic: 71 % of the packets were dropped. Such a high drop rate does not allow correct stream reassembly, which makes Snort's pattern matching easy to evade. In Section 3.3 a similar scenario with Tcpdump caused 59 % CPU usage. A big part of the high CPU load is caused by packet capturing. With a BPF filter which captures only incoming packets, the CPU load for Snort was 94 % whereas the packet drop rate could be reduced to 48 %. If Snort additionally is run in non-promiscuous mode, the CPU usage was 100 % and the packet drop rate 39 %. The packet drop rate is still unacceptably high. Because the CPU is fully exhausted, it does not help to increase the kernel side socket buffer to reduce the drop rate. Snort is too busy processing packets. Here no rules had to be evaluated and it has to be expected that performance further decreases as soon as packets match active rules and their signatures are checked. Additionally, it has to be mentioned that more general traffic will cause a higher load. General traffic has smaller packet sizes and is more diverse, i.e. consists of a higher number of concurrent flows. Packet capturing as well as hash table lookups such as for connection tracking will be more expensive.

To capture packets at their full length with ULOG, the CPU usage was 17 %. Tcpdump used 22 % if only incoming packets are captured and promiscuous mode has been disabled. The difference is too small to suggest that Snort would perform substantially better with ULOG instead of libpcap. As the CPU load for Tcpdump is now much lower than the one for Snort, it becomes clear, that Snort itself suffers from performance problems.

To analyse the performance impact of the rule lookup, all rules have been disabled. In this case the CPU usage came down to 58 %. The packet drop rate was now below one percent. Just the search for matching rules with a moderate number of active rules seems to be very expensive, even if the rules are organised in a tree and none of them matches the incoming packet. Other work [96] declares the rule lookup to consume the majority of CPU time. To achieve a good coverage of attacks with a signature-based IDS, it is important to activate a high number of rules. Using fewer rules has to be traded off against the security requirements. The use of only 27 % of the rules of an old signature database is probably already at the lower limit of what is acceptable. To protect a specific infrastructure against attacks, it is sufficient just to activate relevant rules. To detect attacks against arbitrary targets originating from within a WMN, this is not possible because the attack space is much wider.

The following preprocessors were active during the tests presented here:

Flow performs flow tracking and is required for port-scan detection. The overhead caused by this module should be small. As with the Netperf traffic only one flow exists, no allocations of new flows are required and the lookup per packet should be fast as the hash tables are basically empty.

Frag3 performs IP defragmentation to prevent detection evasion. As the Netperf traffic is not fragmented, the overhead caused by this module should be small.

Stream4 performs TCP stream reassembly and allows stateful inspection. This

module is kept busy by the Netperf traffic to follow sequence numbers and to reassemble the TCP stream.

Xlink2state detects exploits of a Microsoft Exchange vulnerability [44]. This module most likely tracks exchange connections and should not cause an overhead for other flows.

Additional experiments have been carried out to analyse the performance impact of the preprocessors. Again no rules have been activated. If all preprocessors have been disabled, the CPU usage was 31 %, caused by packet capturing and decoding. Enabling only the stream reassembly preprocessor resulted in 43 % CPU utilisation. Enabling the other mentioned preprocessors but not the stream reassembler caused 43 % CPU load as well, whereas the load was distributed among the three modules. Stream reassembly causes the most resource consumption during preprocessing. It involves hash table lookups and memory operations which are both believed to be expensive.

Snort is considered to be a lightweight intrusion detection system. As we could see, it is not lightweight enough to be reliably deployed on resource constrained hardware such as the WG302 under high traffic load. The high CPU usage for the basic services such as packet capturing and stream reassembly and the big performance overhead of the rule lookup even in the absence of evaluated rules leads to a high packet drop rate which renders stream reassembly unusable. This suggests that signature-based content analysis is not feasible in a reliable way in a wireless mesh networking environment without more powerful hardware.

During the SYN-Flood attack a reduction of the reachable throughput has been observed. The kernel was busy doing high priority connection tracking, which caused it to reduce the forwarding capacity. With Snort the data rate was 12296 kbps during the first test which is almost unaffected by the high CPU load. The kernel has a higher priority in general and it seems that copying packets to user space has a lower priority than forwarding them. Non-interference of high CPU usage with packet forwarding is helpful to fulfil Requirement 5.

3.4.2 Bro

Cross-compiling Bro version 1.3.2 for the WG302 required some of effort, due to its complex build process which has not been designed with cross-compilation in mind. Göldi and Hiestand [20] use Bro for their failed connection based anomaly detection. Their policy files have been used in this test. Their TCP script catches Bro's events on timed out and rejected TCP connections. During the test no such events are generated, as Netperf's connection attempts are successful. The UDP and the ICMP tests were disabled, because the scripts have been designed for an older version of Bro and the syntax changed in the meantime. No other policy scripts were active during the tests.

The results were very similar to those with Snort. Without filtering the CPU usage was 99 % and the packet drop rate 73 %. When capturing incoming packets only, the CPU usage was 99 % and the packet drop rate 56 %. Such a high packet drop rate renders the connection tracking unreliable and misses connection attempts as well as connection rejection by TCP reset or ICMP unreachable messages. The Netperf traffic does not match any application protocols, consists of a single flow and does not trigger any events to be delivered.

The computation overhead is caused by Bro's basic services: packet decoding, connection lookup, stream reassembly, application layer protocol lookup and checking if any events need to be delivered.

With the used policy scripts Bro was basically running idle. Compared to the performance of Snort without active rules, the basic performance of Bro is much worse. Bro has been designed to provide a rich set of functionality which is not fully required here. To analyse failed connection attempts no streams need to be reassembled or application layer information to be gathered. Connection tracking is the only functionality which is really required. Even if the checks necessary for the detection algorithm are simple and cheap, it is not possible to use Bro to deploy them on resource constrained hardware. A dedicated more resource saving solution is required.

3.5 Conclusions

In this chapter we have seen, that low cost wireless routers are not overprovisioned with hardware resources. Tracking connection states in the presence of attacks that include a high number of flows exhausts the CPU resources and reduces the forwarding capacity. Packet capturing consumes a lot of CPU time at high throughput. The capturing mechanism as well as its settings have a big impact on performance. Capturing packets at their full length is more expensive than capturing packet headers.

Other work [41, 59, 10] proposes, that a wireless node could monitor its neighbour's packet forwarding behaviour to detect misbehaving nodes. This suggestion has to be treated with care, as it might not be realisable reliably on resource constrained hardware, such as that analysed here. Monitoring neighbours implies promiscuous mode and duplicate packet capturing. If not only the routing behaviour but also potential packet modifications had to be observed, even full packets would have to be captured. It was shown that this causes high CPU load. If the implementation of the observation algorithm is not fast enough, packet drops will occur, which would render such a mechanism unusable. For the same performance reasons, the malicious traffic detection system in this work is restricted to analyse only traffic which gets routed through the monitoring node.

Two existing intrusion detection systems that perform deep packet inspection were shown to be unsuitable under high traffic volume. Both Snort and Bro suffered from very high CPU load which resulted in a large packet drop rate. The rule lookup had the biggest performance impact on Snort, whereas full length packet capturing and stream reassembly contributed a considerable amount. Even if Bro was used to apply cheap detection metrics, it was not fast enough to handle high traffic loads. The test traffic consisted of a single TCP stream with large packet size.

Anomaly detection based on abstract traffic characteristics is believed to be able to detect a wide range of excessive malicious traffic while being resource efficient. Low cost anomaly detection requires capturing of only packet headers and and does not require stream reassembly. Because detection requires a deviation from a baseline of normal behaviour, anomaly detection does not allow to detect all kind of attacks. Some attacks might stay below the deviation threshold or might not exhibit abnormal behaviour at all. Signature-based

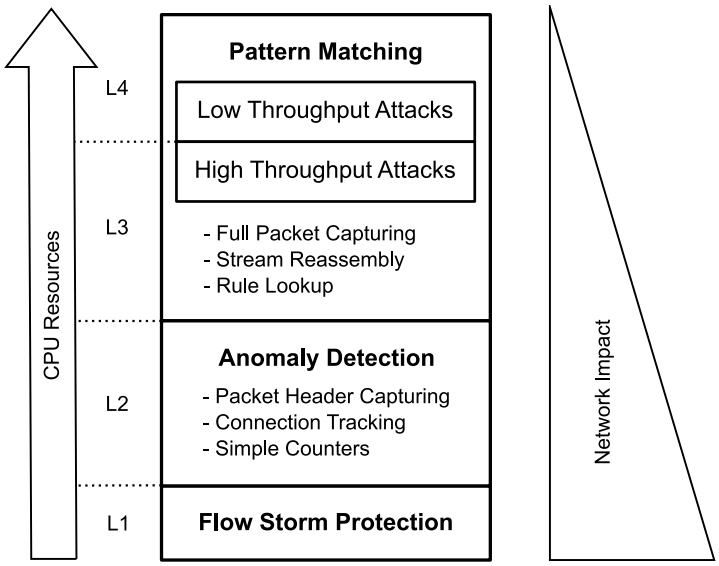


Figure 3.4: Resource dependent IDS stages

intrusion detection on the other side detects attacks based on packet content but has a big performance overhead caused by packet capturing at full length, stream reassembly and rule lookups. It makes sense to use pattern matching to complement anomaly detection to cover a bigger attack space.

Figure 3.4 shows a resource-aware architecture of an IDS combining both detection approaches. The idea is to perform intrusion detection depending on the availability of CPU resources. As long as little CPU time is available, only efficient detection metrics which cover attacks with the biggest impact are deployed. When the CPU load on the system decreases, e.g. because of a lower traffic occurrence, the detection is extended to more expensive metrics which cover a wider range of attacks. This dynamic intrusion detection at different stages allows to reliably cover the widest possible set of attacks. With a static approach either the detected amount of attacks is limited or detection is at risk to become unreliable at higher load levels, e.g. due to incorrect connection tracking or stream reassembly. The following intrusion detection levels are defined:

L1 Flow Storm Protection: Attacks with a high number of flows have a severe impact on the router's performance and its forwarding capacity. It is particularly important to defeat these effects by rate limiting aggressive hosts to ensure proper router operation and intrusion detection.

L2 Anomaly Detection based on abstract characteristics requires only to capture packet headers and to perform connection tracking. Detection is based on counting events and comparing the counters to the baseline. Anomaly detection allows to detect a wide range of excessive malicious traffic in a cheap way.

L3 Pattern Matching (High Throughput): For content inspection it is necessary to capture packets at their full length and to reassemble TCP

streams. Because the rule lookup is expensive, only the most important rules which cover attacks with a big impact on the network are activated at this level. Optimally they only cover attacks, which cannot be detected by the anomaly detection. This might include signatures of popular worms and known DoS flooding tools.

L4 Pattern Matching (Low Throughput): With enough available CPU resources the number of rules can be extended to ones which detect attacks with a small impact on the network. These rules include e.g. privilege escalations.

The IDS designed in this work will address the first two levels. These levels are believed to cover attacks with a big impact on the network's congestion. So far no solution for anomaly detection exists that is lightweight enough to be deployed on resource constrained hardware and is able to cope with high traffic loads. The functionality of this system could be combined with the one of Snort in the future. A primitive approach to obtain a system as shown in Figure 3.4 would be to run Snort in parallel with the IDS of this work, but with a low priority. Available CPU resources would first be used to perform anomaly detection.

Chapter 4

Detection Algorithm

The preliminary performance analysis (Chapter 3) demonstrated the need for a lightweight anomaly detection system for wireless mesh networks. This chapter presents the malicious traffic detection metrics, as they will be implemented in a new dedicated lightweight IDS (Chapter 5). The metrics used generally consist of simple host-based counters. Simple counters have the advantage of low memory requirements and low computational complexity.

Anomaly detection is typically used to inform a network operator about a deviation from normal behaviour. Because here an alert results in automated response (Requirement 2), a very low rate of false positives is required (Requirement 3). Rapid changes in traffic patterns have to be expected, especially as we observe single hosts instead of the overall traffic. To still allow the user of a malware infected host to communicate after mitigative actions have been taken, malicious traffic has to be identified in as much detail as possible (Requirement 4). These requisites generally exclude detection metrics based on:

- IP Address Entropy
- Number of flows
- Packet and flow size
- Packet and data rate
- Similar content

The main approach of this work is based on a slightly modified version of the failed-connection approach used by Göldi and Hiestand [20]. It is expected to be robust against false positives and is generic enough to cover a wide range of attacks. The failed-connection algorithm is described in Section 4.1. Section 4.2 presents a protection against flow storms. The spamming detection in Section 4.3 uses a combination of analysing DNS MX queries and SMTP flows. Certain denial-of-service reflection attacks rely on the ability to spoof the source IP address. Others use IP spoofing to obscure the origin of the attack. Section 4.4 discusses how IP spoofing can be defeated. In Section 4.5 metrics to detect oversized echo requests and ICMP floods with packets other than echo requests are discussed. Table 4.1 shows the attack coverage resulting from these

Attack		Anomaly Detection	Pattern Matching
Scanning	TCP	✓	×
	UDP	✓	×
	ICMP	✓	×
DoS	SYN Flood	✓	×
	UDP Flood (random ports)	✓	×[2]
	UDP Flood (fixed ports)	×/ ✓[1]	×[2]
	ICMP Flood	✓	×[2]
	Exploit	×	✓
	Request Flood (L7)	×/ ✓[3]	×
Spamming		✓	×
IP Spoofing		✓	×
Worm Propagation		×[4]	✓
Botnet Communication		×	✓
Privilege Escalation		×	✓
Password Cracking[5]		×	×

1. Indistinguishable from RTP stream if not rejected with ICMP unreachable messages
2. Detectable with tool-specific packet content
3. Detectable if UDP replies fail due to high load, not detectable if TCP connections get established or single connection transmits requests
4. Detectable in combination with scanning
5. Detectable with protocol-based anomaly detection (failed login attempts)

Table 4.1: Detection coverage of attacks with intrusion detection based on anomaly detection and pattern matching (✓ detection generally possible, × not detectable)

metrics and shows where it makes sense to complement the anomaly-based detection with pattern matching. For a description of these attacks please see Section 2.3.

4.1 Failed Connection Attempt Detection

As we saw in Section 2.3, scanning and denial-of-service traffic lead to a high number of failed connection attempts. The detection algorithm for failed connection attempts and the thresholds are based on the work of Göldi and Hiestand [20], but differs in several points. To detect failed connection attempts due to a timeout, it is necessary to track the state of connections. Even if both UDP and ICMP are not connection oriented, a failed connection attempt can be defined for them.

Here a connection is defined as a bidirectional pair of inverse flows. For UDP and TCP this means the two flows have their source and destination IP-address/port-pairs inverted. For ICMP this means they have their source and destination IP addresses and their ICMP-types inverted, but have the same identifier and same ICMP-code. The inverse of an echo request (type 8, code 0)

e.g. is an echo reply (type 0, code 0).

A connection attempt is defined as the initiation of a new connection by sending one or several packets into the original flow direction. A connection attempt succeeds, if one or several packets are returned in the reply direction and these packets do not indicate that the connection has been rejected e.g. by an immediate TCP reset.

A connection attempt fails, if it times out without any reply or if it gets rejected. A rejected connection attempt is indicated by an ICMP unreachable message or additionally for TCP a reset packet with sequence number zero being returned. It has to be noted, that according to this definition a single connection attempt, which involves sending several packets, will be counted as several failed connection attempts, if the packets are rejected with multiple ICMP unreachable messages or TCP reset packets. This is caused by counting connection attempts based on flows, and connection failures caused by rejects based on packets. The advantage of this definition is that, if a SYN packet is retransmitted and the connection finally succeeds, the first SYN packet is not counted as a failed connection attempt. This increases robustness against false positives. Counting rejected connection attempts based on packets on the other side improves detection speed and softens the requirements for the connection tracking.

4.1.1 TCP

Figure 4.1 shows the failed connection attempt algorithm for TCP. Each observed host is in one of the three TCP failed connection states: DST_NONE, DST_HOST or DST_PORT. Generally a host is in DST_NONE, which means failed connection attempts are simply counted. This stage is most efficient in terms of memory and computational complexity.

If a host has 50 failed connection attempts within a minute, a more detailed check is started. In state DST_HOST the destinations of failed connection attempts are analysed. If more than 400 connection attempts fail for a single destination, it is assumed that this destination is under DoS attack from the observed host. An alert is raised and as both the source and the destination are known, fine grained mitigative action can be taken. As the same host might attack several destinations, it still remains in state DST_HOST. The counters for the destination, which already caused an alert are reset.

If a host is in TCP failed connection state DST_HOST and shows failed connection attempts to more than 100 distinct destinations, it enters state DST_PORT. Such a host might be scanning for vulnerable services and the algorithm tries to identify the services it is scanning. In DST_PORT the number of destination IP addresses is counted for every destination port which shows failed connection attempts. If for a single destination port connection attempts fail for more than 100 distinct destination addresses, this port is identified as being scanned by the source host and responsive action can be taken. Again the counters for this port are reset and the host remains in state DST_PORT, because it might be scanning for several destination ports. Five minutes after a host entered a detail check state DST_HOST or DST_PORT, it is reset to state DST_NONE.

The detection algorithm presented here differs in various points from its original version [20]. Here also ICMP unreachable messages are used to count

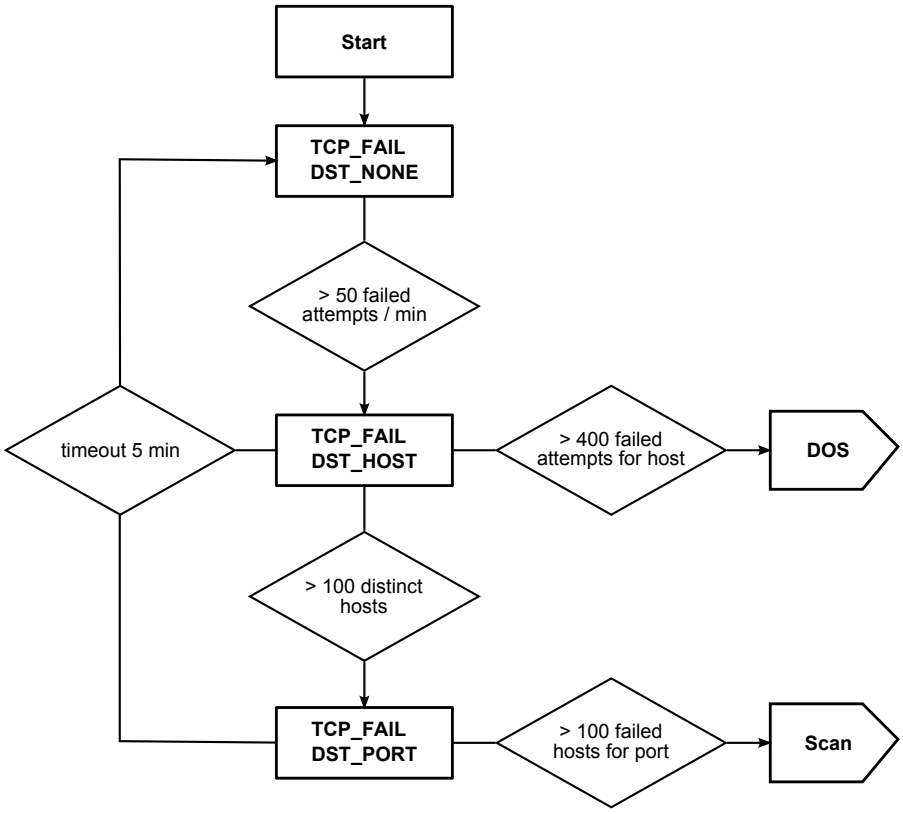


Figure 4.1: Detection algorithm for failed TCP connection attempts

failed TCP connection attempts. Scanning a random target can lead to ICMP unreachable messages being returned, if the target is not routable. By counting unreachable messages, such failed connection attempts can be counted without having to wait for a timeout. The original version of the algorithm starts a detailed analysis, if for a host more than 100 failed connection attempts are counted in two minutes. Here the threshold is 50 failed attempts in one minute. This shorter observation period increases detection speed, but might more likely start a detail check on bursts. As starting detail checks does not lead to immediate remediation, this is not too dangerous. It has to be shown in later evaluation (Section 6.3), if the lower threshold is acceptable.

