OpenLIDS: A Lightweight Intrusion Detection System for Wireless Mesh Networks

Fabian Hugelshofer
f.hugelshofer@lancs.ac.uk

Paul Smith
p.smith@comp.lancs.ac.uk

David Hutchison
dh@comp.lancs.ac.uk

Nicholas J.P. Race
race@comp.lancs.ac.uk

Computing Department
Lancaster University
Lancaster, LA1 4WA, United Kingdom

ABSTRACT

Wireless mesh networks are being used to provide Internet access in a cost efficient manner. Typically, consumer-level wireless access points with modified software are used to route traffic to potentially multiple back-haul points. Malware infected computers generate malicious traffic, which uses valuable network resources and puts other systems at risk. Intrusion detection systems can be used to detect such activity. Cost constraints and the decentralised nature of WMNs make performing intrusion detection on mesh devices desirable. However, these devices are typically resource constrained. This paper describes the results of examining their ability to perform intrusion detection. Our experimental study shows that commonly-used deep packet inspection approaches are unreliable on such hardware. We implement a set of lightweight anomaly detection mechanisms as part of an intrusion detection system, called OpenLIDS. We show that even with the limited hardware resources of a mesh device, it can detect current malware behaviour in an efficient way.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*

General Terms

Experimentation, Measurement, Performance, Security

Keywords

Wireless mesh network, intrusion detection, performance, resource constrained devices

1. INTRODUCTION

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. They are becoming increasingly popular as a way of providing Internet access in a cost efficient way [9, 18, 26]. In these deployments, clients connect to a mesh device, which routes their traffic via other wireless mesh routers in order to reach a mesh node with an Internet uplink. WMNs will typically use IEEE 802.11 to communicate between the wireless mesh routers, and a subset of these routers will have connectivity to the wider Internet.

In many cases, WMNs are operated by the community that uses them, and the hardware for the mesh devices are consumer-level wireless access points that run a customised version of the Linux operating system. These devices are cost-effective, but are usually constrained in processing and memory resources. Cost-effectiveness in a community network is crucial, as more expensive hardware could prevent users from contributing.

By generating malicious traffic, malware infected computers use valuable network resources and put other systems at risk. A fundamental tool in defending against malicious behaviour is an Intrusion Detection System (IDS). In wired enterprise settings, an IDS is normally deployed on dedicated hardware at border points of the network. As Internet uplinks in a WMN can be decentralised, selecting a single location to deploy IDS functionality may not be possible. Performing intrusion detection at all Internet uplinks is undesirable, as users can arbitrarily choose to start sharing their Internet connection to the community, potentially leaving the network unprotected. Using dedicated detection systems at multiple locations in the network would be monetarily expensive. All this suggests that a decentralised approach using the existing hardware, the mesh devices, to perform intrusion detection is necessary.

In this paper, we address the question of whether it is possible to use typical WMN hardware for intrusion detection. To do this, we measured the overhead of performing common intrusion detection tasks on such a platform. We found that traffic pre-processing, a necessary step for intrusion detection systems that perform deep packet inspection, cannot be reliably carried out on a mesh device. Furthermore, signature-based detection techniques employed in IDSs, such as Snort, are not suitable because of memory and processing constraints. However, we did find that packet capturing and connection tracking are sufficiently efficient in most conditions.

In light of these findings, we implemented an intrusion detection system, called OpenLIDS. It uses efficient anomaly-based detection metrics to identify generic classes of attacks, including scanning, resource starvation attacks, and unsolicited email distribution caused by mass-mailing Internet worms. Furthermore, OpenLIDS collects statistics that can be used to determine the plausibility and severity of an attack, which can be used to inform the selection of appropriate remedial action, such as blocking or rate-limiting hosts. We demonstrate the lightweight detection metrics we use are sufficiently efficient for our target platform. Using collected traces of the recently discovered Conficker worm, we show that OpenLIDS is able to successfully detect malicious behaviour. We collected traffic from a WMN deployment, and demonstrated that OpenLIDS did not incorrectly generate intrusion alerts for benign traffic.

2. PERFORMANCE ANALYSIS

To understand the ability of a typical mesh device to conduct operations that are required by an IDS, we initially determined the base-line overhead of forwarding traffic and tracking connections. We then measured the cost associated with packet capturing, traffic pre-processing and pattern matching.

2.1 Experiment Setup

We examine the performance characteristics of a typical mesh device in a two-hop wireless test network. As an example of such a device, we used a Netgear WG302 wireless router. It has an Intel XScale IXP422 processor running at 266 MHz, 32 MB RAM, 8 MB flash memory and a wireless card with an Atheros 5212 chipset, using IEEE 802.11g. Comparable hardware devices are used in existing wireless networks [18, 26]. The operating system used is OpenWRT [19] – a stripped-down Linux distribution for embedded networking devices that is commonly deployed in WMNs. The Linux kernel has version 2.6.24.2, and Madwifi [14] drivers are used for the wireless card.

All measurements were performed in a university office environment. Devices were configured to operate on a channel which was shared with a wireless network with little activity. All other available networks were on orthogonal channels. A two-hop wireless scenario was configured in adhoc mode, where a host sent traffic to the wireless router which forwarded it to a destination host. Routing tables were hard-coded.

For all measurements we generated test traffic for 60 seconds. We measured the average CPU usage on the router using `top` [22]. Every ten seconds process statistics were written to a file. The values were averaged, with the first value omitted so as to exclude the saturation phase at the beginning of an experiment. All measurements were repeated on five occasions on different days at different times of the day. Maximum throughput measurements were subsequently repeated three times per occasion. Unless stated otherwise, a low standard deviation was seen across the results.