In the original version of the algorithm two more states have been defined. Instead of checking for DoS attacks in DST_HOST, the original version waits until the timeout occurs and then changes the state to one of the here unused ones to start a fresh analysis of failed destinations. Doing so delays detection of DoS attacks. Additionally the original detection of denial-of-service attacks requires that at least 1200 connection attempts fail up to at most four distinct destinations. If a potentially small number of failed connection attempts occurs for a slightly higher number of hosts, the state is changed to DST_BENIGN and the detection aborted. This makes false negatives more likely.

The second state, that is not used here, is entered, after a host has been in state DST_PORT for five minutes. In this state the original version of the algorithm counts the number of destinations with failed connection attempts independent of the destination ports. If more than 300 distinct destinations cause failed connection attempts within 5 minutes, the source host is flagged as malicious. As this kind of detection does not give fine-grained information about the targets or the destination ports, it is unsuitable for a system which has to respond automatically.

Another big difference between the algorithm presented here and its original version is that the original version has been designed to inform a network operator about a misbehaving host in the network. As soon as it detects a violation, an alert is raised and the host causing the violation is excluded from further analysis for 24 hours. Here automated response has to be taken and it is possible, that a host is causing several violations which have to be mitigated individually. It is therefore not possible to exclude a host from analysis, after a first alert has been triggered.

Assume a SYN-flood attack with fixed source and destination ports, i.e. all packets belong to the same connection. As long as the connection does not timeout and no TCP reset packets are returned, no failed connection attempt can be counted. To prevent this kind of detection evasion, the TCP SYN tuple check is introduced. This check is illustrated in Figure 4.2. Each observed host is in one of the two TCP SYN tuple states: DST_NONE or DST_TUPLE. Generally a host is in DST_NONE. In this state the number of TCP connection attempts and the number of SYN packets are counted (see section 4.1 for the difference between these two numbers).

If within a minute the difference between the number of SYN packets and the number of connection attempts is greater than 50, the sending host enters state DST_TUPLE. In this state the number of SYN packets per flow or hash tuple are counted. If for a single flow more than 100 SYN packets are sent within 5 minutes, this is identified as a denial-of-service attack.

The original version of the failed connection algorithm is also vulnerable to

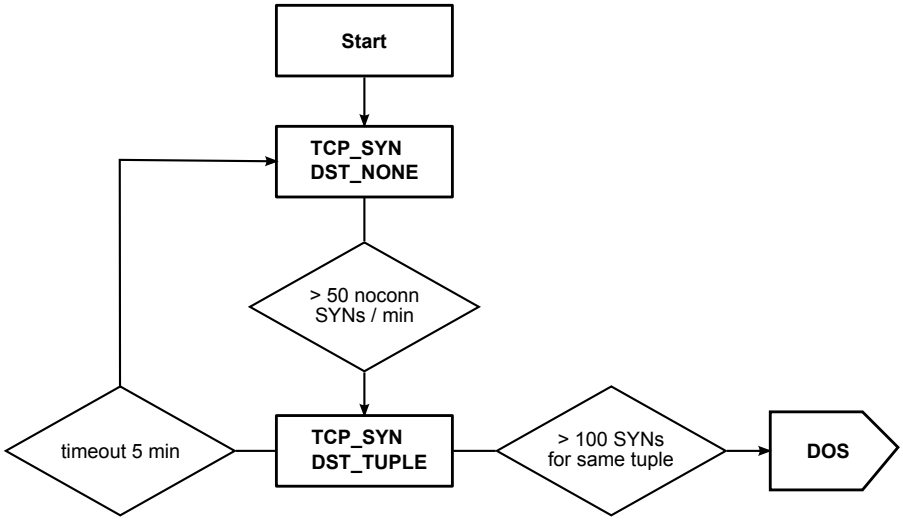


Figure 4.2: Detection algorithm for TCP SYN floods using the same tuple

this kind of detecion evasion, but does not provide any countermeasures.

4.1.2 UDP

The detection of failed connection attempts for UDP is identical to the detection for TCP which has been described before and which is illustrated in Figure 4.1. Connection timeouts without replies and ICMP unreachable message are counted as failed connection attempts for UDP. In Section 2.3.7 we have seen that RTP streams are unidirectional. As an RTP connection does not have packets in the reply direction, it will be counted as a failed connection attempt, as soon as the connection times-out. This only leads to a low number of failed connection attempts, which does not trigger any state transitions. The same principle could be used by a denial-of-service flood. Such a flood with fixed source and destination ports would remain undetected. Here it is not possible to distinguish between benign and malicious and detecting a UDP flood with fixed ports is left open for future work or the pattern-matching intrusion detection. No similar detection mechanism to the TCP SYN tuple detection illustrated in Figure 4.2 will be implemented for UDP.

The original version of the UDP failed connection detection algorithm [20] is slightly different. The main difference is that it has two substates. The structure of both substates is similar to the original TCP detection algorithm described in Section 4.1.1 but they fullfill different tasks. In one substate only ICMP unreachable messages are analysed. The other substate counts every UDP packet of a connection without reply as a failed connection attempt. It is obvious that this causes false positives for RTP streams. The distinction is made to be safer against false positives, if the enterprise firewall rejects blocked UDP packets with a ICMP unreachable message. If more than 20 ICMP unreachable messages are counted, the ICMP unreachable substate is used to analyse the behaviour of a host. If the firewall silently drops blocked UDP packets and less than 20 ICMP unreachable messages are counted, the second substate is used, which

is more error prone. Here the definition of a failed UDP connection attempt is different, and failed connection attempts by timeout and ICMP unreachable messages are treated equally.

4.1.3 ICMP

Echo requests are not essential for communication on the Internet. Because of this non-essential manner, an alert is triggered as soon as more than 200 echo requests per minute fail. No detail check is started to check if failed connection attempts occur to a high number of distinct hosts or to single destinations at a high rate. This allows faster detection and detection of DoS floods addressing a bigger number of addresses to flood a victim's network. On the other side it might increase the probability of false positives and leads to higher granularity of mitigative actions than to prevent DoS attacks against single targets. This relaxation is accepted, because communication of an alert triggering host is not severely affected (Requirement 4).

Several unreplied echo requests with the same identifier lead to a single timeout, as they are part of the same connection. To detect DoS floods with echo requests with identical identifiers, the number of echo requests and responses are counted as well. If the difference between requests and replies is bigger than 220 per minute, an alert is triggered. This value is by 20 higher than the threshold for failed connection attempts to have a buffer for currently unreplied requests.

The ping utility on Windows and Linux send one echo request per second by default. With 200 failed requests allowed per minute it is possible to ping three unreachable destinations at the same time without triggering an alert. The threshold to detect a denial-of-service attack against a single destination in the original algorithm is 300 failed requests per minute on average. If necessary, the threshold here can be raised to the same or even higher value.

4.2 Flow Storm Protection

In Section 3.2 it has been observed that denial-of-service floods with a high number of new flows can cause high CPU load due to connection tracking. This results in decreased forwarding capacity and higher round-trip times. If the kernel uses a lot of CPU resources, user space applications get less runtime. In this case a high number of packets cannot be analysed by the intrusion detection system which slows down the detection or makes it infeasible. To assure proper operation in the presence of such kinds of attacks, a protection against flow storms will be implemented.

During a flow storm which impacts upon the performance of the router, the CPU usage is very high and packet drops occur for capturing. These two metrics are used to activate the flow storm protection. Hosts which initiate a high number of new connection have to be considered the source of the storm. As not all packets can be analysed, the real number of connection attempts is unknown and the real source therefore uncertain. To reduce the load on the router, the number of connection attempts of a host with a high number of connection attempts can be limited to an acceptable level. Because of packet drops during capturing the limit will be higher than the threshold to detect the storm flow. Assume for example, a host initiates 1000 connection attempts

per second which overloads the router. The intrusion detection system is only able to detect 50 connection attempts per second which is enough to reach the threshold. The host is then limited to initiate at most 200 connections per second, which reduces the load. The actual thresholds and limitation rate will be defined in Section 6.1.3. To prevent the packets from being analysed by the kernel's connection tracking, the rate limitation has to be done in the Netfilter raw table. This table does not allow to identify connection attempts, so for TCP the number of SYN packets and for UDP the overall number of packets have to be limited. As the limitation rate will be quite high, it should not affect hosts for which rate limitation has been activated due to a false positive. Because the quality of service is not reduced, false positives can be tolerated here (Requirement 3).

4.3 Spam Detection

Spam sending malware which directly delivers emails to target mailservers will first query the DNS for the IP addres of a mail exchange server and will then establish an SMTP connection to this host. To be more robust against false positives caused by people analysing the MX configuration of certain domains, the threshold to detect an MX-querying spam agent not only includes the number of MX queries but also the number of SMTP connection attempts. If during one minute a host sends more than 5 DNS MX queries and more than 5 SYN packets to SMTP port 25, it is identified as a spammer.

For the case that a spam agent is not sending any DNS queries, e.g. by using the user's ISP's mail server to send mails or by retrieving the address for the mail exchange server through a different channel, the number of connection attempts to an SMTP port is analysed on its own. As some mail servers use TCP port 587 or 465 for submission by clients, these ports are included in detection as well. If a host initiates more than 100 SMTP connections per minute, it is identified as a spammer. A user will not send more than 100 emails in a minute. Sending an email with multiple recipients only creates a single connection in general. If however an email is sent encrypted, it will be sent for every recipient individually. Sending an email with more than 100 recipients using encryption will trigger an alert. Because of the low spread of email encryption, this is not considered to be a problem. A user is not able to send 100 emails per minute, but a computer program can do this easily. Because of lower network quality in terms of stability and throughput and potential NAT and packet filters, it is unlikely that mail servers are deployed in a wireless mesh network. Also the possibility that a user automatically generates individual emails on his computer, e.g. for a newsletter, is rather uncommon. The use of a detection metric based on the number of flows is here considered to be robust against false positives.

As it was discussed in Section 2.3.4, it is possible to use a single connection to send a potentially high number of emails. In this case, counting the number of SMTP connections does not help to detect the anomaly. The detection of this way of spamming is based on the data volume sent email ports. To avoid false positives by clients sending very big emails the threshold has to be sufficiently high. Sending more than 50'000 KiB in 15 min to an SMTP port is considered as malicious here. This corresponds to sending with more than 455 kbps on average

during 15 min. Analysing the data volume over a short period such as a minute is not possible, as benign bursts would likely cause false positives. If in the same 15 min an MX query is observed, the threshold is lowered to 10'000 KiB, as an MX query is a stronger sign for malicious activity.

Compared to the failed connection attempt analysis, it has to be noticed, that the spam detection is less dynamic. Very active benign clients will not necessarily have more failed connection attempts, whereas very active mail senders are at risk to cause false positives. The detection thresholds have therefore to be carefully adjusted.

4.4 IP Spoofing Detection

Spoofing the source IP address helps to obscure the origin of denial-of-service attacks which do not require a reply from the target and is required for different reflection attacks. Defeating IP spoofing is based on two parts. First it is important to block packets with impossible source addresses. This means, for example, that packets coming in on the client wireless interface have to originate from that client subnet or that packets being routed in the mesh network have to have either source or destination IP address in the mesh subnet. The second part to defeat IP spoofing is to detect clients which use more than one IP address within the legal address range.

Detection of IP spoofing is based on counting the number of IP addresses being used by a single network interface card identified by its MAC address. This is only possible, if the real MAC source address can be observed. As soon as a packet gets forwarded by a router, this is not possible anymore. IP spoofing detection has therefore to be done on the first router after the packet gets sent by a client. As a client could receive a new IP address over DHCP after it released its old lease or as a client could be changing its IP address manually, detection has to tolerate a low number of IP addresses being used by a single network interface card. Here spoofing is detected, if a client uses more than three IP addresses within 5 minutes. IP spoofing detection relies on clients not changing their MAC address. Changing the MAC address of a network interface card is possible and would subvert detection. Because packets with impossible source addresses will be blocked, changing the MAC address only allows to spoof IP addresses from a limited range. This makes certain reflection attacks against targets on the Internet impossible, e.g. the Smurf attack.

4.5 Other Metrics

Echo requests sent by the ping tool have a small packet size by default. For Windows the length of the IP packet is 60 B and for Linux 84 B. To saturate a target's network with ICMP echo requests, a high number of packets are sent and they potentially have a large packet size. If the target does not reply to these requests, this can be detected with the failed connection detection. As long as it does reply, the flood remains undetected. To improve detection, a host sending more than 62 echo requests larger than 256 B in a minute is flagged as sending a ping flood of large size. Large sized echo requests can be used to analyse *Maximum Transmission Unit (MTU)* problems. The thresholds used

here should be big enough to still allow this kind of analysis without triggering an alert.

A generic ICMP flood is detected if a host is sending more than 3000 ICMP packets or 300 KiB per minute (50 pps or 41 kbps). This helps to protect against a host sending other ICMP packets than echo requests in a DoS flood. Sending ICMP unreachable messages e.g. makes detection and filtering difficult, if the messages relate to an existing connection. Sending ICMP messages is typically not essential for a network clients connectivity. Mitigation affecting any ICMP traffic originating from a malicious host is acceptable.

Chapter 5

Implementation

This chapter describes the implementation of a **L**ightweight **I**ntrusion **D**etection **S**ystem dedicated to detecting malicious traffic in wireless mesh networks. The system is named *OpenLIDS*, and its anomaly-based detection algorithms were described in Chapter 4. Figure 5.1 gives an overview of OpenLIDS and its components. The following sections discuss them in detail.

5.1 Miscellaneous

OpenLIDS is implemented in C and is released under GPL. It runs as two single-threaded processes which use a domain socket for inter-process communication. One process is responsible for most tasks: packet capturing, connection tracking and the detection of malicious traffic. For security reasons, this process drops privileges to the ones of an unprivileged user after initialisation. Even if OpenLIDS has been designed with care, vulnerabilities in OpenLIDS or the used libaries cannot be excluded. Running with reduced privileges limits the impact of a potential exploit. The second process is responsible for remediative actions. If malicious traffic is detected, it applies new packet filter rules. As this requires root privileges, this process does not drop privileges.

OpenLIDS can be configured using a configuration file. It is possible to set a wide range of program parameters, adjust the detection thresholds and configure input methods. Log messages are generated for different log levels and it is possible to log to standard output, a logfile or syslog.

All data structures are IPv6 enabled, but IPv6-specific functions for packet parsing and changing packet filter rules have not been implemented yet. Parts in the source code which need work to fully support IPv6 have been labelled for easy integration in the future.

5.2 Packet Capturing and Decoding

OpenLIDS supports NFLOG and ULOG as packet capturing methods. In Section 3.3 ULOG performed the best in respect of CPU utilisation and did not have any packet losses. It was not possible to test NFLOG due to portability issues, but its performance is not expected to be much better than ULOG. Support for NFLOG is built-in, but was not evaluated on the WG302.

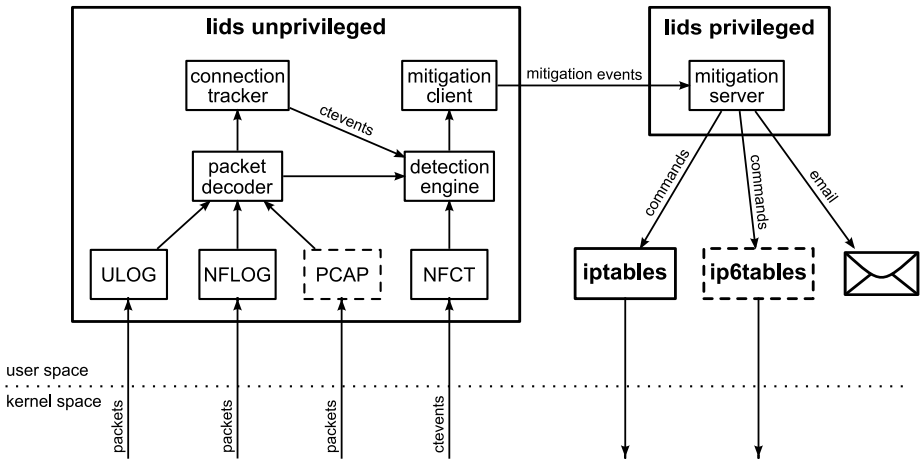


Figure 5.1: OpenLIDS Design Overview. A two process architecture is used whereas the main process runs with reduced privileges. OpenLIDS provides Netfilter ULOG and NFLOG as packet capturing mechanisms and one can chose to track connections in users space or use the connection tracking system of the Linux kernel with the NFCT module. Dashed modules are unimplemented.

Besides good performance, ULOG and NFLOG have the advantage, that IP defragmentation and checksum validation on network and transport layer are already performed by the Netfilter system. Defragmentation with small fragment sizes could be used to obscure transport protocol headers or DNS queries. Checksum validation helps to avoid the analysis of corrupt packets. Because packets are captured at the network layer, no special measures are required to eliminate duplicate packets.

OpenLIDS is mainly driven by incoming packets. Each input module registers the sockets it wants to read from at a central dispatcher. As long as no activity occurs, the application sleeps in a `poll()` call. If a socket becomes readable, the dispatcher calls the handler of that particular socket, which then can read data from it. Both the NFLOG and the ULOG module read multipart messages from the Netlink socket. They split them into single packets and hand the IP packet and the source MAC address to the packet decoder. The packet decoder parses the packet and fills a packet structure with the required information. Packets then get analysed by the detection engine.

The packet decoder parses the packets to obtain protocol specific parameters. It is byte-ordering and byte-alignment aware. Packets are verified to have the required length before each decoding step. Protocol information is filled in a packet structure. This structure contains the source MAC address, if it is available. Further it contains the following information decoded from the particular protocols:

- IP: protocol version, source and destination address, packet length
- TCP: source and destination port, control flags, sequence number
- UDP: source and destination port

- ICMP: type, code, identifier of request and reply messages, inner IP packet of error messages
- DNS: query type

To get all the relevant information, the capture length has to be sufficiently large. An unnecessarily large capture size reduces performance. Generally only packet headers are captured. Regarding the minimum required capturing length, ICMP unreachable messages constitute a worst case. These packets contain also parts of the original packet which caused the error. The headers of this original packet need to be decoded as well. For IPv4 the minimum capture length is 92 B. In every case this allows to capture the IP header (20 B), the ICMP header (8 B), the IP header of the original packet including potential options (60 B) and source and destination ports for UDP and TCP (4 B). It has to be noted that this length differs from the tcpdump snaplen because the snaplen also includes ethernet headers. 92 B is used as capture length for general packets here.

DNS queries potentially have several question sections [47]. Each question consists of a domain name followed by a query type and a query class. To count the number of MX queries, the query type is of interest. As it is rather at the end of each question, DNS queries have to be captured at their full length. This is not a problem, as NFLOG and ULOG allow to specify different capture lengths for different capture rules.

Due to the modular approach of OpenLIDS it would be easy to integrate a module to capture packets with libpcap. Disadvantages of libpcap are that it does not support multipart messages or variant capture lengths. Multipart messages reduce the need for context switches and variant capture length is required to capture DNS queries at their full length while only capturing headers for other packets. Further IP defragmentation would have to be implemented, if protection against defragmentation evasion attacks is desired.

5.3 Connection Tracking

To count connection attempts which fail with a timeout, it is necessary to track the state of connections. In Section 3.2 it has been shown that connection tracking is computationally expensive in the presence of attacks with a high rate of new connections. This section describes the implementation of two connection tracking approaches: using the connection tracking of the Netfilter system and tracking connections in the application itself. They are mutually exclusive and one can choose to use either one or the other.

5.3.1 Kernel Space Connection Tracking

The Netfilter system of the Linux kernel is used to track connections for dynamic NAT and stateful packet inspection. Using the same connection tracking system for intrusion detection is promising, as it avoids having to do the same work twice. Access to the connection tracking system in the kernel is provided over Netlink. It is possible to query the connection table, modify it or get informed about state changes by so called *connection tracking events (ctevents)*. Ctevents are caused by new connections, if a connection state changes e.g. due to packets in reply direction, or if a connection is destroyed. Here only new connection

and destroyed connection events are of interest. In a similar way to NFLOG packet logging, the NETLINK_NETFILTER family of the PF_NETLINK socket is used. For connection tracking just a different Netlink subsystem is used. A user space interface to the Netfilter connection tracking is provided by libnetfilter_conntrack [50]. This library depends on libnfnetlink [52].

Ctevents include a status field and packet and byte counters for both flow directions. The status bit `IPS_SEEN_REPLY` is set, if packets in the connection's reply direction have been received. If a connection destroy event does not have the `IPS_SEEN_REPLY` status bit set or has the counters in reply direction set to zero, this is a connection attempt which potentially failed with a timeout. Both means that no packets have been received in the connection's reply direction. Such a connection could also have failed with an ICMP unreachable message. Distinguishing these two cases is discussed later.

During the analysis of the connection tracking event facility, two bugs in the Linux kernel's connection tracking system have been discovered:

- On a destroy event the conntrack status field did not get copied to the connection tracking event. This prevented determining the status of a connection on its destruction. Evaluating the IPS_SEEN_REPLY flag was therefore not possible.

- On abnormal termination of a connection, the last packet was not used to increase the packet and byte counters. An abnormal termination is e.g. caused by a TCP reset packet rejecting a connection attempt or a reply to an ICMP echo request. It was not possible to identify an unreplied connection by examining the packet counters in the destroy event.

The combination of these two bugs made it impossible to identify connection attempts, which failed with a timeout. I fixed both bugs and my patches have been integrated in the Linux mainline kernel with version 2.6.27-rc1[1].

A destroy event with no packets in the reply direction indicates a connection attempt which failed either with a timeout or with an ICMP unreachable message. It has been defined that each ICMP unreachable message is counted as a failed connection attempt. This allows better accuracy e.g. to detect a UDP denial-of-service flood which uses fixed ports and gets rejected with ICMP unreachable messages. This attack causes only one destroyed connection event but several failed connection attempts due to multiple ICMP unreachable messages. The destroyed connection event of a connection which was rejected with ICMP unreachable messages should not be counted. To only count a destroy event of a connection without reply packets as a failed connection attempt if this connection did not fail with an ICMP unreachable message, a kernel patch has been written. This patch introduces a new conntrack status bit `IPS_SEEN_RELATED`. This bit gets set for a connection, if another was related to it. Related connections have different hash tuples but get associated to each other by a helper. This is e.g. the case for an ICMP unreachable message being related to a connection or an FTP data connections being related to an FTP control connection.

[1] Netfilter kernel bugfixes:
`http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=e57dce60c7478fdeeb9a1ebd311261ec901afe4d`
`http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=718d4ad98e272daebc258e49dc02f52a6a8de9d3`

Protocol [State]	Timeout
TCP SYN SENT	120 s
TCP SYN RCVD	60 s
TCP ESTAB	3600 s
TCP FIN WAIT	120 s
TCP CLOSING	10 s
TCP TIME WAIT	120 s
TCP CLOSE WAIT	60 s
TCP LAST ACK	30 s
UDP	60 s
UDP replied	180 s
ICMP	30 s

Table 5.1: Default connection tracking timeouts

This kernel patch will not be part of the mainline kernel due to its limited use and minor modifications to a hot path. Applying kernel patches for OpenWRT does not impose any problems as it is easily supported by the toolchain and the software for a wireless mesh network is likely to be a custom build anyway. If the Netfilter connection tracking should be used, but applying the kernel patch is not an option, it is possible to disable counting failed connection attempts based on ICMP unreachable messages. Instead these failed connection attempts will be counted based on the timeout as soon as the connection destroy event is received. This behaviour can be enabled by setting the `count_unreachable` configuration option of OpenLIDS to zero. The disadvantages of this are higher detection delays and the inability to detect UDP floods with fixed ports.

A problem with using the kernel connection tracking system is that the timeouts are adjusted not to break open connections. This is important for dynamic NAT and stateful packet inspection. If a connection is removed from the connection table, packets from that connection that arrive later might be dropped. Table 5.1 shows the default connection timeout values on OpenWRT with kernel 2.6.24.2. Important timeouts to detect failed connection attempts are TCP SYN SENT, UDP and ICMP. The TCP SYN SENT timeout value applies for SYN packets being sent as long as no reply packets are received. It is set to 120 s. This causes a long delay until a connection attempt timeout can be detected. It is believed to be safe to reduce this timeout to 30 s: Retransmissions of initial SYN packets create a new connection entry, if it was removed already. The smaller timeout only affects the replying SYN/ACK packet which should easily arrive within 30 seconds. Reducing the UDP timeout e.g. to 30 s could be considered as well, but this is more dangerous. A request that gets cached or causes a complex process on the server might take longer than 30 s to be answered. A shorter UDP timeout is therefore not recommended. The ICMP timeout of 30 s is acceptably low. The timeout values can be set at boot time by modifying `/etc/sysctl.conf`. To reduce the TCP SYN SENT timeout to 30 s one would place a line like this:

```
net.netfilter.nf_conntrack_tcp_timeout_syn_sent=30
```

For a first performance test, a small program has been written. It opens a Netlink socket and registers to receive new and destroyed connection events.

The program then reads the socket and discards all data it receives. Using the same test procedure as in Chapter 3, UDP packets with a payload size of 1000 B and random source ports were sent at maximum rate. Routing this kind of network traffic caused 8 % CPU load in earlier tests. Running the ctevent test tool caused 45 % CPU utilisation. Even if the CPU resources were not fully exhausted, 33 % of all connection events were dropped. The default kernel side socket buffer size in this test was 111 KB. Increasing the buffer size to 2 MB prevented any event drops, but increased the CPU usage to 54 %. With the bigger buffer size it is no problems if the application does not get enough runtime during bursts. The CPU usage was higher because now all events could be sent to user space.

Using the connection tracking of the kernel obviously comes with large costs to transfer the ctevents from kernel to user space. As the events are read from the socket one after the other, a high number of context switches are necessary. Queueing ctevents and sending several to user space in a single multipart message certainly could reduce the load. I discussed this idea on the Netfilter mailing list. The maintainers agreed on the resulting performance improvements, but required that multipart messages would have to be enabled for each socket independently. Like this, receivers which do not support multipart messages could coexist with ones that do at the same time. The ctevent Netlink messages are composed before they get broadcast to all active receivers. Per receiver multipart messages were therefore considered infeasible under the given requirements.

To find other possibilities for performance improvements, the kernel has been profiled with Oprofile [58] during the ctevent read test. It turned out that the system spent almost 10 % of its runtime in `memcpy()`. A big part of these calls were caused by preparing Netlink messages to be broadcast to their recipients. The Netfilter developers spotted the message trimming as a likely cause. Netlink messages are allocated with a fixed and generous size, which is certainly enough to hold all necessary information. The message is then composed. If less than 50 % of the effective space is used, the message gets reallocated with the real size and the data copied. A first rudimentary patch which allocated a much smaller initial message size could could successfully reduce the CPU load. The Netfilter developers provided a final version of the patch, which is now queued for kernel version 2.6.28. The final version calculates the amount of memory which is actually required and uses this value to allocate memory for the message. Profiling also showed that the Netlink connection event path contains a high number of unnecessary calls to a reference counting unit. Objects were retrieved while decrementing the objects reference counter, were analysed and then stored again while incrementing the object's reference counter again. Both retrieving and storing had to be protected with a read lock. A first version was written for a patch, which locks once at the beginning and then just gets the objects without changing the reference counter. This decreased the number of lock operations and the objects did not have to be stored again. This patch has been further improved by the Netfilter maintainers and is now queued for kernel version 2.6.28.

With both connection tracking event performance patches applied, the CPU usage dropped from 54 % to 49 %. Sending the ctevents to user space causes still a big overhead. Future experiments will be carried out with the patches applied and it has to be shown how the performance for kernel connection tracking

compares to tracking connections in user space.

5.3.2 User Space Connection Tracking

Using the connection tracking of the kernel turned out to be not as cheap as expected because the connection tracking events have to be sent from kernel to user space. Packets are available in user space anyway as they have to be captured for detection analysis. Connection tracking in user space is therefore considered as an alternative. Tracking connections in user space brings the overhead of hash table lookups and memory allocations for new connections. The advantage of the user space approach is that no kernel patch is required and that timeout values can be freely adjusted.

The implementation of the connection tracking module in OpenLIDS is kept simple. Similar to the Netfilter connection tracking, each connection has two hash tuples to represent its flows in both directions. Each hash tuple contains source and destination IP addresses, for TCP and UDP source and destination ports and for ICMP type, code and identifier. Each connection has a status field, a timeout value and a list entry node for a timeout list. TCP connection states are not maintained.

All hash tables in OpenLIDS use the implementation of the Jenkins hash function [27] from the Linux kernel. Calculating hash values depends on a random seed. This makes DoS attacks difficult which aim to overfill a single hash table bucket. The number of buckets in a hash table is always a power of two. Converting a hash value to a bucket index can be done by applying a simple bitmask. No modulo computations have to be performed.

After packets have been decoded, they get handed to the connection tracking module. First the packet gets converted to a hash tuple. This tuple is used for a hash table lookup. If no equal tuple already exists, a new connection structure is allocated and stored in the hash table and a timeout list. In this case a new connection event is generated which is delivered to the detection engine. If in the hash table an equal hash tuple already exists, this its connection is retrieved and the connection timeout is updated. If the packet was in the connection's reply direction status flag `FS_SEEN_REPLY` is set.

Timeout handling is implemented in a very simple and efficient way. Only two types of timeouts exist: short timeouts occur after 30 s and long timeouts after 5 min. Every connection is in a sorted linked timeout list. Each timeout type has its dedicated list. So far unreplied connections and ICMP connections have a short timeout, whereas replied connections have a long timeout. TCP connections for which a FIN or RST packet has been observed, have a short timeout as well. These connections are marked as terminating by setting the connection status flag `FS_TERMINATING`. Every packet refreshes the connection timeout. For this the connection is removed from its timeout list and added to the end of the timeout list corresponding to its status. The linked list implementation from the Linux kernel is used. This list allows insertion and removal of elements in constant time. In this way, the timeout lists are always sorted at negligible costs. Every second the connection tracking module checks for expired connections. For this the expiry time of connections at the head of both linked lists are compared to the current time. Because the lists are sorted, checking for timeouts is cheap. If an expired connection is found, a destroyed connection event is generated and delivered to the detection engine. The status

field is included, such that it can be decided, if the event has to be counted as a failed connection attempt.

ICMP error messages are treated slightly different. They always relate to another connection and are of singular use. No connection is allocated for them. ICMP messages are examined before the hash table lookup occurs. For ICMP error messages, the error causing inner IP packet is used to lookup the connection hash table. If a connection is found, its `FS_SEEN_RELATED` status flag is set. Analysis of ICMP error messages is terminated at this point. ICMP request and reply messages (echo, timestamp, information, address mask) are still tracked and they get processed like other packets.

In the presence of attacks with a high rate of new connections, the connection tracking poses the risk to use up all available memory. To prevent this, the number of tracked connections is limited. The maximum number of tracked connections is by default 10'000. As each connection uses 144 B the memory consumption of all connections is limited to 1.44 MB. If the maximum number of connections is reached and a new connection should be created, it is tried to destroy an existing connection and replace it with the new one. A destroy event is generated and delivered to the detection engine in this case. First, it is checked if there is a connection which already expired. If none expired, it is checked if there is a connection which will expire in less than half of its timeout period. Halving the timeouts increases the maximum number of newly tracked connections per second and still gives a reasonable time for replies to arrive. If no connection entry can be replaced, no new connection structure is allocated.

5.4 Anomaly Detection

The anomaly detection module receives packets and connection tracking events for analysis. Each observed host has a host entry structure which contains the simple counters required for the detection algorithms. On reception of a packet or an event, these counters are incremented. No violation checks are performed at this stage. Checking for violations is cheap, but doing so on every received packet creates an unnecessary overhead. Instead violation checks are performed periodically every 10 seconds. The basic counters are reset every 60 seconds.

Hash tables lookups to the host table and potential host structure allocations are not done for every packet or connection event. A lookup is only performed if a host:

- Initiates a new connection (TCP, UDP and ICMP)
- Suffers from a failed connection attempt by a timeout
- Sends TCP SYN packets with the ACK bit cleared
- Sends to TCP ports 25, 465 or 587 (outgoing email)
- Sends DNS queries
- Sends UDP packets and spoofing protection is enabled for that host
- Sends ICMP packets
- Receives an ICMP destination unreachable message

- Receives a TCP RST packet with sequence number zero
- Receives an echo reply

This covers all network clients in or outside the mesh network. Network servers which only passively serve requests are not monitored unless they send ICMP messages. Observing all clients is necessary for the flow storm protection and the SYN tuple check. With the OpenLIDS configuration option `observe_net` it is possible to restrict the observed hosts to be from certain subnets. This particularly makes sense, if every mesh box runs OpenLIDS and each box only observes its client network and not the whole wireless mesh network range or hosts outside of the WMN. To protect the host entries from using up all available memory the maximum number of hosts in the host table can be limited. With a size of 204 B per host structure including counters observing 10'000 hosts will use 2 MB of RAM.

For IP spoofing detection, direct communication between the sender and the router running OpenLIDS is required. No intermediate routers are allowed. This makes it necessary to specify the networks which are directly connected to the router. Spoofing detection can be enabled for a network with the OpenLIDS configuration option `observe_mac`. All host entries are stored in a hash table. For networks without spoofing detection, host entries are identified by their IP address only. For networks with spoofing detection enabled, a host has one *hard host entry* and potentially multiple *IP host entries*. The hard host entry contains the host's counters and is used to lookup the host by its MAC address. This is only the case, if the host has to be looked up because it was the source of a packet. The IP host entries are used to lookup the host by an IP address as it is done for connection tracking events or packets which are sent to this host. An IP host entry does not contain any detection data but a pointer to the corresponding hard host entry. The hard host entry is then used to count packets and events. For every new IP address a host uses, a new IP host entry is created. All IP host entries are stored in a linked list of their hard host. During the violation checks, the number of IP host entries of a hard host is counted and compared to the threshold. IPv4 and IPv6 addresses are counted independently. Every host entry contains the time it has last been used in a lookup. Host entries expire after not being used for more than 5 min. This is true for IP host as well as hard host entries.

Apart from simple counting, more detailed analysis can be activated for a host as explained in Section 4.1. Detail checks for TCP and UDP failed connection attempts and TCP SYN floods using the same connection are independent and can potentially be active simultaneously for a single host. The number of simultaneous detail checks can be limited to prevent memory exhaustion. The maximum number of simultaneous detail checks is 10 by default. If a new detail check is to be started and 10 are already active, this check is postponed until another check finishes.

For hosts in TCP and UDP failed connection state DST_HOST a hash table is allocated, which contains the IP addresses that failed, including a counter for how many times a connection attempt to this address failed. In state DST_PORT, where the number of failed connection attempts is evaluated per destination port, two hash tables are allocated. One uses the destination IP address and the destination port as key and counts the number of failed connection attempts to this destination. The second hash table counts the number of

distinct destinations which failed for a specific port. If for one IP-address-port pair a connection fails for the first time, the counter of this port is increased. To detect SYN flood attacks which use the same connection, a hash table with the destination IP address, source port and destination port as key values is used. Hash table entries related to a detected violation are cleared as soon as the alert is triggered.

5.5 Mitigation

As soon as a violation, as defined in Chapter 4, is detected, the mitigation client sends a mitigation event to the mitigation server in the privileged process. To enable fast activation of mitigative actions, the privileged process runs with higher scheduler priorities (`nice(-1)`). The mitigation event contains only a minimal amount of information. Only the type of the violation, the address of the host causing the violation and where necessary the destination IP address, transport layer protocol and destination port are sent. The mitigation server then decides on how to respond to the violation. Currently the default action is to silently drop traffic sent from the violating host in a fine-grained way. For the different mitigation events this means to block:

- Scanning: Traffic to the identified destination port
- DoS: Traffic to the identified destination address
- Spam: Traffic to TCP ports 25, 465 and 587
- IP Spoofing: All traffic
- Ping: All echo requests
- Large Ping: All echo requests bigger than 256 B
- ICMP: All ICMP traffic

For a host causing a high load by initiating new connections at a high rate, the response is to limit the number of SYN packets or UDP packets this host can send per second. Except in the case of IP spoofing, the remediative action has no severe effect on the clients abilities to communicate. If a host however is scanning targets for highly common services such as DNS or HTTP and communication to these ports is blocked, the effects of remediation are severe. Limiting the maximum available bandwidth or packet rate would the client still allow to communicate. The limits would have to be chosen aggressively enough such that the malicious traffic is reduced to an acceptable level (Requirement 2), but loose enough such that benign traffic from the throttled host gets enough network resources (Requirement 4). Finding good rate limitation values for throttling is difficult, because preventing congestion highly depends on the network's current load, the channel quality and the network's capacity, which varies with the number of hops to the next Internet uplink. Choosing small values will prevent congestion, but will reduce the quality of service for the client, especially because the provided bandwidth will likely be used by the malicious traffic. Because of these problems bandwidth limitation has not been implemented in OpenLIDS. Traffic shaping support for *Tc* [24] could be integrated in future versions. Packet

rate limitations are supported with Iptables and are already used to mitigate flow storms.

Packet filters are updated over a pipe to *Iptables-Restore*. Rules are placed in a user configurable Netfilter chain. The user has to create this chain and include it in the packet filtering workflow. The chain is flushed on start and termination of OpenLIDS. To prevent the host from stressing the router's Netfilter connection tracking even if the traffic is blocked, the rules have to be placed in the Netfilter raw table. For hosts which are identified by their MAC address, the filter rules are based on the source MAC address rather than the source IP address. They stay valid, even if the host changes its IP address. By default filtering rules are active for one day, in case a host is identified by its MAC address, and three hours, if a host is identified by its IP address. The timeout for hosts identified by IP address is shorter, because the IP address could be released and get used by another host, whereas the MAC address is unique. Both timeouts can be defined by the user. Flow storm mitigation is active for 15 minutes. Every 30 seconds it is checked if a rule has to be deactivated.