2.2 Packet Forwarding

To understand the base-line overhead of forwarding packets (the fundamental role of a mesh device), we generated a single TCP stream with `thrulay` [27] at the maximum possible sending rate. We use the maximum possible sending rate to demonstrate a worst-case two-hop scenario. Forwarding the packets caused an average CPU utilisation of 64.6 % on the mesh device. The average goodput was 13.3 Mbps. We obtained similar values using a Linksys WRT54G3G with a recent OpenWRT release instead of the Netgear WG302, or by measuring the CPU usage by analysing the remaining CPU time with a process with a low priority. If the stream is directly transmitted between the hosts, without being forwarded by the router, the average goodput was 25.8 Mbps. The high CPU usage to forward the TCP stream is caused by hardware and predominantly software interrupt processing. Hardware interrupts are caused by retrieving packets from the network card. The packets then traverse the IP stack, which includes the Netfilter system [17]. Such a high base-line resource usage under high traffic load implies that an IDS needs to be efficient, otherwise it may not get enough runtime to process all the packets.

2.3 Connection Tracking

Apart from forwarding packets, another essential task of the router is to track the state of network connections. This is essential for Network Address Translation (NAT) and Stateful Packet Inspection (SPI). Connection tracking in the Linux kernel is performed by the Netfilter system. In OpenWRT, connection tracking is active by default, and some WMNs [9, 18] perform NAT on every mesh device.

With three measurements we evaluate the per-packet costs associated with tracking new and existing connections. First, we sent SYN packets with random source ports at different rates. This results in most packets being identified as belonging to a new connection. We also sent UDP packets with an IP length of 40 B, equivalent to the combined size of the IP and TCP headers. All UDP packets use the same ports and therefore belong to the same connection. To compare against a base-line without connection tracking, we sent SYN packets and disabled connection tracking with the Netfilter NOTRACK target. By default, the maximum number of observed connections is set to 4096 on the WG302. If the connection table is full, a connection for which no reply packets was observed is replaced, or the new one is not entered into the table. Figure 1 shows the CPU usage on the router. The packet rate corresponds to the number of forwarded packets on the router. All packets were dropped at the destination.

It can be seen that while forwarding the SYN packets, the CPU usage increases along with the packet rate until the CPU resources are exhausted. The high workload in kernel-space does not allow to forward more than 3705 pps, existing TCP connections completely stall and user-space processes get hardly any runtime. The router's operation is severely affected. These results suggest, that the router should be protected against the effect of new connections at high rates (e.g., with Netfilter limit or hashlimit).

Forwarding the UDP packets requires a hash table lookup to obtain the corresponding connection, but no new connection structures need to be initialised and entered into the connection table. This requires less resources and the maximum forwarding capacity is limited by the wireless channel. By comparing the UDP packets to the SYN packets with connection tracking disabled, we can see that the lookup alone is costly as well. In all experiments which analyse the

CPU usage for different packet rates, we observed a high deviation specifically at 800 pps. The cause of this deviation is a matter for further investigation.

2.4 Packet Capturing

To be able to determine the existence of attacks, an IDS has to capture network traffic, which involves making packets accessible in user-space. In this section, we investigate the overhead associated with different approaches for packet capturing, including PCAP, PCAP MMAP and ULOG[1]. We measure the CPU usage and the ratio of dropped packets, i.e., packets that are forwarded by the router, but not made available to the capturing application.

2.4.1 PCAP

A popular way to capture packets is with the high-level capturing library *libpcap* [13]. Packets can be filtered in kernel-space with the BSD Packet Filter (BPF) language [15]. Using libpcap to capture packets results in capturing every packet twice: once when it arrives at the router and once again when it is forwarded. To prevent this, we use capture filters on the link-layer to only capture incoming packets. For all experiments with libpcap, we used non-promiscuous mode, which captures only packets actually being sent through the mesh device performing the analysis.

To analyse the performance of libpcap, we developed a test tool that reads packets from the socket and silently discards them. We used `thrulay` to generate a TCP stream at the maximum possible throughput. We also generated UDP packets with an IP length of 40 B. Table 1 shows the CPU load on the router while capturing the packets. Capturing packets at their full length is slightly more expensive than only capturing up to the first 92 B of each IP packet. Recall the base-line CPU utilisation for forwarding the TCP stream was 64.6 %; capturing packets increases utilisation further and leaves even less resources for intrusion detection. No packet drops were observed for the TCP stream. Capturing small UDP packets at high packet rates uses up almost all CPU resources, which results in a packet drop rate of 44.6 %.

[1]We tried to use Netfilter NFLOG [17] and PF_RING [7] on our test platform, but were unable to because of portability issues.

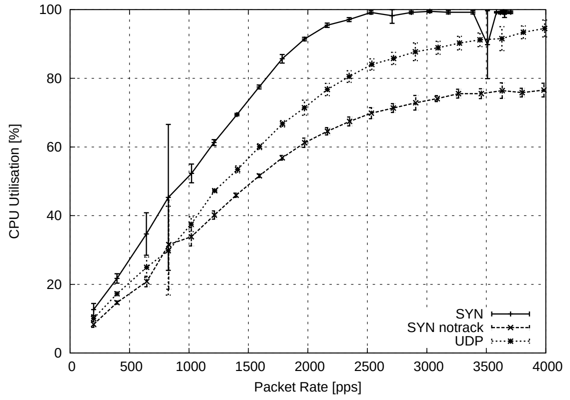


Figure 1: CPU utilisation to forward SYN packets with random source ports, with and without Netfilter connection tracking, and small UDP packets with fixed ports.