Apart from active mitigation by changing packet filter rules, OpenLIDS logs mitigation events with log level notification. It is also possible to notify an operator by email. Email notification is built on libsmtp-- [37]. Active mitigation can be deactivated completely if OpenLIDS should be run as an IDS only.

5.6 Deployment

By using the command line argument `-d` OpenLIDS can be run as a daemon in the background. On wireless mesh boxes it is recommended to either log to a remote syslog or to set the loglevel to warning or error to prevent the log information from using up space in RAM or on the filesystem. It is recommended to test OpenLIDS without active mitigation first to adjust detection thresholds if necessary. The maximum values of the counters are logged on log level debug on program termination to give some help. Detail check state transitions are logged on log level info. To run OpenLIDS, Netfilter rules for capturing, mitigation and spoofing protection have to be set up. Appendix B gives advice on this. Appendix A describes compilation and installation.

Ideally, all mesh boxes run OpenLIDS and clients connect to the network with IP addresses from a specified range per mesh box. If the network does not allow incoming connection attempts, it is sufficient that each mesh box only monitors its client network range. Complete coverage can be achieved, if the software for the routers is provided by a wireless community and the configuration of OpenLIDS (e.g. setting the network range for which the MAC address should be used as identifier) is done with a dynamic script.

In a general case however, it cannot be expected that each mesh box runs OpenLIDS. If users install software on their routers from scratch without using a prebuilt image or if the prebuilt image does not contain OpenLIDS by default, only a subset of the routers will run OpenLIDS. To protect the Internet uplinks from congestion and the network from malicious ingress traffic, if neither NAT is applied nor incoming connections are filtered, it is recommended that every mesh box with an Internet uplink runs OpenLIDS. To prevent the effects of congestion caused by malicious traffic in the wireless mesh network, it is important to block

traffic as close to its source as possible. This basically means the more sensors and the more diverse coverage the better. In the case that not every meshbox runs OpenLIDS, the ones which do will have to monitor the whole wireless mesh network range. This means that potentially multiple routers monitor the same host. Any of them might be the first which detects a violation. If the violation is caused by spamming, active TCP connections will stall independently of which mesh box is blocking. Further connection attempts could trigger remediation on other sensors closer to the source as well. For the other types of violation a similar effect will occur. A scanning host might first be blocked by a sensor several hops away. As the host continues scanning closers sensors will react as well and the host is finally blocked by the closest sensor. The network regions further away are protected from the effects of the malicious traffic, as long as they are out of radio range.

If mesh boxes are monitoring more than their own client network and clients are masqueraded with NAT, a single observed IP address can represent multiple clients. This means that several clients share the same counters and all get affected in case of remediation. Further, thresholds are reached faster. The thresholds could be raised, but this would allow attacks to remain undetected. There is no real solution for this except making every client visible by running OpenLIDS on every meshbox or not to use NAT. This is a general problem of anomaly based intrusion detection systems. No shared counters exist for content pattern matching. As long as only a small number of clients get masqueraded by a single mesh box, the problem is not very severe. For mesh boxes with a high number of clients the severity can be reduced by installing more mesh boxes at the same place to distribute the number of clients.

Chapter 6

Evaluation

In this chapter OpenLIDS is evaluated in terms of performance (Section 6.1), time until malicious traffic is detected (Section 6.2), the risk of false positives (Section 6.3), interference with the router's operational tasks (Section 6.4) and security (Section 6.5). For experiments the same two-hop wireless setup is used as in Chapter 3, if not indicated otherwise.

6.1 Performance Analysis

This section analyses the performance of OpenLIDS with traffic at high throughput. First the performance of connection tracking in user and kernel space is compared (Section 6.1.1). Section 6.1.2 analyses the performance of analysing benign traffic and Sections 6.1.3 and Section 6.1.4 analyse attack conditions.

6.1.1 Connection Tracking

The implementation of OpenLIDS provides the possibility to use the connection tracking of the Netfilter system or to perform connection tracking itself. To test the performance of both approaches a UDP flood with random source ports was generated at maximum throughput. The IP packet length was chosen as 1028 B and the packet rate was 1753 pps on average. The violation checks have been disabled, such that OpenLIDS does not take remediative actions during these tests.

To route the test traffic without OpenLIDS running the CPU usage on the router was 8 %. In Section 5.3.1 it was shown that simply receiving the Netlink connection tracking events from the kernel already drives the CPU load up to 49 %. Performing connection tracking in the application does not have this overhead. OpenLIDS when using the kernel connection tracking caused 99 % CPU usage for the described traffic. Apart from the high CPU load, 62 % of all new connection events were dropped. The packet capturing drop rate was 1 %. The high load is caused by receiving the connection tracking events, receiving the packets and issuing a hash table lookup to obtain the host structure to count the connection initiation. As the hash table only contained a single host entry, no lists had to be followed to find the entry and hashing therefore constitutes the main overhead of the lookup.

Using the kernel connection tracking for OpenLIDS has performance problems during high rates of new connections, even if the system without OpenLIDS running performs fine. Tracking connections in user space performs even worse. The CPU was exhausted with 100 % usage and 93 % of all packets could not be analysed. Here the user space connection tracking has to perform a hash table lookup to check if the connection already exists. As it does not, a new connection is allocated and stored in the table which causes another hash calculation. With this high drop rate, the limit of maximal 10'000 connections in the hash table was not reached. Each connection has two entries in the hash table, one for each flow direction. The average length of the linked lists of the buckets with 1024 buckets in the hash table was 14. As the rate of analysed packets is low and we will see later, that computing hash values at high rates is not very expensive, it can be concluded that the performance overhead is caused by allocating connection structures and by following linked lists. In Section 3.2.3 it was observed that hash table lookups and allocations of connection tracking structures are expensive at high rates. Here the rates are lower, but the application is using CPU resources for packet capturing and host table lookups as well.

The use of the Netfilter connection tracking for traffic with a high rate of new connections suffers from high CPU load and event drops. It is however more efficient than tracking connections in the application itself. Kernel connection tracking will therefore be used in further experiments. The high CPU usage and event drop rate, which occurs already during a flood with UDP packets of large size, demonstrates the need for the flow storm protection.

6.1.2 Benign Traffic

Small Number of Observed Hosts

Here the performance overhead of analysing benign traffic at high throughput is tested. A TCP steam with large packet sizes has been generated with Netperf, a network trace from the Wray community WMN has been replayed at maximum speed and a UDP stream with 40 B packet size and fixed source and destination ports has been generated. As described in Section 5.4 a lookup to the host table is only performed when required. This is not the case for most TCP and UDP packets. To test the overhead of the lookup, IP spoofing detection has been enabled in a second test. Enabling IP spoofing detection causes a lookup for all UDP packets.

The results of these measurements are shown in Figure 6.1. Routing the Netperf and the Wray traffic without OpenLIDS running caused 8 % and 9 % CPU usage. Analysing the traffic with OpenLIDS caused 13 % and 18 %. The Wray traffic consists of a much higher number of flows and contains more packets which cause a hash table lookup. Analysing this traffic is therefore slightly more expensive. The performance overhead of analysing normal traffic at high packet rates caused by OpenLIDS is very small.

The UDP stream with the small packet size had a packet rate of 3793 pps. This is much higher than the 1862 pps of the Wray trace. Higher packet rates cause higher capturing and decoding load which can be seen at the CPU load of 26 % for the UDP stream. Enabling IP spoofing protection further increases the load to 31 %. The difference of 5 % is caused by the hash calculation for the

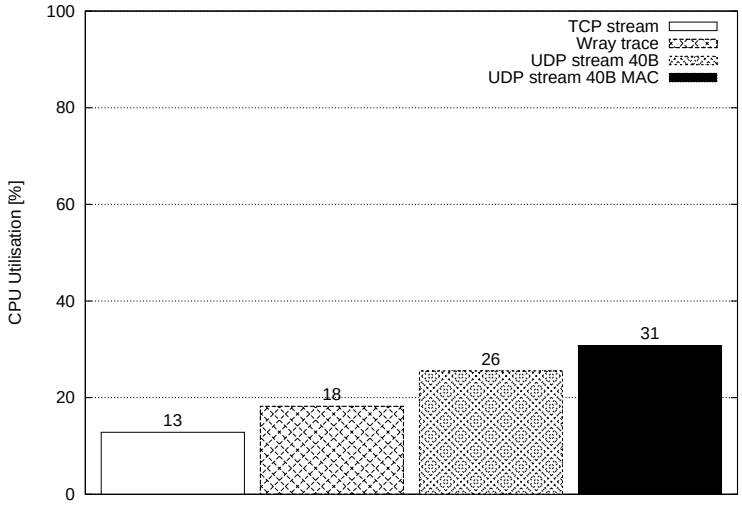


Figure 6.1: OpenLIDS performance to analyse benign traffic. *MAC* denotes enabled IP spoofing protection which causes a host table lookup for every packet. Analysing benign traffic at a high load is efficient and leaves resources for other tasks.

host table lookup. Even with high packet rates, hash calculations contribute only a small part to the total load.

Capturing the packet headers, decoding them and calculating a hash for the lookup is very efficient with OpenLIDS. Here the 512 buckets of the host structure hash table were only sparsely filled. For the generated traffic only one host had to be observed. The Wray trace contained 43 client hosts. This does not allow to evaluate the effect of following the hash lists, as they are too short. The effect of a full hash table is discussed below.

Bro failed to reliably run simple anomaly detection algorithms under high traffic load. The implementation of a dedicated anomaly detection system allows to cope with high traffic volumes while still providing CPU resources for other tasks.

High Number of Observed Hosts

A hash table lookup consists of calculating a hash to obtain a bucket in the table. Then the linked list of this bucket is followed and the entries compared for equality to the key used in the lookup. Here the performance impact of following a long hash list to find a particular host entry is tested. The number of buckets has been set to 128 and the hash table is filled by sending 10'000 UDP packets with random source addresses. The average list length was therefore 78. Then a UDP stream of packets with 40 B size and fixed ports was generated at maximum speed. The destination dropped all packets it received. In this test OpenLIDS has been modified to perform a hash table lookup for every packet.

Later created host entries will be at the head of the linked list. The host entry of the UDP source has been created before the hash table was filled and is therefore at the tail of the list. For cross comparison another UDP stream was

generated, whereas the host entry of this source was at the beginning of the list and the list did not need to be followed.

The CPU usage in all CPU diagrams was obtained by reading `/proc/stat` every second. OpenLIDS uses the same approach to detect high CPU usage, but reads the file every 5 seconds to be more resistant to bursts. If the router's CPU resources were exhausted during the tests, the process measuring the CPU usage did sometimes not get executed for several seconds. Measured CPU usage values have been tagged with a time value with a resolution of one second.

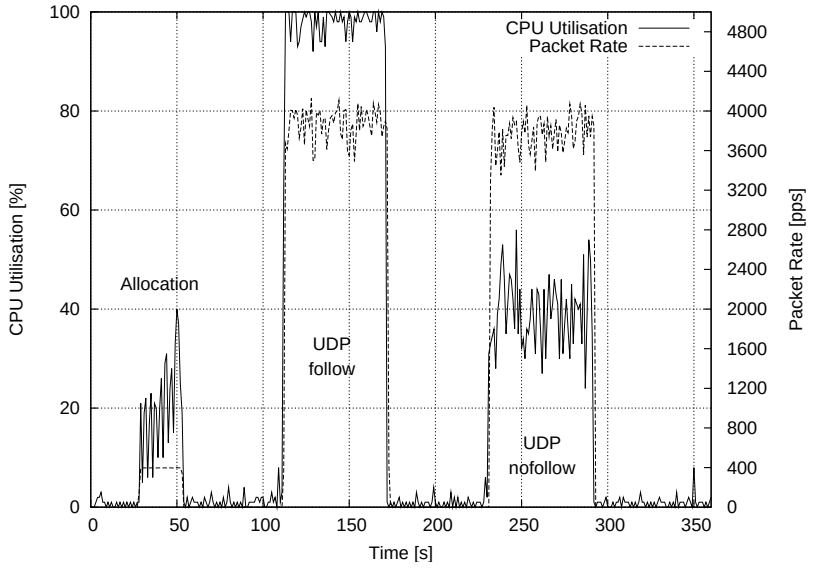
Figure 6.2(a) shows the CPU load and the packet rate of this experiment. The packet rate has been measured at the destination. From second 28 to 53 the host table is filled with a packet rate of 400 pps. Each packet triggers a connection tracking event and both the packet and the ctevent cause a hash table lookup. The first lookup causes a host structure to be allocated. It can clearly be observed that as the hash table gets filled, the cost of the lookup increases. The small spikes during idle times are caused by violation checks every 10 seconds, timeout checks every 30 seconds and counter resets every 60 seconds. The slightly bigger spike at second 250 is caused by deallocating a high number of host structures which timed-out.

From second 112 to 173 the first UDP stream is generated at a packet rate close to 4000 pps. The host structure of this traffic source is at the end of the linked list of its bucket. To retrieve it, the list has to be followed and the addresses of its entries have to be compared for equality. The CPU was fully exhausted and almost 24 % of the packets were dropped. The host structure of the source of the UDP stream generated from second 232 to 292 is at the head of the linked list of its bucket. For these packets only a hash value had to be computed, but the lists did not have to be followed.

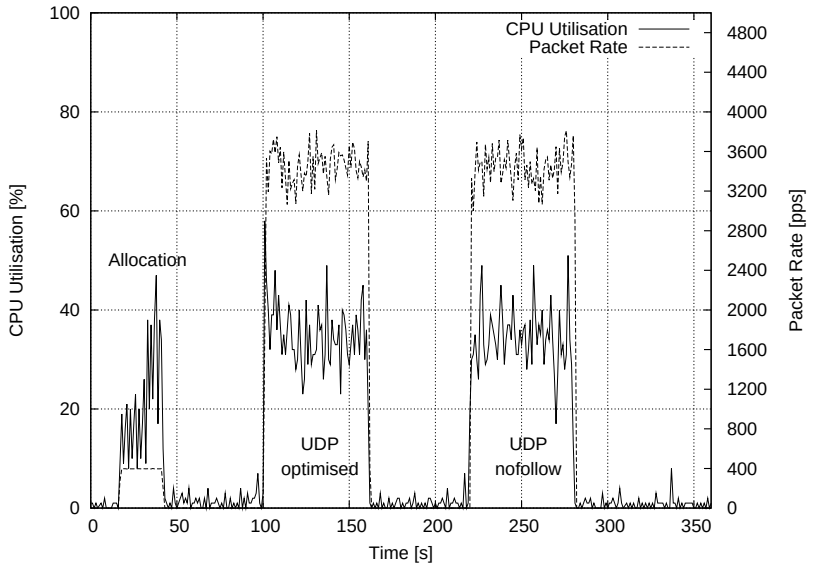
The performance impact of following the linked list and comparing the addresses is unexpectedly big, but similar performance drawbacks have already been observed for the rule lookup of Snort (Section 3.4.1). The situation presented here is a worst case scenario. The host looked up at a high rate is always at the end of the linked list. Typically, the hash table contains less entries, has more buckets, the packet rate is lower and a host entry is found before the end of the list is reached.

As a consequence of the significant performance impact for following the linked list, the hash table implementation has been optimised. On every lookup the host structure which was found is moved to the head of the list. This is possible as the linked list implementation allows insertion and removal of elements in constant time. With this optimisation, frequent lookups for the same host result in better performance. Worst case lookup is still as bad as before, but the worst case is now that all hosts are looked up in a round robing manner. This is less likely than the worst case before which was a frequent lookup of the host structure at the tail of the list.

The result of the optimisation is shown in Figure 6.2(b). The performance to lookup a host structure which was at the tail of the list on the fist lookup could be drastically improved as expected. The spike in the CPU usage at the beginning of the first UDP stream can have been caused by following the list over its full length on the first lookup. Because of its effectiveness, the same optimisation was applied to all hash tables of OpenLIDS.



(a) Original



(b) Optimised

Figure 6.2: Performance impact of following the linked lists of the host table with many entries for a UDP stream with high packet rates with the original and an optimised hash table implementation. The optimised implementation moves the entry of a successful lookup to the head of the tail and can successfully improve the average performance.

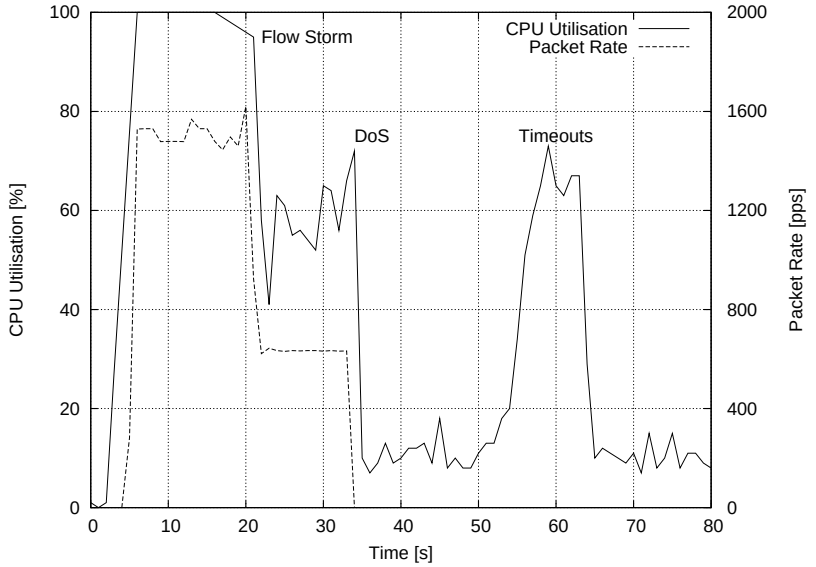


Figure 6.3: Flow storm protection and denial-of-service attack detection during a SYN flood attack with random source ports and high packet rates. After the flow storm protection is activated, the sending rate is throttled to 600 pps which reduces the CPU load on the router and provides resources for operational tasks. After the SYN flood is detected it is blocked. Existing connection structures time-out after 30 seconds.

6.1.3 Denial-of-Service Floods

SYN Flood with Random Ports

Here the detection of a SYN flood with a high packet rate and random source ports is evaluated. In Section 3.2.3 and Section 6.1.1 we have seen that traffic with a high rate of new connections causes high load on the router, which impacts its operational tasks as well as the intrusion detection. The flow storm protection detects high load situations based on the CPU usage and the occurrence of packet and event drops and throttles clients with a high rate of new connections. This restores the capability to operate in a usual way.

During a SYN flood with random source ports at full throughput only 3 % of the new connection events have been received by OpenLIDS. The threshold to activate flow storm protection for a certain host is set to 500 new connections per minute. If 1000 SYN packets are sent per second and only 3 % can be analysed, it will take 17 s until the threshold is reached. As soon as a flow storm is detected, the sending host will be limited to send at most 600 packets per second. As we will see, this limitation is sufficient to reduce the CPU load.

For this experiment, a SYN flood with random source ports has been generated at maximum throughput. The destination silently dropped all packets it received. The CPU usage and the packet rate is shown in Figure 6.3. The SYN flood starts after four seconds. The host structure is created with a delay of 3 seconds. At second 14 a first violation check is performed, the flow storm is detected and the packet rate of the source is limited to 600 pps. The limitation

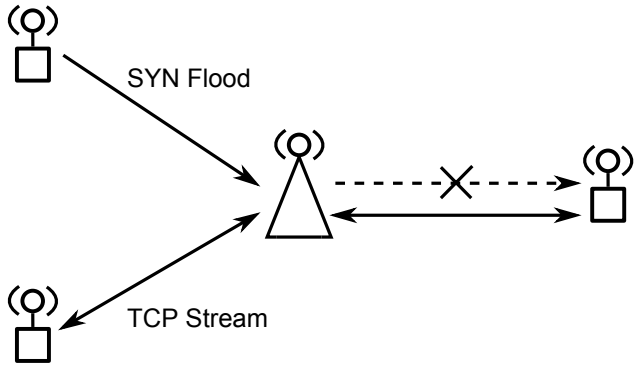


Figure 6.4: Setup for quality of service measurements with Thrulay during SYN flood attack

becomes effective at second 19 and Iptables reports success at second 20. The delay to create the host structure and the delay between flow storm detection and successful modification of the Netfilter table show that hardly any computational resources are left while the system is heavily overloaded with kernel computations. The higher priority of the privileged OpenLIDS process and the Iptables process do not help to prevent the delay of the filter activation. The kernel still has a higher priority and uses almost all CPU time. After the rate limitation is activated, the CPU load is reduced immediately. No more packets or connection events were dropped later.

The connection tracking table of the kernel can store at most 4096 entries with the default settings of OpenWRT. As the source is sending at a high rate, the table is filled within seconds. To be able to allocate entries for further connections, existing connections without any packets in the reply direction are destroyed immediately and a destroyed connection event is generated. Because the buffers have already been filled with new connection events and the application hardly gets any runtime, less than 50 failed connection attempts have been counted at the first violation check at second 14. The overflowing connection table has the advantage that the connection attempt can be counted as failed before the usual timeout of 30 seconds. Up until the second violation check at second 24, enough destroyed connection events were received and the detail check for failed TCP connections is activated. Until the third violation check at second 34 more than 400 failed connection attempts have been counted and the denial-of-service attack is detected. All packets from the source to the attacked target are dropped immediately. As the source is still sending and the packets have still to be processed, the CPU usage does not go back to zero but stays around 10 %. The connection table is still filled with 4096 connection entries. They time-out 30 seconds after their allocation. This causes a further increase in CPU usage near second 60.

Even if the attack is blocked at the router, it continues going on, but the impact is now local only. No more uplink bandwith is wasted and the wireless channel out of the source's radio range is not congested anymore. To analyse the effect of a blocked attack on the local wireless channel, another experiment was carried out. Again a two-hop setup was used (see Figure 6.4). One host sent a TCP stream to another host in the network. A third host started a SYN

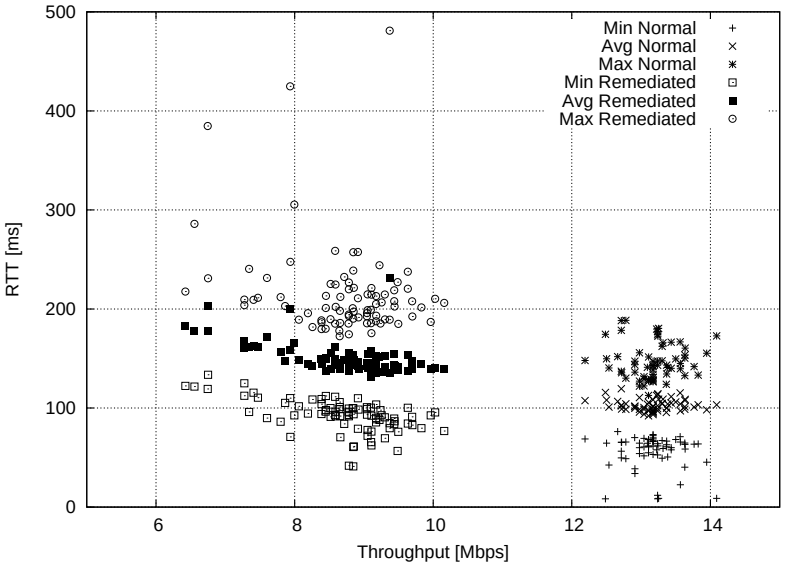


Figure 6.5: Round-trip times and throughput in normal situation and in the presence of a blocked SYN flood attack at a high rate. The data points between 12 and 14 Mbps were measured in the absence of the attack, the ones between 6 and 8 Mbps while the attack rate was limited to 600 pps and the data points between 8 and 10 Mbps were measured after the attack was blocked completely. As long as no mitigation was active, the attack caused the TCP stream to stall completely.

flood attack against the same destination. OpenLIDS detected and blocked the attack at the wireless router in the middle. *Thrulay* [74] has been used to generate the TCP stream and analyse throughput and round-trip times. Both values were measured in one second intervals.

The results of these measurements are shown in Figure 6.5. During the first 60 seconds only the TCP stream was active. The throughput was between 12 and 14 Mbps and round-trip times varied between a few milliseconds and 200 ms. Then the SYN flood attack started. The router became very busy tracking the high number of new connections, such that the TCP stream totally stalled. The data points of this phase are not shown in the diagram. After 10 seconds the flow storm was detected and the detail check for the failed TCP connection attempts was started after 20 seconds. The TCP stream started to recover at the same time. The recovery did not start immediately after the flow storm protection got activated. In the next 10 seconds the throughput was below 8 Mbps. Then the SYN flood was detected and blocked and the throughput stabilised between 8 and 10 Mbps. As the SYN packets were not forwarded anymore, the average and minimum round-trip time clearly decrease. Because the SYN flood still occupies the wireless channel, the thoughput remained below and the round-trip times above the values which have been measured in the absence of the attack.

OpenLIDS is able to reduce the load on the router in a first step which already allows benign communication to recover and enables correct execution of operational tasks on the router. After the attack is blocked completely, the quality of service is still affected, but the effects are limited to the source's radio range and other parts of the wireless mesh network are not affected anymore.

SYN Flood with Fixed Ports

As a continuous SYN flood attack with fixed source and destination ports for all packets does not lead to any failed connection attempts, the SYN tuple check has been implemented. Here the performance of this check is tested at maximal throughput.

Figure 6.6 shows the CPU utilisation and the packet rate during this test. After 4 seconds the first packets are received and the CPU usage increases due to the high packet rate and the need to perform a lookup to the host hash table for every packet. After 12 seconds the first violation check is performed and as the difference between new connections and SYN packets is bigger than 50, the SYN tuple check is activated. Every SYN packet causes now another hash table lookup to be counted per connection. A higher CPU load is not clearly observable. There are two peaks in CPU load but also two decreases. The decreases correlate with temporary decreases in the packet rate. It can be concluded that activating the SYN tuple check in the presence of a single stream does not lead to much higher costs, but a small increase in CPU usage has to be expected. Analysing SYN packets of multiple connections would be more expensive, as more hash table entries have to be allocated and the table would be fuller. As normal connections do not have a high SYN rate, activating the SYN tuple check does not cause a big overhead. This is particularly important, if a host enters the SYN tuple check state without carrying out a SYN flood attack.

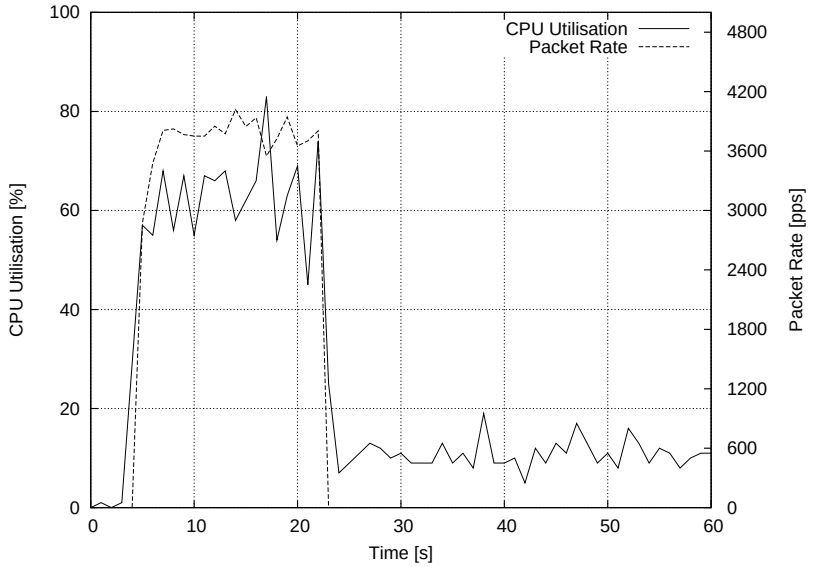


Figure 6.6: Detection of a SYN flood attack with fixed ports and high packet rates with the SYN tuple detail check.

6.1.4 Scanning

A real world scan of random IP addresses on the Internet has been performed to obtain a network trace with real replies (see Section 6.2.2). To test the performance of detecting scanning traffic, this trace was replayed at maximum speed. Figure 6.7 shows the CPU usage and the packet rate. Blocking the source to contact the scanned port is not enough to reduce the load here. As the trace continues to be replayed after filter rules have been activated, ICMP messages and TCP resets of the scan would continue to be analysed. Here the filter rules blocked all traffic instead of just the packets coming from the scanning host.

The scan starts at second 2 and the CPU resources get exhausted because of the connection tracking overhead. Almost every packet belongs to a new connection, which is similar to the SYN flood with random source ports. The first violation check is performed at second 12. The number of counted new connections was only 313 at this point which is not enough to activate the flow storm protection. During the SYN flood attack already the first violation check was sufficient to detect the flow storm. To have faster response times, the threshold could be easily decreased to 300. A higher chance of false positives is not a problem here, as the rate limitation is very unlikely to affect benign hosts. The flow storm is detected at second 23, but it took 19 seconds to change the Netfilter tables. This again demonstrates that the system is totally overloaded. User space applications hardly get any runtime.

For the violation check at second 33, the number of failed connection attempts was only 30. The flow storm protection was not activated at this time yet. The next violation check at second 43 then activated the detail check which counts the number of destinations of failed connection attempts. Due to the overhead for accessing the hash table to count the destinations, the CPU

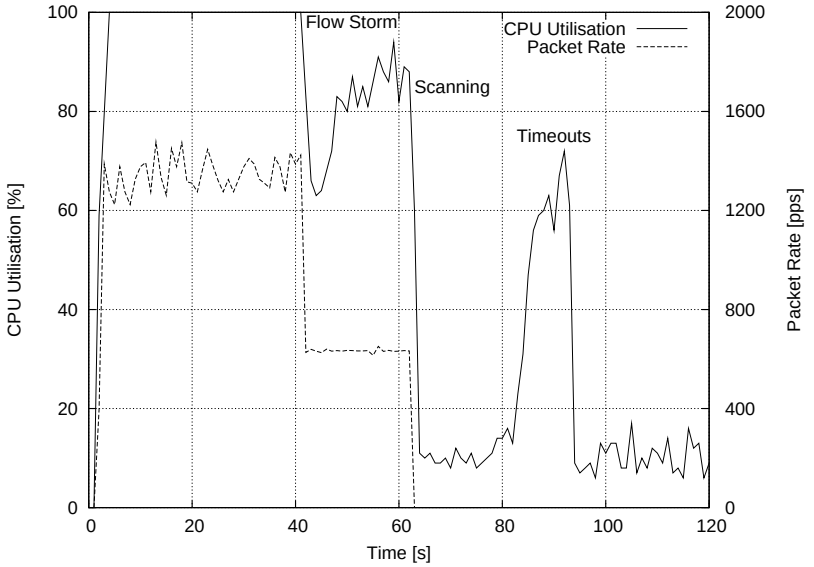


Figure 6.7: Detection of scanning with high packet rate. Detection of the flow storm took longer and activating two detail checks in a row leads to higher time to detection.

usage increases. At the next detail check at second 53 the number of destinations of failed connection attempts was higher than 150 and the host enters state DST_PORT. As in this state a lookup to a destination table and a port hash table is performed, the CPU usage again increases slightly. The scan is finally detected and blocked at second 63.

Activating two detail checks and checking for violations every 10 seconds leads to a lower bound for the detection time. As we have seen in Section 6.1.2, the overhead for violation checks is not substantial, even if a high number of hosts is monitored. A check period of 10 seconds is considered as unnecessarily long. Shorter violation checks will increase the time to detection which is particularly important to detect flow storms. The period of violation checks has been changed to 2 s. For all experiments shown in this chapter the original period of 10 s is used.

6.1.5 Implications

In this section as well as in Chapters 3 and Section 5.3.1 it could be observed that several operations are comparably expensive in terms of computational overhead. List lookups were most expensive, but the performance impact of memory operations (allocation, deallocation and copying) and hash value calculations was observable too. Because the resources on common wireless mesh boxes are constrained, these operations cannot be performed at high rates. This limits the possibilities of software running on such devices. Trade-offs can be made between memory and computational resource usage. Hash tables with a larger number of buckets occupy more memory, but have shorter linked lists and are therefore faster.

OpenLIDS reorganises the linked lists of hash tables to reduce the average costs of lookups. It allocates memory on the stack in favour of dynamic memory allocation wherever possible. New hash table entries for detail checks are only allocated as long as the thresholds are not reached yet and the allocation is really required for detection. In normal conditions, when host structures are already allocated and no detail checks are active, hardly any dynamic memory operations are performed and the main performance overhead is caused by host table lookups. These lookups are only performed when necessary and obtaining the bucket index does not involve divisions. Similar techniques can be used by other applications for constrained hardware.

6.2 Time to Detection

In this section the time until malicious traffic is detected is analysed. DoS floods (Section 6.2.1) and scanning traffic (6.2.2) are simulated. No experiments with real or simulated worms have been performed here. The failed connection attempt algorithm was already shown to be effective by [20].

6.2.1 Denial-of-Service Floods

Denial-of-service attacks are detected based on failed connection attempts. As long as the target is able to reply to requests, be it SYN packets, ICMP echo requests, or UDP requests on application layer, the attack is not detected. As soon as the target gets overwhelmed or if UDP is used to saturate the target's network capacity rather than issuing application layer requests, connection attempts time-out and can be counted as failed connection attempts. In the experiments of this section the target silently dropped packets and the packet rate therefore corresponds to the rate of failed connection attempts. UDP floods with fixed source and destination ports have been excluded from detection, due to possible confusion with certain kinds of benign traffic (see Section 4.1.2).

Here the detection of denial-of-service attacks at lower rates is evaluated. Time to detection mainly depends on the type of traffic, the packet rate, and the time of periodic violation checks as well as counter resets. Figure 6.8 illustrates the effect of the periodic activities on the detection time of a UDP flood in a worst case scenario. This example assumes a packet rate of 4.9 pps and the use of random source ports. The first failed connection attempts can be counted after 60 seconds, when the first connection attempts start to time-out. From then on, the failed connection attempt counter of the source steadily increases. In a worst case scenario the counter is reset just shortly before the threshold of 50 failed connection attempts is reached. The counter reset occurs once every 60 seconds. This possibly delays detection up to the time which is needed to reach the threshold again. At second 80 a violation check is performed, but at that time the counter is at 49 failed connection attempts instead of the 50 which are required to activate the detail check for failed connections. Even if the threshold is reached shortly after, it takes another 10 seconds until the next violation check. At second 90 the detail check is started which counts failed connection attempts per destination. The counter value shown in the diagram starting at this point is the counter of the destination. At second 170 a violation check is performed, where again the counter value was just below the

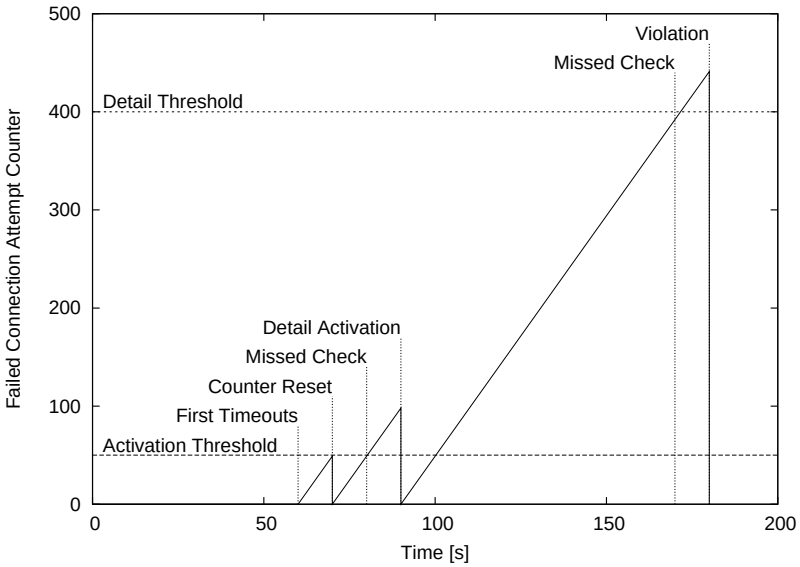


Figure 6.8: Worst case time to detection of a UDP "flood" with 4.9 pps. The exact moment of periodic counter resets and violation checks affects the worst case time to detection.

Attack	10 pps		100 pps		1000 pps	
TCP (random port)	85 s	(45 s)	50 s	(40 s)	20 s	(10 s)
TCP (fixed port)	21 s	(11 s)	20 s	(10 s)	14 s	(4 s)
UDP (random port)	121 s	(71 s)	62 s	(52 s)	22 s	(12 s)
Ping	30 s		6 s		5 s	

Table 6.1: Time to detecion (and start of detail check) for denial-of-service attacks at different rates. Awaiting connection timeouts at low rates causes a big part of the delay. At higher rates connections are destoyed immediately due to an overflowing connection table.

threshold of 400 failed connection attempts for a single destination. The attack is finally detected after 180 seconds. Detection takes a lot of time, but this is acceptable because the packet rate is very low and detection of this traffic is therefore not of high importance (Requirement 1). Detection will be faster for higher packet rates, but also in this case it is possible that violation checks are performed shortly before the thresholds are reached and another 10 seconds pass until OpenLIDS reacts. Because of these detection delays caused by periodic violation checks, the period at which the violation checks are performed are set to 2 seconds in the final version of OpenLIDS. Here the 10 second period is used.

Table 6.1 shows the time to detection of different DoS attacks at different packet rates. The values in brackets indicate the time at which the detail check has been started. The values have been obtained with single measurements each. Because the time to detection depends on the time of periodic activities, a high number of experiments would have been necessary to obtain meaningful

minimum, average and maximum detection times. The values obtained from single measurements are sufficient to discuss some peculiarities.

During a SYN flood, first connection timeouts are received after 30 seconds. For the experiment with 10 pps, a violation check is performed after 35 seconds. This check examines the failed connection counter just before it reaches 50. The detail check is activated one violation check later. For the experiment with 1000 pps the detail check is started just after 10 seconds. This is possible because the connection tracking table starts to overflow shortly after 4 seconds. Compared to the SYN flood with random source ports, the detection of the SYN flood with fixed source ports is much faster. First, the information about number of SYN packets and new connection attempts is available immediately. It is not based on counting timeouts which cause a delay of 30 seconds. Second, the attack is detected 100 packets after the detail check has been started instead of 400 for the failed connection check.