Method	TCP full	TCP header	UDP
PCAP	80.7 %	78.2 %	94.7 %
PCAP MMAP	80.3 %	83.5 %	98.4 %
ULOG	79.7 %	73.6 %	83.0 %

Table 1: CPU utilisation to capture a TCP stream and small UDP packets with PCAP, PCAP MMAP and ULOG.

2.4.2 PCAP MMAP

Libpcap MMAP [37] is similar to the standard libpcap version but uses a memory-mapped ring buffer instead of a packet socket to make packets accessible in user-space. It has been reported to reduce packet drops during high traffic loads [7].

Performance of capturing full packets of the TCP stream is similar to the standard version, as shown in Table 1. The use of the memory-mapped buffer should be faster, as the packets do not have to be copied to user-space. However, managing the ring buffer involves performing division operations, which are expensive. We can see this at the higher CPU load to capture only packet headers. This is because the slot length is smaller and on the router, division by a small number is more expensive than by a large number. Dividing 20,000 by 4 was 2.5 times slower than dividing by 40. Similar operations would be performed during packet capturing. No packet drops occurred while capturing the TCP stream. Capturing small UDP packets fully exhausts the CPU resources but leads to minimal packet loss (0.1 %). However, because the resources are exhausted by kernel operations (i.e., the divisions), the forwarding capacity is reduced. The UDP packet rate with the standard version of libpcap was 4363 pps, this was reduced to 3778 pps with the memory-mapped version. Almost all forwarded packets were captured, but 13.4 % less packets were forwarded. The low drop rate is therefore misleading, as the router is doing less actual work while leaving fewer resources available for other applications.

2.4.3 ULOG

The Netfilter system [17] allows the filtering and logging of packets. With the Netlink ULOG target, packets can be exported to user-space with the help of the Netlink communication system. ULOG supports multi-part messages, where a single `read()` operation can retrieve up to 50 bundled packets from the kernel. This reduces the number of necessary context switches, and should improve performance.

ULOG has the advantage that IP checksums are validated before packets are handed to the Netfilter system, and the connection tracking subsystem then performs IP defragmentation before letting packets traverse further [2]. Packets with an invalid transport layer checksum can be filtered, as they will have state INVALID. By performing these operation in the kernel, it is not necessary to implement them in the IDS in user-space, which should enable performance improvement without increasing the risk of detection evasion.

To analyse the performance of ULOG, we implemented a test program that simply read multi-part messages from the Netlink ULOG socket, and decoded them to count the number of packets. Again, the CPU load on the router is shown in Table 1. Performance is similar to the PCAP approaches while capturing full packets of the TCP stream. There is a performance improvement when capturing only packet headers and performance is significantly better to capture the small UDP packets. This is because the multipart messages are filled slower with smaller capture or packet lengths than with full packets, resulting in messages being flushed into user-space less regularly and therefore causing fewer context switches. Capturing both the TCP stream and the UDP packets did not result in any packet drops.

2.5 Traffic Pre-processing

Having captured network traffic, an intrusion detection system, which performs deep packet inspection, typically performs pre-processing activities to normalise packets before continuing with the analysis. Example forms of pre-processing include IP defragmentation, checksum validation, connection tracking and stream re-assembly. Such manipulations are important to prevent detection evasion. To determine the overhead of pre-processing, we conducted some experiments using the Snort [24] and Bro [20] intrusion detection systems. Both of them capture packets at their full length. The standard version of libpcap was used.

Snort is a signature-based intrusion detection system. To determine the overhead associated with traffic pre-processing with Snort we used version 2.4.4 and disabled all rules (signatures) – all that remains is packet capturing and pre-processing activities. On our test network, we generated a single TCP stream at maximum sending rate with `thrulay`. Starting Snort caused 4 MB RAM to be used. The test traffic caused 92.7 % CPU utilisation and 13.1 % of all packets were dropped. To determine the cost of pre-processing, we disabled the previously enabled pre-processors *flow*, *frag3*, *stream4* and *xlink2state*, as well as all rules. This resulted in a CPU usage of 86.5 % and the packet drop rate was reduced to 7.2 %. Performing pre-processing in the context of a system already close to its resource limit involves costs that impact the ability to perform detection.

Bro is a real-time network analysis platform that can be used to perform network intrusion detection by anomaly detection and pattern matching. Its core is the event engine which performs policy-neutral analysis of network traffic at different semantic levels. It generates high-level events (e.g., as a connection's state changes or for HTTP requests and replies), which can be analysed by user-specified policy scripts. To generate high-level events, Bro tracks connections and runs application-layer protocol analysers.

To analyse the performance of Bro, we used version 1.3.2 with no user-specified policy scripts enabled. We set capture filters to collect all incoming packets. Because no event handlers were specified, Bro skipped the work associated with creating events. We generated a TCP stream at maximum throughput, using `thrulay`. No application-layer protocol analysis was performed on the traffic, and dynamic protocol detection was deactivated.

Capturing and processing the generated TCP stream with Bro fully exhausted the CPU resources, and caused 37.6 % of all packets to be dropped. This demonstrates that packet capturing and pre-processing under high traffic loads is prohibitively expensive using Bro; running protocol analysers and processing events (intrusion detection tasks) would further increase this cost.

Unlike in our experiments, all packets are not normally captured using Bro. It comes with default policy scripts, called *Bro-Lite*, that, for performance reasons, only captures and analyses TCP control packets (i.e., SYNs, FINs, and RSTs), TCP fragments and some services (e.g., HTTP, FTP control channel, SMTP, and telnet). However, Paxson [21] suggests as an increasing number of application protocols need to be analysed for malicious activity, and on non-standard ports, then kernel-level packet filtering (as is done in Bro-Lite) will become less useful.

In Section 2.3, we demonstrated the costs for performing connection tracking using the Netfilter system. We performed a similar analysis with Bro[2] that performs its own connection tracking, with and without its *connection compressor* [8] enabled. The aim of the connection compressor is to improve system performance in situations where high rates of half-open connections are observed, for example, during SYN flood or scanning attacks. In such situations, only a minimal state structure is allocated for half-open connections. The instantiation of full connection state is deferred until a packet from both endpoints is seen. Memory requirements for the minimal state are substantially lower than for the full state. Consequently, large memory blocks holding several states can be allocated, instead of individual smaller blocks, and timeout-handling is simplified.

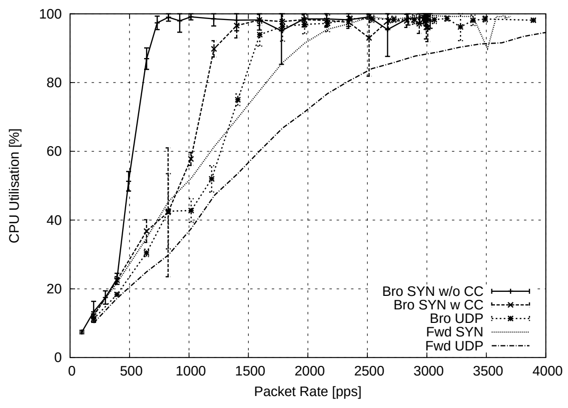
Figure 2(a) shows the CPU usage for Bro to analyse SYN packets with random source ports and UDP packets with a size of 40 B and fixed ports at different packet rates. Figure 2(b) shows the packet drop rate. As we have shown earlier, the connection tracking system of the Linux kernel gets overloaded by SYN packets with random source ports at high packet rates. Tracking connections also in user-space constitutes a second performance overhead which, together with the packet capturing overhead, causes the CPU resources to get exhausted at low packet rates. It can be seen that using the connection compressor improves performance. Both the CPU usage and the packet drop rate increase later with increasing packet rates. The more efficient memory management and the postponed allocation of the TCP analysers pay off. Here only SYN packets are processed. For a complete handshake, performance with the connection compressor would be worse than without, because both a minimal state and later a full state would have to be allocated. The connection compressor only improves performance for unreplied connection attempts and it is only applicable for TCP and not ICMP or UDP.

Packet drops under high throughput make Bro and Snort's stream re-assembly and therefore deep packet inspection unreliable. Small packets and especially new connections overload Bro already at packet rates far from the maximum channel capacity. Both systems get overloaded even without actually performing any detection, just by capturing and performing pre-processing.

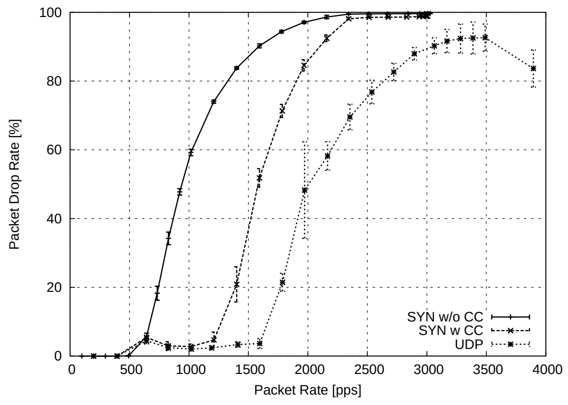
2.6 Signature Processing

A widely-used approach to detecting malicious network behaviour is to match observed network traffic against signatures of known attacks. To determine whether this is viable on lightweight mesh devices, we conducted some experiments using Snort. We used version 2.4.4 of Snort, and the latest official and freely available set of rules (signatures)

[2]We did not choose Snort because we will show that Snort's signature-based detection has general limitations.



(a) CPU Usage



(b) Packet Drops

Figure 2: Performance of Bro analysing SYN packets with random source ports, with and without connection compressing, and small UDP packets with fixed ports.

that date from 2005. For these experiments, we generated a single TCP stream, using `thrulay`.

To determine the cost of signature processing, Snort's default configuration was used, and a set of 958 rules out of 3573 was activated (bad-traffic, exploit, scan, dos, ddos, dns, tftp, icmp, netbios, smtp, virus). Starting Snort with this limited set of detection rules caused the system to crash due to memory exhaustion. We had to run Snort with a slower, but more memory efficient, pattern matching algorithm (configuration option `search-method lowmem`). With this setting, Snort used 11 MB of RAM.

Snort rules are organised in port groups that are defined by the protocol, source and destination port. To identify which rules will have to be evaluated for a packet, a fast multi-pattern search for the longest content string of each rule of a packet's port group is performed on the packet's payload. If this initial string matching algorithm finds a potentially matching rule, the rule's other mandatory fields (e.g., source and destination IP addresses) are checked and, upon success, the optional conditions of that rule validated. This can include an expensive pattern matching operation which uses all the keywords of a rule and also validates their position. This two-phase approach has the advantage that not all rules need to be fully evaluated. The port group evaluated for the `thrulay` traffic only contained 61 generic IP rules and rules referring to any TCP source and destination port. None of them will ever be fully evaluated because the source and target IP addresses do not match.