The detection of the UDP flood with random source ports is similar to the detection of the SYN flood with random source ports with the difference that the connection timeout is 60 seconds. The effect of the overflowing connection table takes already place at 100 pps. Shortly after 40 seconds the the table is full and destroyed connection events are generated. For the detail check after 42 seconds however, less than 50 failed connection attempts could have been counted. Small delays in generating and receiving destroyed connection events as well as the one second accuracy of OpenLIDS are the cause for that. With the help of the overflowing connection table effect, the detection of UDP floods at higher packet rates is not much slower than the detection of SYN floods with random source ports, even if the connection timeout is twice as long. After the detail check has been activated for the UDP flood at 10 pps, 50 seconds pass until the attack is detected. At the detail check after 40 seconds, the threshold has not been reached yet.

The detection of ICMP echo request floods is different, as it does not involve any detail checks and is based on events which are observable immediately without awaiting timeouts, similar to the detection of SYN floods with fixed ports. A violation is detected, if the difference between echo requests and echo reqplies is greater than 220. At a packet rate of 10 pps, this is the cause after 22 s. As here the detection is quite fast even for small packet rates, increasing the threshold could make sense to be more resilient against possible false alerts while still providing sufficiently fast detection (Requirement 2).

Generally, it can be concluded that the higher the packet rate is, the faster the attack is detected, as long as the router is not overloaded. Higher packet rates result in a faster increase of counter values and eventually cause the connection table to overflow before timeouts occur. This is a nice property, as remediation takes place faster for attacks which cause a higher damage to the network. Not using detail checks for ICMP also results in much faster detection. For smaller packet rates, the connection timeout has a big impact on the time to detection. As the timeouts cannot be adjusted arbitrarily, if the kernel connection tracking is used, performing connection tracking in user space would lead to faster detection. As the difference diminishes with higher packet rates and the kernel connection tracking is more lightweight, using user space connection tracking for faster detection has to be well considered. The final version of OpenLIDS performs violation checks every 2 seconds instead of 10. This will lead to faster detection.

Reply	Internet	Lancaster
TCP SYN/ACK	0.4 %	0.6 %
TCP RST	2.1 %	1.8 %
ICMP destination unreachable	2.0 %	11.5 %
ICMP TTL exceeded	1.1 %	0 %

Table 6.2: Replies of scanning TCP port 22 on random addresses and addresses in the Lancaster University network. The local scan leads to a higher rate of ICMP destination unreachable messages due to unroutable addresses.

6.2.2 Scanning

To test the detection of scanning traffic, two real world scans have been performed to obtain meaningfull network traces. SYN packets have been sent to SSH port 22 to 500'000 random IP addresses and to all addresses in the Lancaster University network 148.88.0.0/16. The scan rate was 10 pps. The SYN packets and the replies have been captured with Tcpdump. The SYN/ACK packets of successfull connection attempts were replied with a reset packet to terminate the connection immediately. The network traces obtained here have been replayed with Tcpreplay to simulate a scanning host. A random scan and a local scan have been chosen to represent different scanning strategies.

Table 6.2 shows the replies which have been received during the scans. The Lancaster University network has a high number of Windows machines. Windows does not run an SSH service by default and it is likely that most of the machines are running the Windows Firewall which drops connection attempts instead of replying with TCP resets. A high number of existing hosts did therefore not reply at all. The scan of the Lancaster University resulted in a comparably high number of ICMP destination unreachable messages. The university has about 17'000 students and 2000 members of staff. A high number of addresses out of the 16 bit network block seems not to be used. It has been observed that the number of ICMP replies sent from a single router was limited to a rate of 1 pps. With slower scanning speeds the ratio of ICMP replies would therefore be higher, whereas it would decrease at higher scanning speeds.

Apart from ICMP unreachable messages, the Internet scan resulted in ICMP time exceeded messages being replied. As the TTL was set to 255, there probably existed some routing loops. The time exceeded messages are not counted as failed connection attempts. They could e.g. be caused by Traceroute. Interestingly, a small number of TCP reset packets which were returned, did not have the sequence number set to zero. This is in violation with the TCP specification. As this is used to distinguish reset packets on connection attempts from reset packet on loss of synchronisation, these reset packets will not be counted as failed connection attempts.

Table 6.3 shows the time to detection for replaying both scan traces at different speeds. The first time in the brackets indicates the time at which the host enters TCP failed connection state DST_HOST and the second DST_PORT. Again these times have been obtained with single measurements.

Because rejected connection attempts can be counted immediately without awaiting a timeout, the first detail check can be started faster as during the SYN flood attack. This is particularly observable at 100 packets per second.

Attack	10 pps		100 pps		1000 pps	
Internet	114 s	(44 s / 104 s)	50 s	(20 s / 40 s)	31 s	(11 s / 21 s)
University	77 s	(37 s / 57 s)	25 s	(5 s / 15 s)	30 s	(10 s / 20 s)

Table 6.3: Time to detecion (and start of detail checks) for a TCP scan at different rates. More reject packets during the local scan result in faster detection because fewer timeouts have to be awaited.

The Internet trace at that this speed contained 50 reject messages within the first 10 seconds. The violation check was just performed before the threshold was reached. The University trace already contains 50 reject messages in the first 4 seconds and the DST_HOST state is activated immediately. To reach state DST_PORT failed connection attempts have to be observed for 150 distinct destinations. In the time from 20 to 30 seconds the Internet trace contained 36 reject messages. This was not enough to activate state DST_PORT after 30 seconds. The University trace contained 157 reject messages in the first 10 seconds in state DST_HOST and the port check could be activated on the next violation check. For both traces the scan is detected 10 seconds after the port check has been activated. The reason, why the detection with the University trace at 100 pps is faster than the detection at 1000 pps, is that the first violation check was performed earlier.

The reject messages did not help to enter state DST_HOST faster at 10 pps in these experiments. The Internet trace contained 21 reject messages at the time of the violation check before the state is changed whereas the University trace contained 30. The result here is the same as if the connections just timed out after 30 seconds. For both traces 20 seconds were needed to reach the threshold of 150 distinct destinations of failed connection attempts to move to state DST_PORT. An interesting effect can be observed for the University trace before the scan is detected. 20 seconds are spent in state DST_PORT whereas the internet scan spent only 10 seconds in the same state. This could be explained by just not having reached the threshold on the first violation check. The detection threshold is 100 failed connection attempts to the same port which is exactly the maximum number of failed connection attempts a host sending with 10 pps can have in 10 seconds. Because detection highly depends on the exact moment of the violation check, it is likely that during a violation check after approximately 10 seconds the threshold has not been reached yet. This explanation would be reasonable, but here the detection takes longer for another reason. In the time from 27 to 37 seconds, 9 hosts reply with a SYN/ACK and 18 connection attempts get rejected. Accepted connections are not counted as failed connection attempts and rejected connection attempts are counted immediately. The remaining 73 connection attempts time-out 30 later, i.e. from second 57 to 67. In the same time only 10 connection attempts get rejected. This means only 83 failed connection attempts are counted until the first violation check after the host entered state DST_HOST. This is not enough to detect the scan at this time.

Nr	Trace Start Time	Hosts	Packets	Volume	Rate	∅ Len
1	23 April 2008 18:00	43	757'374	567 MiB	1.32 Mbps	785 B
2	21 April 2008 17:00	39	652'436	445 MiB	1.04 Mbps	715 B
3	22 April 2008 00:00	24	644'474	512 MiB	1.19 Mbps	833 B
4	20 April 2008 16:00	37	632'001	287 MiB	0.67 Mbps	476 B
5	26 April 2008 03:00	10	631'846	534 MiB	1.25 Mbps	901 B
6	15 April 2008 14:00	23	617'548	241 MiB	0.56 Mbps	409 B
7	23 April 2008 00:00	19	616'778	504 MiB	1.18 Mbps	857 B
8	15 April 2008 15:00	25	612'149	226 MiB	0.53 Mbps	387 B
9	23 April 2008 01:00	18	611'566	502 MiB	1.17 Mbps	861 B
10	22 April 2008 01:00	23	610'766	495 MiB	1.15 Mbps	850 B
11	23 April 2008 02:00	16	596'180	498 MiB	1.16 Mbps	876 B
12	21 April 2008 11:00	31	574'397	323 MiB	0.75 Mbps	590 B
13	21 April 2008 18:00	32	553'115	338 MiB	0.91 Mbps	641 B
14	14 April 2008 20:00	42	518'089	360 MiB	0.84 Mbps	729 B

Table 6.4: Summary of the Wray traces

Nr	Trace Start Time	Detail Check State Transitions
1	23 April 2008 18:00	one host 8 times TCP_SYN DST_TUPLE
4	20 April 2008 16:00	one host 6 times TCP_SYN DST_TUPLE
6	15 April 2008 14:00	one host once UDP_FAIL DST_HOST
11	23 April 2008 02:00	one host once UDP_FAIL DST_HOST
14	14 April 2008 20:00	one host 7 times TCP_SYN DST_TUPLE, once TCP_FAIL DST_HOST and 3 times UDP_FAIL DST_HOST with subsequent DST_PORT

Table 6.5: Detail check state transitions during Wray trace replay. All state transitions were caused by peer-to-peer applications. Except for the application in trace 6 (BBC iPlayer), there was no risk of a false positive.

6.3 False Positives

To test OpenLIDS for false positives network traces from the Wray community wireless mesh network have been replayed in Section 6.3.1, different peer-to-peer applications have been run in Section 6.3.2 and network diagnosis tool have been used in Section 6.3.3.

6.3.1 Wray Traces

To test the detection algorithms of OpenLIDS with real world traffic, captured network traces from the Wray village community network (see Section 3.2.2) were replayed at real-time speed. The 14 hours with the most number of packets have been selected. Table 6.4 shows the capturing time, the number of clients in the WMN, the number of packets, the data transfer volume, average data rates and average packet sizes of these traces.

Replaying the Wray traffic did not cause any false positives, but several detail checks were started. This is not a big problem because it does not lead to remediative actions, but it is unsolicited because activating detail checks

consumes memory and CPU resources. Activating many detail checks could use up all available memory or, if the maximum number of simultaneous detail check was reached, would cause checks to be postponed. Table 6.5 shows the traces which caused detail checks and and shows which state transitions occurred. The IP address of the hosts causing more detailed analysis were distinct in all traces and the five hosts were accessing four different mesh boxes.

While replaying traces 1, 4 and 14 the TCP SYN tuple check was activated multiple times for a single host. All of these network traces showed similar traffic patterns. The host which entered state TCP_SYN DST_TUPLE was trying to connect to a high number of destinations on unassigned ports. Most of the connection requests did not get replied. The host resent the SYN packet some seconds apart twice before giving up. It retried to establish a connection to some destinations up to 7 times. Because of resending SYN packets several times, if no reply was received, the difference between SYN packets and connection attempts was greater than 50 and the SYN tuple detail check was activated. As the connection attempts occurred to a high number of destinations and the source port changed between connection attempts to the same destination, there was no danger for a false positive. The threshold was always reached 50 or 60 seconds after the counters have been reset for the last time. Increasing the threshold to e.g. 70 would be sufficient to prevent the activation of the SYN tuple check. The destination IP addresses were mostly in client ranges of ISPs. It is possible that the contacted destinations would actually accept the connection requests, but are behind a NAT router which silently drops connection requests. Contacting a high number of client computers on random ports suggests that the hosts causing the detail checks were running a peer-to-peer application. We will see in Section 6.3.2 that BitTorrent causes the same behaviour as described here.

The hosts in traces 1, 4 and 14 did generally not cause more than 30 failed connection attempts per minute. In trace 14 the number of failed connection attempts however was once higher than 50 per minute and the host entered state TCP_FAILED DST_HOST. Because the host retried to establish a connection to a single destination several times connection attempts failed to less than 150 destinations in 5 minutes and no transition to state DST_PORT occurred. If it had, no violation would have been detected because the application uses random ports. Scanning is only detected, if more than 100 connection attempts fail to a single port.

In trace 6, a host was sending UDP packets to many destinations with both source and destination port 1948. Up to 64 failed connection attempts per minute were counted which caused the host to enter state UDP_FAILED DST_HOST. Most of them were caused by ICMP destination unreachable messages (host and port unreachable). Failed connection attempts occurred multiple times for single destinations such that the threshold for a transition to state DST_PORT was not reached. If it had been reached, the traffic would likely have been identified as scanning traffic because a single port is contacted. The application generating the suspicious traffic was identified as BBC's iPlayer [8]. It allows to download BBC TV and radio programs up to 7 days after they have been broadcasted and the programs can be stored on a user's computer up to 30 days. The application operates in a P2P manner. Content is exchanged between users who want to see the same shows. Apart from the ICMP unreachable messages a considerable amount of ICMP time exceeded messages was caused to be returned. These are not counted as failed connection attempts.

Both traces 11 and 14 caused a host to enter state UDP_FAIL DST_HOST. These hosts were sending UDP packets to a high number of destinations on random ports and a high number of packets was not replied. The host in trace 11 caused up to 70 and the host in trace 14 up to 261 failed connection attempts per minute. It is unsure, if both hosts ran the same application. If they did, the host from trace 14 was much more active. Because most destinations were in ISP's client ranges, it is assumed that the application generating the suspicious traffic was using a P2P protocol. In trace 14 the failed connection attempts were frequent and distributed enough to enter state DST_PORT several times. Because random destination ports were used, the traffic was not identified as scanning traffic.

6.3.2 Peer-to-Peer Applications

In the previous Section 6.3.1 several detail checks were activated due to peer-to-peer traffic in the Wray network. This section analyses OpenLIDS in combination with two popular P2P protocols. The experiments were carried out on a 1 Mbps ADSL connection and NAT was applied.

BitTorrent

BitTorrent [65] is a P2P protocol for file sharing. Here the *Transmission* [85] client implementation was used to download the first 100 MiB of a 4.1 GiB file to a Linux machine. Transmission reported 288 seeders and established connections to 50 peers. A high number peers were contacted without success. SYN packets are resent twice before the connection attempt is given up and connection attempts are repeated to single hosts. This is exactly the behaviour which was observed in the Wray traces 1, 4 and 14. Also here this lead to repeated activation of the SYN tuple check. The maximum number of failed connection attempts per minute was 48 and no TCP_FAIL check was stared. Because no fixed ports are used, starting detail checks would not lead to false postives.

eDonkey

Another test was performed with *aMule* [4]. This client supports the eDonkey and the Kademlia [42] protocols. aMule limits the number sources of a single file to 300, the total number of connections to 500 and the number of new connections per 5 seconds to 20 by default. The limitation of the rate of new connections poses an upper bound for failed connection attempt. Assuming all connection attempts will fail, the default setting is not low enough to prevent activation of detail checks or a potential violation detection.

For this test a 80 MB file was downloaded. The file had 277 sources. aMule sent a high number of UDP packets to a high number of hosts and the majority of these packets were not replied. *Wireshark* [91] decoded most of these unreplied packets as Kademlia requests and they were mainly sent in the first minutes of the download. They seem to be related to connecting to the Kademlia network. The high number of unreplied UDP packets caused the host to enter state UDP_FAIL DST_HOST and the state was changed to DST_PORT two minutes later. No violation was detected. aMule and other eDonkey clients use preconfigured standard ports. It is possible to change them, but it was

observed that a high number of packets were sent to UDP ports 4672 and 4665. aMule sent UDP packets to 273 distinct destinations during bootstrap and did not receive any replies back from 231 destinations. The maximum number of failed UDP connection attempts per minute was 91. Even if all packets would have been sent to a single port, this would not have been enough to be identified as scanning traffic. A more active client however could cause a false positive.

Skype

Skype [76] is an IP telephony and instant messaging application which uses a P2P model for communication. Skype for Linux with version 2.0 was used in this test. The application was started and instant messages were sent and phone calls were made to two communication partners. Skype did not cause more than 10 failed connection attempts per minute and is therefore not in danger to cause false positives.

6.3.3 Network Diagnosis Tools

Nmap

Nmap [54] is a network scanner for network exploration and security audits. It supports horizontal scanning of whole network ranges as well as vertical scanning of several ports on single hosts. Depending on the scan speed, the number of scanned destinations and the number of ports scanned per destination Nmap diagnosis will be identified as scanning traffic or as a DoS attack. Port scanning on a single host is possible without causing an alert, if less than 400 TCP or UDP ports are scanned within 5 minutes. Horizontal scanning will not be blocked, if no more than 150 destinations are scanned in 5 minutes.

Traceroute

Traceroute [84] is used to analyse routing paths. On modern Linux systems, traceroute by default sends UDP packets to the traced destination while varying the *Time-to-Live (TTL)* value in the IP header. Every packet is addressed to a different port on the destination address to be able to associate replies with sent packets. The TTL of the first packet is set to one and is increased with every packet until the packet finally reaches its destination. The short TTL causes routers on the path to the destination to drop the packet and reply with an ICMP time exceeded message. This allows to trace the path which the packets take. The time exceeded messages are not counted as failed connection attempts, but as soon as the packets reach the destination, they will be rejected with ICMP port unreachable messages. If the destination or a router in between is silently dropping the UDP packets, the dropped packets are counted as failed connection attempts. By default traceroute sends three series of at most 30 packets. It stops sending as soon as ICMP destination unreachable message are received. In the worst case, all 90 packets are silently dropped by the first-hop router. If one destination is traced more than four times within 5 minutes, this could be identified as a DoS attack. While tracing a server on the Internet from a British ADSL connection, all packets were replied and 15 packets were rejected by the destination with ICMP destination unreachable messages. This would allow to traceroute the same destination at least 26 times in 5 minutes

without being blocked. Traceroute is therefore unlikely to cause false positives on common use.

6.4 Impact on Routing Operation

Requirement 5 defines that OpenLIDS shall not hinder the operational tasks of the router. Section 6.4.1 discusses the impact of running OpenLIDS on the packet forwarding behaviour and Section 6.4.2 the CPU and memory ressource usage.

6.4.1 Packet Forwarding

To analyse the effect of OpenLIDS on the maximum throughput and packet delays, Thrulay was used to generate a TCP stream which was forwarded by the router. Three packet streams of 60 seconds duration were generated each with OpenLIDS running and without. No impact on the routing behaviour could be observed, neither on the maximum throughput nor on round-trip times. On average, round-trip times were even one percent lower and throughput 0.3 % higher with OpenLIDS running. This difference is caused by statistical deviations.

During the analysis of Snort in Section 3.4.1 it was observed that a high CPU load caused by computations in user space did not affect the routing behaviour. High load caused by connection tracking in kernel space reduced the forwarding capacity of the router. In Section 3.2.3 where OpenLIDS was not running the average rate at which SYN packets with random ports could be sent was 2737 pps. In Section 6.1.3 where OpenLIDS was running it was only 1500 pps. The difference could be caused by a higher load in kernel space to prepare and send the connection tracking events. This situation arises only in the presence of excessive malicious traffic and OpenLIDS helps to restore normal operation. The temporary impact of OpenLIDS on the forwarding rate is therefore justified.

6.4.2 Hardware Resources

CPU Resources

OpenLIDS is lightweight and does not consume a lot of CPU resources in normal conditions, even under high traffic load. Analysing the Wray trace 1 at full speed caused 18 % CPU usage, whereas replaying trace 14 at its real-time speed caused only 2.7% on average. Enough resources are left for other operational tasks and potential content-based intrusion detection. High packet rates of very small packets increase the CPU usage. To ensure other important tasks such as routing table calculations are provided with enough runtime, the scheduling priority of the unprivileged OpenLIDS process, which performs the most resource intensive operations, could be decreased.