In this experiment, Snort exhausted all CPU resources, which causes 34.6 % of all packets to be dropped. Such a high drop rate makes it likely that an ongoing attack would not be detected, because the information identifying an attack is in the packets that cannot be analysed. Here, the number of rules in the port group is small and no rules matched the incoming packets and therefore no extensive pattern matching algorithms were run. Pattern matching is the most expensive task a NIDS must perform [3, 39]. It is expected that performance would further decrease with a larger number of rules in the port group (e.g., for port 80) and with the complete evaluation of signatures.

2.7 Performance Analysis Summary

In this section, we have analysed the ability of a typical mesh router to perform common intrusion detection tasks. As a base-line, we have shown that simply forwarding traffic at high data rates consumes significant quantities of CPU resources. A part of these resources are used for connection tracking, i.e., looking up the connection a packet belongs to, and these costs increase if packets belonging to new connections are being forwarded.

Packet capturing is one of the fundamental tasks an IDS must perform. With three different packet capturing approaches we have shown that capturing increases CPU utilisation further, but there are sufficient resources to capture full-length TCP packets at high data rates. Problems exist with small packets, where only Netfilter ULOG was able to capture small UDP packets at high rates without drops.

Existing intrusion detection systems Snort and Bro suffered from packet drops when pre-processing a TCP stream with high throughput, even without performing the actual intrusion detection. Both systems perform deep packet inspection which requires TCP stream re-assembly, connection tracking, checksum validation and IP defragmentation, to avoid detection evasion. A high rate of packet drops make such systems unreliable. We therefore propose to perform intrusion detection which does not perform deep packet inspection. This results in fewer requirements for pre-processing. Such a system could be more resilient to packet drops or even allow packet sampling.

We have shown that Snort's signature-based approach is unreliable at high packet rates, caused by a high amount of packet drops. Only a limited set of signatures was active in our tests, the port group of the test traffic did not contain many rules and no rules were actually evaluated. More general traffic will have worse performance, as port groups will contain more rules and some of them will get evaluated in detail. The space of attacks which could originate from clients in a WMN is diverse, and to achieve a good coverage of attacks a high number of active rules are required. Further, signature databases grow continuously. This will increase the memory requirements. For these reasons, we consider the rule-based approach as not suitable, even at lower traffic rates.

3. LIGHTWEIGHT IDS

In Section 2, we show how performing intrusion detection on the platforms that are commonly used in WMNs is challenging. In particular, deep packet inspection approaches are not usable reliably, and signature-based intrusion detection has additional limitations in terms of memory requirements. In this section, we describe an anomaly-based IDS, named *OpenLIDS*, that can be deployed on the hardware typically used in WMN settings and that is efficient enough to keep-up with higher traffic rates. We implement mechanisms that detect generic classes of attack traffic, which were exhibited in a number of recent malware outbreaks. The set of detection metrics is extendable, as long as they are efficient enough.

We aim to detect port scanning (e.g., malware propagation behaviour) and resource consumption attacks (e.g., TCP SYN flooding attacks), as they have been shown to have a significant impact on a wireless network's performance, and continue to be observed in current malware [5, 31]. Stone-Gross et al. [32] analysed the effect of such malware on a single-hop wireless network with a wired backhaul. In the scenario they analysed, the traffic generated by a malware-infected computer significantly degraded the network's performance.

Additionally, we use simple mechanisms to detect mass-mailing worms. Wong et al. [36] observed that during separate outbreaks of the mass-mailing worms *MyDoom* and *SoBig*, the volume of TCP traffic increased by 50 %.

3.1 Detection Mechanisms

We use a number of simple host-based anomaly detection mechanisms to identify malicious behaviour. To determine the existence of scanning behaviour and resource consumption attacks, we use an algorithm based on that developed by Wagner et al. [35]. We employ a simple mechanism that looks at DNS MX queries and SMTP traffic to detect mass-mailing worms and spam email activity, the latter of which often originates from compromised computers. IP sender addresses can be spoofed to obscure the origin of traffic; we use a simple mechanism to detect this. To have enough detailed information to choose appropriate remedies in case of an alert, we collect data about the network activity of each host.

To be able to detect an attack, the detection thresholds have to be sufficiently low. On the other hand, the thresholds have to be high enough not to cause false alerts for benign traffic. The thresholds presented here are either based on other work (as for the failed connection attempt analysis) or on what we define as benign behaviour. They will have to be adapted to the environment they will be deployed in. We describe the detection mechanisms in more detail.

3.1.1 *Failed Connection Attempt Analysis*

Scanning behaviour and certain resource consumption attacks cause abnormal numbers of failed connection attempts. We elaborate on what we use to determine a failed connection attempt for TCP and UDP protocols shortly. Wagner et al. [35] propose a multi-level approach to detecting such activity that initially performs low-cost traffic analysis on a per-host basis, and subsequently more detailed and expensive analysis, if suspicious activity is detected. Figure 3 illustrates the multi-level detection approach. It shows the states a host that is being monitored can be in for each protocol (rectangles), the tests performed in order to determine state-changes and alert generation (diamonds), and the alerts that can be generated (rect-triangles). The thresholds