Memory Resources

In terms of memory consumption OpenLIDS is very efficient. The binary size is only 75 KiB. Reading the *Resident Set Size (RSS)* from `/proc/$PID/statm` and summing the values of both processes up after OpenLIDS was started showed

a usage of 756 KiB of RAM. Replaying Wray trace 14 which caused several detail checks increased the memory usage to 792 KiB. The memory usage highly depends on the number of observed hosts, detail checks and number of tracked connections in user space. All values can be limited in the configuration file to prevent OpenLIDS from using up all available memory. The absolute memory usage can also be limited with the *pam_limits* module, if *Pluggable Authentication Modules (PAM)* are available. This is not the case on OpenWRT by default.

6.5 Security

Using OpenLIDS shall not give additional attack surface (Requirement 6). This section discusses the security implications of using OpenLIDS in respect to an attacker tricking automated response for a benign user (Section 6.5.1) and the safety of a router running OpenLIDS (Section 6.5.2).

6.5.1 Automated Response

Intrusion detection in combination with automated response is dangerous because it could be misused by an attacker to cause DoS conditions for legitimate users. OpenLIDS is vulnerable to be misused to deny access to legitimate users with two classes of attacks: forged reject messages can be sent to a user such that they are counted as failed connection attempts or the source IP address can be spoofed to fake malicious traffic to come from a benign user's IP address.

Counting arbitrary ICMP destination unreachable messages and TCP reset packets can be prevented with stateful packet inspection. With SPI the attacker must be aware of existing connections of the target. Packets not corresponding to existing connections will be filtered and not analysed by OpenLIDS (see Appendix B). The attacker can still sniff the network to obtain information about existing connections and then send reject packets matching these connections. Sending TCP reset packets will lead to a connection abortion and the connection tracking entry is removed. It will not be possible to forge a high number of TCP reset packets to increase the failed connection attempt counters of the target. ICMP unreachable messages on the other hand do not cause the connection tracking entry to be removed. Sending a high number of ICMP unreachable messages matching an existing connection can be used to trigger remediation against a non-existing DoS attacks which would corrupt communication from the client to the endpoint of the attacked connection. If a connection to the user's DNS server or even all of his connections are attacked, this leads to severe communication disruption. If the attacker has the ability to send matching reject packets of existing connections, she also has the ability to disrupt these connections without misusing OpenLIDS. As remediative actions last for 3 hours, or one day if the client is identified by it's MAC address, the result of communication disruptive attacks misusing OpenLIDS become much more persistent. The attacker can stop the attack once OpenLIDS starts to block the target.

In the second class of attacks, an attacker forges the IP address of a benign user to send traffic that will be identified as malicious. If packets are routed through a mesh box running OpenLIDS they will be accounted to the faked

source instead to the real sender. If the attacker is attacking an IP address in a client network, she has to be part of the same network due to packet filters. A packet with an IP address of a client is only allowed to come in on the network interface of the client network. Client networks can have IP spoofing detection enabled. Because a host is allowed to use up to three IP addresses in 5 minutes, the IP spoofing might not be detected. For IP spoofing detection, hosts are observed by their MAC address. As long as the attacker does not modify the hardware address of her network card, the traffic will not be accounted to the target but to the attacker herself. The spoofing attack will not work in this case. If the MAC address is modified it will be impossible to defeat the attack, but it will require the attackers local presence. If an IP address in the wireless mesh network and not an IP address in a client network is attacked, the attack is more difficult to defeat. Here it is not possible to observe MAC addresses of clients and packet filters cannot defeat spoofing of IP addresses from the mesh network's IP address range. In an ideal case every mesh box runs OpenLIDS and only client networks have to be monitored. In this situation IP spoofing attacks to trick OpenLIDS to respond are not a big problem as discussed above. More generally the whole wireless mesh network address range will be monitored and OpenLIDS is vulnerable to spoofing attacks.

Packet filters, stateful packet inspection and observation of clients by their MAC address help to defeat misuse of OpenLIDS to cause DoS conditions, but not in all cases. Using OpenLIDS increases the risk of DoS attacks against clients in the network. Requirement 6 cannot be fulfilled completely. The risk of misuse attacks in an open network has to be compared to the risk which is posed by malicious traffic.

6.5.2 Application Security

OpenLIDS processes user-generated network data. Improper data handling could lead to exploitable vulnerabilities. Care has been taken not to make any assumptions about data inputs and the data lengths are validated. Still it might be possible that vulnerabilities exist in OpenLIDS or its underlaying libraries. For this reason the process performing packet decoding and anomaly detection runs with reduced privileges. This will limit the impact of a potential exploit. The mitigation process running with root privileges has only a limited set of clearly defined data inputs.

Valgrind [86] was used to check OpenLIDS for programming errors. Several errors in *libc6* were detected. There were memory reads of invalid size and small memory leaks of constant size caused by `__nss_database_lookup()` which is indirectly called from OpenLIDS by calling `getpwnam()`. These errors are known and are not severe[1]. No other errors were detected in OpenLIDS or its libraries.

[1] `https://bugs.launchpad.net/ubuntu/+source/glibc/+bug/59449`

Chapter 7

Summary

7.1 Conclusions

This work addresses the problem of excessive malicious traffic which causes congestion in wireless mesh networks. The approach of this work is to use existing hardware – the mesh boxes – to detect an block malicious flows and reduce congestion on the wireless mesh network and on the Internet uplinks to protect the quality of service. A common wireless router which could be used as a mesh box in a WMN was shown to have constrained hardware resources which limit the operations that can be performed. Existing intrusion detection systems – Snort and Bro – were shown to have poor performance on this device under high traffic load. They are not lightweight enough, even if Snort is referred to as a lightweight intrusion detection system and Bro was used to perform cheap detection algorithms.

As a consequence of these performance problems, a dynamic load dependent architecture for intrusion detection was proposed. A system using this architecture adopts the set of detection algorithms depending on the available CPU resources. During lower traffic loads, more expensive detection metrics are included which allows to cover the widest possible range of attacks with the available resources. Content based detection approaches e.g. require capturing of packets at their full length and require TCP stream reassembly to prevent detection evasion. This was shown to be more resource consuming than capturing packet headers and performing anomaly based intrusion detection.

A novel anomaly-based lightweight intrusion detection system was implemented. OpenLIDS is able to detect a wide range of malicious traffic and automatically blocks detected malicious flows in a fine-grained way. Even under high traffic volumes it does not consume a lot of resources. To the best of our knowledge, no prior systems exist that have these characteristics, and it therefore builds an important missing block in the proposed intrusion detection architecture. Apart from detecting malicious traffic, OpenLIDS protects the operation of the router during attacks that involve a high rate of new connections. Such attacks cause a high load on the connection tracking system of the Linux kernel and severely affect the router's operational tasks and its forwarding capacity.

Anomaly-based detection metrics can be used to identify a wide range of

attacks. Because automated response requires detailed information about the malicious flows and a high certainty about a malicious cause, several common anomaly detection metrics cannot be used here. The detection algorithms based on failed connection attempts provide the required information and have a low rate of false positives. It is, however, not possible to detect all kind of malicious traffic. Attacks which do not exhibit anomalous behaviour on lower layers cannot be detected. This includes non-scanning worm propagation or UDP denial-of-service floods which use fixed source and destination ports. Both attacks show similar characteristics to benign traffic. Complementing the detection algorithms with content-based approaches allows to extend the range of detected attacks and leads to a better protection of the network, if sufficient resources are available.

OpenLIDS has been tested with real network traces from the Wray community WMN and different peer-to-peer applications. No false positives occurred during these tests. P2P applications showed the highest risk of false positives. If they behave too aggressively and use fixed ports such as BBC's iPlayer, they could potentially trigger remediation. False positives are less likely for aMule and very unlikely for BitTorrent. Except the usage of network diagnosis tools such as Nmap, no other benign traffic showed a high risk of causing a violation.

It was required that an intrusion prevention system with automated response shall not give any attack surface to be misused to cause remediative actions against benign hosts. The use of stateful inspection, identification of hosts by their MAC address and specification of the networks to be observed allows to defeat some attacks but not all. It was therefore not possible to perfectly fulfil this requirement and using OpenLIDS with automated response poses additional risks which have to be considered.

7.2 Future Work

In this work it has been assumed that malicious traffic has a significant impact on the quality of service of wireless mesh networks. Previous work analysed the impact of malware infected hosts on traditional wireless infrastructure networks and higher round-trip times, higher packet drop rates and a reduction of the sending rate were observed due to congestion. The effect of excessive malicious traffic in wireless mesh networks has still to be quantified by measurements or simulation. The results of this quantification together with research on the frequency of excessive malicious traffic in common wireless mesh networks will allow to estimate the risk of malicious traffic. This is important as running an intrusion prevention system which replies automatically to events constitutes a risk itself.

It was shown that it is feasible to deploy low cost detection metrics on common wireless mesh boxes even under high traffic volumes. The dynamic load dependent architecture for intrusion detection was proposed to extend the set of detection metrics to more expensive ones, if sufficient resources are available. An IDS using this architecture needs to be implemented and tested. More research on low cost detection metrics is required to extend the set of detectable attacks on the low-cost level. These metrics are required to have a low rate of false positives and to allow identification of malicious flows in a fine-grained way.

Every wireless mesh box running OpenLIDS performs intrusion detection

on its own and independent of other instances of OpenLIDS on other mesh boxes. Collaborative intrusion detection could lead to higher efficiency and better detection. Single hosts are monitored by multiple instances, if packets are sent on a path on which several routers run OpenLIDS and these are configured to observe more than just their own client networks. One possibility to get better performance would be to implement a hand-over algorithm that negotiates which mesh box has to monitor a certain host. This is ideally the closest instance of OpenLIDS to this host.

OpenLIDS has only been tested with a limited set of network traces, applications and artificially generated malicious traffic. More excessive long-term tests are required to get more certainty about false positives and detection of malicious traffic. These tests have to be carried out in wireless mesh networks, but tests in wired networks will be usefull as well.

Netfilter ULOG and NFLOG are supported as packet capturing mechanisms. NFLOG could not be tested on the router used in this work because of portability problems. NFLOG needs to be tested, if it should be used on other devices.

Support for IPv6 was prepared but not fully implemented because ULOG only supports IPv4. IPv6 specific functions should be implemented to allow the use of OpenLIDS in networks which already support IPv6.

7.3 Recommendations

To protect the network optimally against attacks the following recommendations should be followed:

- Deploy OpenLIDS on as many mesh boxes as possible
- Use kernel space connection tracking for performance reasons if possible
- Filter impossible source addresses to defeat IP spoofing
- Use stateful packet inspection and drop packet with invalid states to block certain types of scanning and protect OpenLIDS from misuse
- Activate IP spoofing detection for client networks to detect IP spoofing and protect OpenLIDS from misuse
- Disable reply to ICMP broadcast packets to prevent being used as a reflector in a Smurf attack

7.4 Other Contributions

Apart from gaining a better understanding of the possibilities and limitations of common wireless mesh boxes and implementing a dedicated lightweight intrusion detection system for wireless mesh networks, this work resulted in other achievements by:

- Presenting intermediate results at the Multi Service Networks worksop 2008
- Contributing to an IEEE workshop paper [40]

- Contributing to the EU FP7 INTERSECTION project on intrusion tolerance systems
- Discovering and fixing two bugs in the Linux kernel and contributing to two performance optimisation patches
- Fixing portability issues in Netfilter libraries
- Filing bug reports, fixing bugs, upgrading existing packages and providing new packages for OpenWRT
- Filing a bug report on the MadWifi wireless drivers

Appendix A

Installation Guide

A.1 Installation

A.1.1 OpenWRT

1. Checkout a copy of OpenWRT

```
svn co https://svn.openwrt.org/openwrt/trunk/ trunk
cd trunk
```

2. Checkout additional packages

```
scripts/feeds update
```

3. Install the packages of this work, if they are not included in OpenWRT

```
cp -r $CD/openwrt/packages/lids ./package/
cp -r $CD/openlids ./package/lids/lids
cp -r $CD/openwrt/packages/libnetfilter-conntrack \
  ./package/
cp -r $CD/openwrt/packages/libnetfilter-log \
  ./package/
cp -r $CD/openwrt/packages/libsmtp-- ./package/
cp -r $CD/openwrt/kernel_packages/netlink.mk \
  ./package/kernel/modules/
```

4. Activate additional packages

```
scripts/feeds install libnfnetlink
scripts/feeds install libintl
```

5. Find out which kernel version (e.g. 2.6.26) is used for your architecture (e.g. ixp4xx)

```
ARCH=ixp4xx
KVERS=‘grep LINUX_VERSION \
  target/linux/$ARCH/Makefile | \
  sed -r ’s/.*=([0-9]+\.[0-9]+\.[0-9]+).*/\1/’‘
echo $KVERS
```

6. Install kernel patches (read `$CD/openwrt/kernel_patches/readme.txt` for detailed information)

```
PATCHES=$CD/openwrt/kernel_patches
PATCHDIR=target/linux/generic-2.6/patches-$KVERS/
cp $PATCHES/500-ulog_mac.patch $PATCHDIR
cp $PATCHES/501-conntrack_status.patch $PATCHDIR
cp $PATCHES/503-nf_ct_kill.patch $PATCHDIR
cp $PATCHES/504-nf_ct_kill_acct.patch $PATCHDIR
cp $PATCHES/505-nf_ct_seen_related.patch $PATCHDIR
cp $PATCHES/506-nf_ct_remove_lock.patch $PATCHDIR
cp $PATCHES/507-nf_ct_opt_alloc.patch $PATCHDIR
```

7. Configure the installation for your router and include OpenLIDS

```
make menuconfig
```

   Select Network > lids: y

8. Compile OpenWRT

```
make
```

9. Install OpenWRT on your router according to the documentation

A.1.2 Debian

These instructions are for Debian Lenny. Installation on other releases would vary because of different kernel and libary versions. The Netfilter libraries on Debian Etch e.g. are too old.

1. Apply kernel patches if possible (read `$CD/openwrt/kernel_patches/readme.txt` for a detailed description of the patches and information on which patches should be applied)

```
PATCHES=$CD/openwrt/kernel_patches
cd /usr/src/linux
patch -p1 < $PATCHES/501-conntrack_status.patch
patch -p1 < $PATCHES/503-nf_ct_kill.patch
patch -p1 < $PATCHES/504-nf_ct_kill_acct.patch
patch -p1 < $PATCHES/505-nf_ct_seen_related.patch
```

2. Make sure the following kernel options are built-in or loaded as modules:
   - IP_NF_TARGET_ULOG
   - NETFILTER_NETLINK_LOG
   - NETFILTER_XT_TARGET_NFLOG
   - NF_CONNTRACK
   - NF_CONNTRACK_IPV4
   - NF_CT_NETLINK
   - NF_CONNTRACK_EVENTS

- NETFILTER_NETLINK
- NETFILTER_XT_MATCH_LENGTH
- NETFILTER_XT_MATCH_LIMIT
- NETFILTER_XT_MATCH_MAC
- NETFILTER_XT_MATCH_MULTIPORT
- IP_NF_RAW

3. Recompile and install the kernel
4. Install Netfilter libraries

```
sudo aptitude install libnfnetlink-dev \
  libnetfilter-log-dev libnetfilter-conntrack-dev
```

5. Install libsmtp--
   (a) Download from `http://libsmtp.sourceforge.net`
   (b) Extract archive

```
tar -jxf libsmtp---0.1.0.tar.bz2
cd libsmtp---0.1.0
```

   (c) Fix a bug if the library is compiled without SSL support

```
grep -v '@strip@ \*\.so' \
  src/plugins/Makefile.in > \
  src/plugins/Makefile.in.new
mv src/plugins/Makefile.in.new \
  src/plugins/Makefile.in
```

   (d) Compile and install

```
./configure --disable-ssl
make
sudo make install
```

6. Compile and install OpenLIDS

```
cp -r $CD/openlids .
cd openlids
make
sudo make install
```

A.2 Configuration

1. Consider reducing the connection tracking timeout for TCP connections in state SYN SENT by adding the following line to `/etc/sysctl.conf`:

```
net.netfilter.nf_conntrack_tcp_timeout_syn_sent=30
```

2. Set up Iptables rules as described in Appendix B

3. Modify configuration file `/etc/lids.conf`

   - Set user and group OpenLIDS runs as (options `user` and `group`)
   - Set log destination (e.g. `logfile=/var/log/lids`)
   - Set log level
   - Specify networks that should be monitored (option `observe_net`)
   - Specify networks for which DNS queries should be monitored (option `observe_dns`)
   - Specify in which networks hosts should be identified by their MAC address (option `observe_mac`)
   - Disable active mitigation for testing by setting `enable_mitigation=0`
   - Have a look at the other configuration options and configure OpenLIDS according to your needs

Start OpenLIDS with `lids` or run it in background with `lids -d`. Display a short help screen with `lids --help`.

Appendix B

Iptables Script

The user is responsible for setting up Iptables rules which are required to run OpenLIDS and which help to defeat certain attacks. This appendix gives advice on important aspects. An example Iptables script is shown in Listing B.1.

B.1 Setup

The Iptables script presented here is designed for a wireless mesh box with three network interfaces as shown in Figure B.1. The router has a wireless network interface to the wireless mesh network (ath0) and another to provide access to the network to client computers (ath1). It further has a wired network interface (eth0) which is connected to the Internet over a broadband connection. This connection is optional. If it is not available, traffic from clients will be routed to another Internet uplink point in the WMN. In this case addresses from the client network 192.168.1.0/24 will be translated into the router's address in the mesh network 172.16.0.0/16. NAT is applied to all packets leaving the public interface eth0.

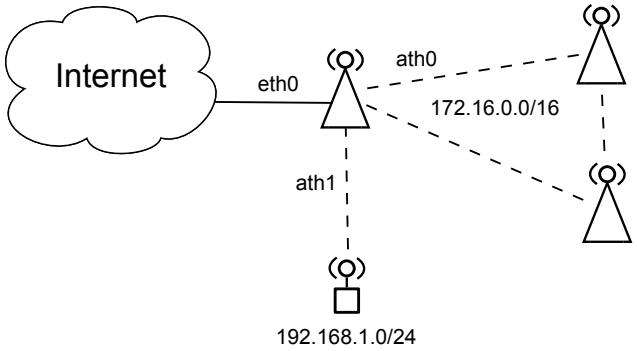


Figure B.1: Network setup for the described Iptables script

B.2 Script Description

OpenLIDS Spoofing Protection

In Section 6.5.1 it was described how attackers can misuse OpenLIDS to cause denial-of-service conditions against benign users by spoofing their IP address. It is important to only allow packets with source IP addresses from the wireless mesh network or the client network to arrive on the corresponding network interfaces. To prevent connection tracking events from being generated for spoofed addresses, this should be done in the Netfilter raw table.

The *dropspoof* chain (Line 23) enables this kind of protection by dropping packets with invalid source addresses. All incoming packets are sent through this chain from the PREROUTING chain in the raw table (Line 51). Because other routers can act as Internet uplinks, it is possible that packets coming in on the wireless mesh network interface have a public source IP address. The rules require that if a packet has an IP address from 172.16.0.0/16, it must come in on ath0.

OpenLIDS Mitigation Chain

The mitigation chain *lidsmitig* (Line 29) is used by OpenLIDS to place filter rules to block or throttle hosts which cause a violation. To be effective against flow storms and to prevent connection tracking events from being generated, this chain should be in the raw table. Its name can be specified with the OpenLIDS configuration option `chain` from the mitigation configuration section. This chain has to be created by the user and it has to be referenced from the PREROUTING chain in the raw table (Line 51). It will be automatically flushed by OpenLIDS, so no rules should be placed in this chain manually.

Stateful Packet Filtering

To prevent TCP scanning techniques which do not use normal SYN packets and to protect OpenLIDS from attacks against benign users by sending arbitrary reject messages (Section 6.5.1) it is recommended to use stateful packet inspection. The *dropinval* chain (Line 31) drops all packets with illegal connection states. These are TCP packets corresponding to a new connection without having the SYN flag set and the ACK flag cleared, ICMP packets which are not echo requests and are not related to an existing connection and packets with invalid checksums. To prevent OpenLIDS from analysing bogus packets, the *dropinval* chain is referenced from the FORWARD, INPUT and OUTPUT chain before packets are captured.

OpenLIDS Packet Capturing

The *lidslog* chain (Line 42) causes packets to be logged with Netfilter ULOG. Parameter `--ulog-qthreshold 50` causes up to 50 packets to be queued in one multipart message. DNS queries are captured at their full length to be able to analyse the query type. Only up to the first 92 B are captured for other packets (`--ulog-cprange 92`).

The FORWARD, INPUT and OUTPUT chain reference the *lidslog* chain. Not only incoming packets but also outgoing ones are captured. It is possible,

that the meshbox directly sends ICMP unreachable messages to a client, if no route exists to a requested destination.

Packet Forwarding

Here no outgoing communication in the wireless networks is blocked. Only new connections coming from the Internet are blocked. To prevent IP spoofing, the source addresses are verified to be from a reasonable network range. Additional rules for DHCP will have to be specified.

Listing B.1: Iptables Script

```
#!/bin/sh

IPT=/sbin/tables

# mesh network
NETATH0="172.16.0.0/16"
# client network
NETATH1="192.168.1.0/24"

### reset rules
# default policies
$IPT -P FORWARD DROP
$IPT -P INPUT DROP
$IPT -P OUTPUT DROP
# delete all rules and chains
$IPT -F
$IPT -t raw -F
$IPT -t nat -F
$IPT -X
$IPT -t raw -X

### chain definitions
# dropspoof chain: drops packets with a spoofed address
$IPT -t raw -N dropspoof
$IPT -t raw -A dropspoof -i ! ath0 -s ${NETATH0} \
    -j DROP
$IPT -t raw -A dropspoof -i ! ath1 -s ${NETATH1} \
    -j DROP
# lidsmitig chain: used to place automated response
$IPT -t raw -N lidsmitig
# dropinval chain: drops packets with invalid state
$IPT -N dropinval
$IPT -A dropinval -m state \
    --state ESTABLISHED,RELATED -j RETURN
$IPT -A dropinval -p tcp ! --syn -m state \
    --state NEW -j DROP
$IPT -A dropinval -p icmp --icmp-type \
    ! echo-request -m state --state NEW -j DROP
$IPT -A dropinval -m state --state NEW -j RETURN
```

```
40 $IPT -A dropinval -m state --state INVALID -j DROP
41 $IPT -A dropinval -j DROP
42 # lidslog chain: exports packets to lids
43 $IPT -N lidslog
44 $IPT -A lidslog -p udp --dport 53 -j ULOG \
45     --ulog-qthreshold 50
46 $IPT -A lidslog -p udp --dport 53 -j RETURN
47 $IPT -A lidslog -j ULOG --ulog-cprange 92 \
48     --ulog-qthreshold 50
49
50 ### special tables
51 # RAW table (before conntrack)
52 $IPT -t raw -A PREROUTING -j dropspoof
53 $IPT -t raw -A PREROUTING -j lidsmitig
54 # NAT
55 $IPT -t nat -A POSTROUTING -o eth0 -j MASQUERADE
56 $IPT -t nat -A POSTROUTING -o ath0 -s ${NETATH1} \
57     -j MASQUERADE
58
59 ### main chains
60
61 # forward chain
62 $IPT -A FORWARD -j dropinval
63 $IPT -A FORWARD -j lidslog
64 $IPT -A FORWARD -m state \
65     --state ESTABLISHED,RELATED -j ACCEPT
66 # mesh forwarding
67 $IPT -A FORWARD -i ath0 -o ath0 -s ${NETATH0} \
68     -j ACCEPT
69 # mesh to internet
70 $IPT -A FORWARD -i ath0 -o eth0 -s ${NETATH0} \
71     -j ACCEPT
72 # wireless clients to everywhere
73 $IPT -A FORWARD -i ath1 -s ${NETATH1} -j ACCEPT
74
75 # input chain
76 $IPT -A INPUT -j dropinval
77 $IPT -A INPUT -j lidslog
78 $IPT -A INPUT -m state \
79     --state ESTABLISHED,RELATED -j ACCEPT
80 $IPT -A INPUT -i lo -j ACCEPT
81 $IPT -A INPUT -i ath0 -s ${NETATH0} -j ACCEPT
82 $IPT -A INPUT -i ath1 -s ${NETATH1} -j ACCEPT
83 $IPT -A INPUT -i eth0 -p tcp --dport 22 -j ACCEPT
84
85 # output chain
86 $IPT -A OUTPUT -j dropinval
87 $IPT -A OUTPUT -j lidslog
88 $IPT -A OUTPUT -m state \
89     --state ESTABLISHED,RELATED -j ACCEPT
```

```
90 $IPT -A OUTPUT -o lo -j ACCEPT
91 $IPT -A OUTPUT -o ath0 -j ACCEPT
92 $IPT -A OUTPUT -o ath1 -j ACCEPT
93 $IPT -A OUTPUT -o eth0 -j ACCEPT
94
95 ### reset counters
96 $IPT -Z
97 $IPT -t raw -Z
98 $IPT -t nat -Z
```

Bibliography

[1] Imad Aad, Jean-Pierre Hubaux, and Edward W. Knightly. Denial of Service Resilience in Ad Hoc Networks. In *MobiCom '04: Proceedings of the 10th annual international conference on Mobile computing and networking*, pages 202–215, New York, NY, USA, 2004. ACM.

[2] Steven Adair. Georgian Attacks: Remember Estonia? `http://www.shadowserver.org/wiki/pmwiki.php?n=Calendar.20080813`, August 2008.

[3] Air-Stream Community Wireless Network. `http://www.air-stream.org.au`, April 2008.

[4] aMule. `http://www.amule.org`, August 2008.

[5] S. Antonatos, P. Akritidis, E.P. Markatos, and K.G. Anagnostakis. Defending against hitlist worms using network address space randomization. *Computer Networks*, 51(12):3471–3490, 2007.

[6] Pablo Neira Ayuso. Netfilter's connection tracking system. *LOGIN;, The USENIX magazine*, 32(3):34–39, 2006.

[7] F. Baker. RFC 1812: Requirements for IP Version 4 Routers, 1995.

[8] BBC iPlayer. `http://www.bbc.co.uk/iplayer/`, August 2008.

[9] John Bellardo and Stefan Savage. 802.11 Denial-of-Service Attacks: Real Vulnerabilities and Practical Solutions. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 2–2, Berkeley, CA, USA, 2003. USENIX Association.

[10] L. Bononi and C. Tacconi. A Wireless Intrusion Detection System for Secure Clustering and Routing in Ad Hoc Networks. *Proceedings of the 9th International Information Security Conference*, pages 398–414, 2006.

[11] Daniela Brauckhoff, Bernhard Tellenbach, Arno Wagner, Marting May, and Anukool Lakhina. Impact of Packet Sampling on Anomaly Detection Metrics. *Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 159–164, 2006.

[12] Mariusz Burdach. Hardening the TCP/IP stack to SYN attacks. `http://www.securityfocus.com/infocus/1729`, Mai 2008.

[13] Thomas M. Chen, Geng-Sheng Kuo, Zheng-Ping Li, and Guo-Mei Zhu. *Security in Wireless Mesh Networks*, chapter Intrusion Detection in Wireless Mesh Networks. CRC Press, 2007.

[14] T. Clausen and P. Jacquet. RFC 3626: Optimized Link State Routing Protocol (OLSR), 2003.

[15] Luca Deri. Improving Passive Packet Capture: Beyond Device Polling. *Proceedings of SANE*, 2004, 2004.

[16] Thomas Dübendorfer, Arno Wagner, Theus Hossmann, and Bernhard Plattner. Flow-Level Traffic Analysis of the Blaster and Sobig Worm Outbreaks in an Internet Backbone. *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2005.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, 1999.

[18] Freifunk. `http://www.freifunk.net`, April 2008.

[19] Ramakrishna Gummadi, David Wetherall, Ben Greenstein, and Srinivasan Seshan. Understanding and Mitigating the Impact of RF Interference on 802.11 Networks. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 385–396, New York, NY, USA, 2007. ACM.

[20] Chistoph Göldi and Roman Hiestand. Scan Detection Based Identification of Worm-Infected Hosts. Master's thesis, ETH Zurich, 2005.

[21] Warren Harrop and Grenville Armitage. Defining and Evaluating Greynets (Sparse Darknets). *Proceedings of the The IEEE Conference on Local Computer Networks 30th Anniversary (LCN'05)*, pages 344–350, 2005.

[22] L.T. Heberlein, G.V. Dias, K.N. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 296–304, 1990.

[23] M. Heusse, F. Rousseau, G. Berger-Sabbatel, and A. Duda. Performance anomaly of 802.11 b. *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, 2, 2003.

[24] Bert Hubert. `http://lartc.org`, August 2008.

[25] Jonathan Ishmael and Nicholas P. Race. Building a rural community mesh network. In *Broadband Europe*, Geneva, Switzerland, December 2006.

[26] Amit P. Jardosh, Krishna N. Ramachandran, Kevin C. Almeroth, and Elizabeth M. Belding-Royer. Understanding Congestion in IEEE 802.11b Wireless Networks. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet measurement*, pages 1–14, New York, NY, USA, 2005. ACM.

[27] Robert Jenkins. Hash Functions for Hash Table Lookup. *Dobb's Journal*, pages 107–109, 1997.

[28] Rick Jones. Netperf. `http://www.netperf.org`, Mai 2008.

[29] Jaeyeon Jung, Stuart E. Schechter, and Arthur W. Berger. Fast Detection of Scanning Worm Infections. *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 59–81, 2004.

[30] A. Kamerman and L. Monteban. WaveLAN®-II: A high-performance wireless LAN for the unlicensed band: Wireless. *Bell Labs technical journal*, 2 (3):118–133, 1997.

[31] Chris Karlof and David Wagner. Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures. *Ad Hoc Networks*, 1(2-3):293–315, 2003.

[32] J. Klensin. RFC 2821: Simple Mail Transfer Protocol, 2001.

[33] Dimitrios Koutsonikolas, Jagadeesh Dyaberi, Prashant Garimella, Sonia Fahmy, and Y. Charlie Hu. On TCP Throughput and Window Size in a Multihop Wireless Network Testbed. *Proceedings of the the second ACM international workshop on Wireless network testbeds, experimental evaluation and characterization*, pages 51–58, 2007.

[34] Abhishek Kumar, Vern Paxson, and Nicholas Weaver. Exploiting Underlying Structure for Detailed Reconstruction of an Internet-scale Event. *Proc. ACM IMC*, 2005.

[35] Robert Lemos. Admins warned of brute-force SSH attacks. `http://www.securityfocus.com/news/11518`, Mai 2008.

[36] libpcap: Packet Capture Library. `http://www.tcpdump.org`, April 2008.

[37] libsmtp++/libsmtp−−. `http://libsmtp.sourceforge.net`, August 2008.

[38] Locustworld. `http://www.locustworld.com`, April 2008.

[39] MadWifi. `http://www.madwifi.org`, Mai 2008.

[40] Dwight Makaroff, Paul Smith, Nicholas J. P. Race, and David Hutchison. Intrusion Detection Systems for Community Wireless Mesh Networks. *Second IEEE International Workshop on Enabling Technologies and Standards for Wireless Mesh Networking. MeshTech'08*, September 2008.

[41] Sergio Marti, T. J. Giuli, Kevin Lai, and Mary Baker. Mitigating Routing Misbehavior in Mobile Ad Hoc Networks. In *Mobile Computing and Networking*, pages 255–265, 2000.

[42] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, 258:263, 2002.

[43] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. *Proc. Winter'93 USENIX Conference*, 1993.

[44] Microsoft Security Bulletin MS05-021: Vulnerability in Exchange Server Could Allow Remote Code Execution. `http://www.microsoft.com/technet/security/bulletin/MS05-021.mspx`, May 2008.

[45] Andrew Miklas. LKML: Linksys WRT54G and the GPL. http://lkml.org/lkml/2003/6/7/164, Mai 2008.

[46] MIT Roofnet. `http://pdos.csail.mit.edu/roofnet/`, April 2008.

[47] P. Mockapetris. RFC 1035: Domain Names - Implementation and Specification, 1987.

[48] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Slammer Worm Dissection: Inside the Slammer Worm. *IEEE Security & Privacy*, 1(4):33–39, 2003.

[49] Y. Musashi, R. Matsuba, and K. Sugitani. Indirect Detection of Mass Mailing Worm-Infected PC terminals for Learners. *Proc. ICETA2004*, 2004.

[50] Netfilter libnetfilter_conntrack. `http://www.netfilter.org/projects/libnetfilter_conntrack/`, June 2008.

[51] Netfilter libnetfilter_log. `http://www.netfilter.org/projects/libnetfilter_log/`, June 2008.

[52] Netfilter libnfnetlink. `http://www.netfilter.org/projects/libnfnetlink/`, June 2008.

[53] Netfilter Project. `http://www.netfilter.org`, Mai 2008.

[54] Nmap – Free Security Scanner for Network Exploration and Security Audits. `http://nmap.org`, August 2008.

[55] NYCwireless. `http://www.nycwireless.net`, April 2008.

[56] Open Wireless. `http://www.openwireless.ch`, April 2008.

[57] OpenWRT. `http://www.openwrt.org`, April 2008.

[58] Oprofile – A System Profiler for Linux. `http://oprofile.sourceforge.net`, July 2008.

[59] A. Patwardhan, J. Parker, A. Joshi, M. Iorga, and T. Karygiannis. Secure Routing and Intrusion Detection in Ad Hoc Networks. *Third IEEE International Conference on Pervasive Computing and Communications, Kauaii Island, Hawaii, March*, pages 8–12, 2005.

[60] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Proceedings of the 7th conference on USENIX Security Symposium, 1998-Volume 7 table of contents*, pages 3–3, 1998.

[61] C. Perkins, E. Belding-Royer, and S. Das. RFC 3561: Ad hoc On-Demand Distance Vector (AODV) Routing, 2003.

[62] J. Postel. RFC 768: User Datagram Protocol, 1980.

[63] J. Postel. RFC 792: Internet Control Message Protocol, 1981.

[64] J. Postel. RFC 793: Transmission Control Protocol, 1981.

[65] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The Bittorrent P2P File-Sharing System: Measurements and Analysis. *LECTURE NOTES IN COMPUTER SCIENCE*, 3640:205, 2005.

[66] Procps - The /proc file system utilities. `http://procps.sourceforge.net`, Mai 2008.

[67] Svetlana Radosavac, Nassir Benammar, and John S. Baras. Cross-layer attacks in wireless ad hoc networks. In *2004 Conference on Information Sciences and Systems*, 2004.

[68] Martin Roesch. Snort-Lightweight Intrusion Detection for Networks. *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, 1999.

[69] Rusty Russell. Linux 2.4 Packet Filtering HOWTO. `http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html`, Mai 2008.

[70] Rusty Russell and Harald Welte. Linux netfilter Hacking HOWTO. `http://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html`, Mai 2008.

[71] H. Schulzrinne, A. Rao, and R. Lanphier. RFC 2326: Real Time Streaming Protocol (RTSP), 1998.

[72] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobso. RFC 3350: RTP: A Transport Protocol for Real-Time Applications, 2003.

[73] Seattle Wireless. `http://www.seattlewireless.net`, April 2008.

[74] Stanislav Shalunov. `http://shlang.com/thrulay/`, August 2008.

[75] Colleen Shannon and David Moore. The Spread of the Witty Worm. *IEEE Security and Privacy*, 2(4), 2004.

[76] Skype. `http://www.skype.com`, August 2008.

[77] R. Sombrutzki, A. Zubow, M. Kurth, and J.-P. Redich. Self-Organization in Community Mesh Networks The Berlin Roofnet. In *Operator-Assisted (Wireless Mesh) Community Networks*, pages 1–11, Berlin, Germany, September 2006.

[78] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in Your Spare Time. *Proceedings of the 11th USENIX Security Symposium*, 7, 2002.

[79] Joe Stewart. Storm Worm DDoS Attack. http://www.secureworks.com/research/threats/storm-worm, Mail 2008.

[80] Brett Stone-Gross, Christo Wilson, Kevin Almeroth, Elizabeth Belding, Heather Zheng, and Konstantina Papagiannaki. Malware in IEEE 802.11 Wireless Networks. In *Proceedings of the Passive and Active Measurement Conference (PAM)*, 2008.

[81] Tcpdump. http://www.tcpdump.org, April 2008.

[82] Tcpreplay: Pcap editing and replay tools for *NIX. http://tcpreplay.synfin.net, April 2008.

[83] F. Tobagi and L. Kleinrock. Packet Switching in Radio Channels: Part II–The Hidden Terminal Problem in Carrier Sense Multiple-Access and the Busy-Tone Solution. *Communications, IEEE Transactions on [legacy, pre-1988]*, 23(12):1417–1433, 1975.

[84] Traceroute. http://dmitry.butskoy.name/traceroute/, August 2008.

[85] Transmission. http://www.transmissionbt.com, August 2008.

[86] Valgrind. http://www.valgrind.org, August 2008.

[87] Arno Wagner and Bernhard Plattner. Entropy based worm and anomaly detection in fast IP networks. *Enabling Technologies: Infrastructure for Collaborative Enterprise, 2005. 14th IEEE International Workshops on*, pages 172–177, 2005.

[88] Arno Wagner, Thomas Dubendorfer, Roman Hiestand, Christoph Göldi, and Bernhard Plattner. A Fast Worm Scan Detection Tool for VPN Congestion Avoidance. *LECTURE NOTES IN COMPUTER SCIENCE*, 4064: 181, 2006.

[89] Y. Waizumi, M. Tsuji, H. Tsunoda, N. Ansari, and Y. Nemoto. Distributed Early Worm Detection Based on Payload Histograms. *IEEE International Conference on Communications. ICC'07*, pages 1404–1408, 2007.

[90] Harald Welte. The netfilter.org "ulogd" project. http://www.netfilter.org/projects/ulogd/, Mai 2008.

[91] Wireshark. http://www.wireshark.org, August 2008.

[92] C. Wong, S. Bielski, J.M. McCune, and C. Wang. A Study of Mass-mailing Worms. *Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 1–10, 2004.

[93] Phil Wood. libpcap-mmap: Memory Mapped Packet Capture Library. http://public.lanl.gov/cpw/, April 2008.

[94] Wray Community Wireless Mesh Network. http://wray.lancs.ac.uk, April 2008.

[95] Wenyuan Xu, Wade Trappe, Yanyong Zhang, and Timothy Wood. The Feasibility of Launching and Detecting Jamming Attacks in Wireless Networks. In *MobiHoc '05: Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*, pages 46–57, New York, NY, USA, 2005. ACM.

[96] Atsushi Yoshioka and Min Sik Kim. Traffic-Aware Packet Matching for Intrusion Detection Systems. *Fourth International Conference on Broadband Communications, Networks and Systems. BROADNETS 2007*, pages 309–310, 2007.