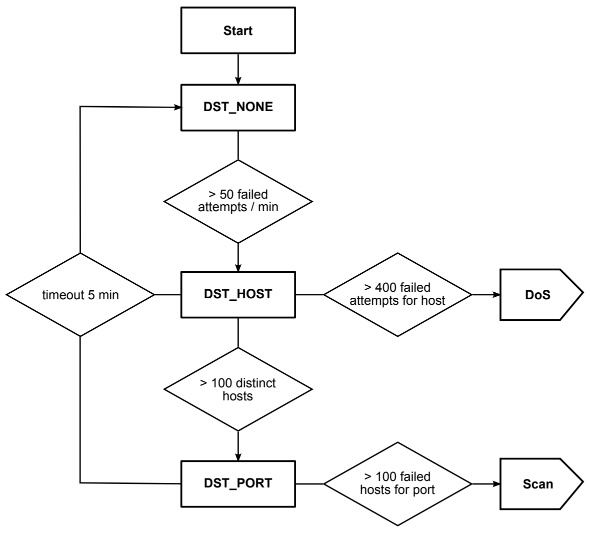

Start
DST_NONE
> 50 failed attempts / min
timeout 5 min
DST_HOST
> 400 failed attempts for host
DoS
> 100 distinct hosts
DST_PORT
> 100 failed hosts for port
Scan

Figure 3: Failed connection attempt detection algorithm

are based on the original algorithm, which was shown to be successful at detecting network worms without causing false positives.

Initially, each monitored host is in state DST_NONE. If more than 50 failed connection attempts are observed for a host within a minute, it is flagged as behaving suspiciously and moves into the DST_HOST state. Subsequent failed connection attempts of that host are counted for each *destination* IP address individually. When more than 400 failed connection attempts are counted for a single destination, the host is confirmed to be carrying out a DoS attack, and an alert is generated. If connection attempts fail to more than 100 *distinct* destination addresses, the host is suspected of scanning and moves into the DST_PORT state. If in this state, connection attempts to a *single destination port* fail for more than 100 distinct destination addresses, the host is considered to be scanning, and an alert is generated. After five minutes in the same state, a host will return to state DST_NONE.

For TCP, a failed connection attempt is generated when, in response to an initial SYN packet, an RST packet is generated with its sequence number set to zero, an intermediate router or destination sends an ICMP unreachable message, or a connection times out without any reply. We use a flow-based definition of a connection for UDP. Most UDP-based application protocols are bidirectional. Therefore, a UDP flow without a packet in the reply direction is counted as a failed connection attempt. Flows of unidirectional UDP-based application protocols (e.g., RTP [25]) are counted as failed connection attempts. Because this happens once per flow, the effect is limited, i.e., we don't anticipate hosts to typically generate in excess of 50 such flows in a minute. The original algorithm presented by Wagner et al. [35] counts every UDP packet without a reply as failed connection attempt, and would trigger an alert for RTP traffic. Our flow-based definition of a UDP connection does not have this drawback, but is unable to distinguish an RTP stream from a UDP DoS flood with fixed source and destination ports. To avoid false positives, we are not able to detect such floods.

3.1.2 Spam Detection

Mass-mailing worms like SoBig or MyDoom send emails directly to a target's email server [36]. To send emails to a specific domain, one must first determine the mail exchange (MX) server for it, which can be achieved using DNS. Because DNS MX queries are not necessary to send emails via an ISP's mail server, observing them can be used to identify spam sending hosts [16]. Once the target's MX server address is known, one or more SMTP connections will typically be established. 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. To detect mass-mailing worms and spam sources, we generate an alert if during one minute a host sends more than three DNS MX queries and initiates more than three flows to TCP ports 25 (SMTP).

To send spam email it is not necessary to issue DNS MX queries; target servers could be hard-coded into the malware or downloaded. Furthermore, instead of sending emails directly to the destination server, they could be sent over a relay, such as the user's ISP's mail servers. To cover these two cases we also count the number of flows to TCP ports 25, 587 or 465 (SMTP, submission and SMTP over SSL). If a host initiates more than ten such flows per minute, it is identified as a spammer. Because emails with multiple recipients are normally sent within the same flow, a user is not expected to send emails at rates higher than this. Potential email servers within the WMN (an unlikely occurrence) can be white-listed.

3.1.3 ICMP Analysis

ICMP echo requests are used as a preparation for worm propagation and to saturate a target's network links. They are not essential for a network's operation and can be blocked without severe side effects. To keep the detection of malicious ICMP behaviour simple and resource efficient, we analyse the difference between echo requests and echo replies. If within one minute more than 130 echo requests are not replied to, an alert is raised. This threshold is high enough to constantly ping two non-replying destinations with one ICMP echo request per second each.

In addition to analysing the difference between echo requests and replies, we raise an alert if a host sends more than 62 echo requests that are larger than 256 B, more than 3000 ICMP packets or 300 KiB of ICMP traffic per minute. Large echo requests can be used e.g., to analyse *Maximum Transmission Unit (MTU)* problems, but are not usual. However, they can be used to starve a target's network resources. Analysing general ICMP rates allows us to detect DoS floods that receive replies and attacks that use other ICMP packet types.

3.1.4 IP Spoofing Detection

IP source address spoofing is required in certain reflection attacks [29], and can be used to obscure the source of an attack. We recommend to filter unreasonable source addresses, i.e., those not from a network's own address range. Furthermore, it is possible to specify network interfaces or subnetworks on which clients are directly connected to a mesh device, i.e., without intermediate routers. By doing this, it allows the observation of a packet's source MAC address. If these measures are in place, we can detect IP spoofing by raising an alert when within five minutes more than three IP addresses are used by a single MAC address. This assumes that the MAC address itself is not spoofed.

3.1.5 Annotation

To be able to estimate the severity and plausibility of an alert and to choose the best suited remedies, we collect general data about each host's network activity and use this information to annotate detection alerts. We collect the following metrics:

- Number of packets and bytes sent
- Ratio of incoming and outgoing traffic
- Average packet size and standard deviation
- Average flow size and standard deviation

Annotating alerts makes responding to an event easier and may justify harder responses. An alert about a scanning host is, for example, more plausible if the average packet and flow size is low and has a small deviation and it is more severe (has a higher impact on the network quality) if it has a high packet rate.

3.2 Implementation Details

OpenLIDS was designed to operate on wireless mesh routers that run Linux[3]. It is written in C and uses ULOG to capture packets. Generally, up to the first 92 B of a packet are captured. This allows us to analyse the TCP source and destination ports of an inner packet of an ICMP error message, if all IP options are used. DNS queries are captured at full-length.

Because data structures are allocated for each observed host, it is important to specify the networks that should be observed. It is likely that no connection attempts from the Internet are allowed. Observation of WMN clients is therefore sufficient.

Captured packets and connection events (see Section 3.3) cause a hash table lookup to obtain a host's data structure to increase simple counters. Every 60 seconds these counters are reset. If a host is in a state other than DST_NONE, failed connection attempts are counted in a hash table. This has slightly higher resource requirements, but is necessary to analyse failed connection attempts to single destinations or ports.

Checking if detection thresholds are reached is performed periodically every two seconds, rather than on every packet. We expect this to have better performance. Finally, if an alert is generated the counters causing this alert are reset and we stop capturing that traffic which caused the alert. Optionally, the traffic which was identified as malicious can be automatically blocked. Both is achieved by modifying the Netfilter rules and these modifications stay active for a specifiable amount of time.

3.3 Connection Tracking

We have seen in Section 2 that tracking connections in kernel-space introduces a performance overhead. Stream re-assembly in the IDS requires connection tracking to be performed in user-space as well. Bro gets overloaded with relatively low rates of new connections. As in OpenLIDS

[3]OpenLIDS is available for download at `http://www.comp.lancs.ac.uk/openlids/`.

we do not re-assemble TCP streams, but need connection tracking to analyse connection time-outs, we want to take advantage of the connection tracking system in the kernel. This already performs the functions we need, and we expect a performance gain by avoiding 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)*. We analyse the events on seeing a new connection and on removing one from the connection table. We use the new events for generic analysis and the spam detection, and the destroyed connection events for the failed connection attempt analysis and to obtain the flow size.

A failed connection attempt is described by a destroyed connection event, which indicates that no packets have been received in the reply direction. The destroyed connection event could be generated by a timeout or the attempt being rejected with an ICMP unreachable message. To distinguish between these two cases, we introduce a new status flag which is set when related packets are associated with a connection (e.g., ICMP error messages). To use this flag a kernel patch is required that allows destination unreachable messages to be counted without awaiting a potential timeout. In the absence of this patch, either the connection timeout and the ICMP message need to be counted as two events, even if they are caused by a single failed connection attempt, or the ICMP unreachable message is ignored and 30 s waited for the connection timeout. The latter approach cannot be used to detect a UDP flow that aims to starve a destination's network capacity and is rejected with multiple ICMP unreachable messages. Applying the patch in a WMN should not be a problem, as custom builds of OpenWRT are used anyway and patches are easy to integrate.

A problem with using the kernel connection tracking system is that the timeouts are set not to break open connections. This is important for dynamic NAT and stateful packet inspection. If a connection is removed from the connection table because of a timeout, packets from that connection which arrive later might be dropped. The default timeout for TCP connections in state SYN SENT is 120 s, which would cause long delay in detection. We believe it 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 will typically arrive within 30 seconds. The timeout for UDP connections without a packet in the reply direction is set to 60 s. Reducing the UDP timeout, e.g., to 30 s could be considered as well, but depending on the application this modification is more dangerous, so we do not recommend that. The timeout for ICMP requests is 30 s, which is acceptably low.

The cessation of connection tracking events after detecting an attack can be achieved if the malicious traffic is automatically blocked or marked as not to be tracked in the Netfilter raw table. BPF support for Netlink is under development and some features are currently still missing (e.g., blocking events based on port numbers). As soon as these are implemented, connection tracking events can be filtered in kernel space even without active mitigation or exclusion of the malicious traffic from SPI.

4. EVALUATION OF OPENLIDS

The performance of OpenLIDS is evaluated using the same setup as that described in Section 2. To determine whether OpenLIDS is able to detect malware, we test the implemented scanning detection algorithm with a recent Internet worm, called Conficker. We also test it for false positives with real network traces from a deployed community WMN[4], and with two popular peer-to-peer file-sharing applications.

4.1 Wray Community WMN Traces

We have access to a WMN that provides Internet connectivity to members of a rural village, called Wray [9]. The network consists of twelve mesh boxes that serve up to a total of 50 users at one time. Internet connectivity is provided at a single point in the village. The mesh devices in this network use different logical network interfaces for intra-mesh communication and client access.

To have a set of real-world network traces for our experiments, traffic in the Wray network was captured for 18 days in Spring, 2008. On every mesh box the first 54 B of every IP packet entering or leaving a client network interface was captured. To represent traffic at a single point, we merged all the network traces that we collected. We then split the merged traces into hour-long segments and all broadcast messages were removed, leaving only routable traffic. During the busiest hour, a maximum number of 43 hosts were seen and the average data rate observed was 1.32 Mbps.

In experiments using the Wray traffic, the traces were replayed with `tcpreplay` [33]. The packets were padded to their original length and sent to the router. The router had a default route set to a receiver, which simply dropped everything. The packet flow contained both ingress and egress traffic.

4.2 OpenLIDS Performance Analysis

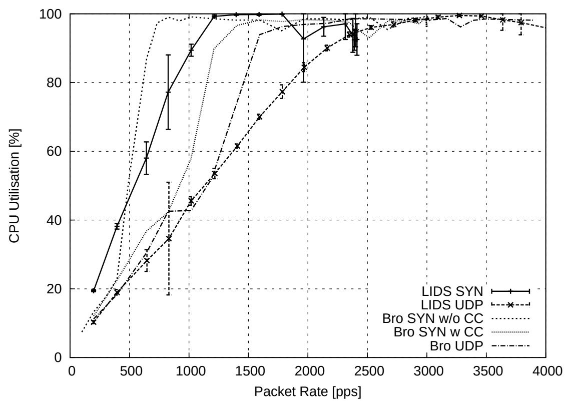
To determine the potential performance benefits associated with the implementation approach of OpenLIDS, we conducted similar experiments to those we were performing in Section 2. We determine how well OpenLIDS performs to analyse a TCP stream and the Wray traffic sent at high data rates. We further test the performance to handle small packets, and new connections at different rates. Finally, we discuss some of the overheads of using OpenLIDS in terms of memory usage and impact on packet forwarding.

4.2.1 *High Throughput Analysis*

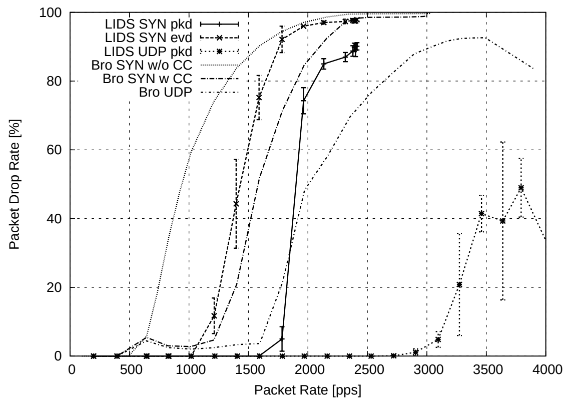
For OpenLIDS, analysing a TCP stream generated with `thrulay` at the maximum possible sending rate caused a CPU utilisation of 78.8,% and no packet drops occurred. For the same experiment, CPU resources were fully exhausted with Snort and Bro, and they both suffered from packet drops. In Section 2.4, we described how the performance overhead to capture packets with ULOG (which OpenLIDS employs) was comparable to PCAP (which Bro and Snort uses). Not tracking connections in user-space and not performing stream re-assembly leads to much better performance. In contrast to Snort and Bro, which suffered from packet drops while only doing pre-processing, OpenLIDS is performing detection in this experiment.

Replaying the Wray trace from the busiest hour at the maximum possible sending rate, rather than its real speed,

[4]The Wray and Conficker traces are available for download at `http://www.comp.lancs.ac.uk/openlids/`.



(a) CPU Usage



(b) Packet and Connection Tracking Event Drops

Figure 4: Performance of OpenLIDS analysing SYN packets with random source ports and small UDP packets with fixed ports, compared to the performance of Bro.

caused a CPU usage of 78.6 %, again without any packet drops. Only hosts in the WMN were observed (and no hosts on the Internet), which typically only causes one hash table lookup per packet. Because the Wray traffic includes more source addresses and flows than the `thrulay` traffic, more host structures were allocated and more connection tracking events generated. Replaying the traffic faster than its real speed caused some state changes in the failed connection algorithm.

4.2.2 Connection Tracking

Similar to the analysis performed in Section 2.3, the performance of OpenLIDS was tested with SYN packets with random source ports and 40 B long UDP packets with fixed ports. The CPU usage on the router is shown in Figure 4(a) and drop rates are shown in Figure 4(b). Because OpenLIDS receives packets and connection tracking events independently, we show the packet drop rate as well as the rate of dropped new connection events. The graphs also contain the previous results for Bro for comparison.

The UDP packet rate at which the CPU resources get exhausted is considerably higher than with Bro. Packet drops start to occur after 3000 pps. This is much higher than with Bro which already suffers from packet drops at 1800 pps. This improved performance is achieved by not tracking connections in user-space and by using ULOG for packet capturing, which performs better for smaller packet sizes than PCAP.

Each time a SYN packet arrives at OpenLIDS, a new connection event is generated, the packet is captured, and as soon as the connection table in the kernel starts overflowing, a destroyed connection event is generated. Because these three units have to be transmitted to user-space and be processed by OpenLIDS, the CPU usage is higher than for Bro when the connection compressor is in use, but lower than without it. For new connections, the use of the connection tracking system of the kernel does not improve performance. The event drop rate is higher than the packet drop rate, which is likely to be caused by the use of multi-part messages for packet capturing. It can be observed that the maximum achievable SYN packet rate is 2412, whereas it was 3705 without OpenLIDS and the `nf_conntrack_netlink` kernel module. In conclusion, generating the connection tracking events in kernel-space is not as cheap as we expected and limits the maximum forwarding capacity.

4.2.3 Overhead of using OpenLIDS

At a mere 85 KiB, the binary file for OpenLIDS is small, and uses 836 KiB of RAM when it is started. To observe a single host, data structures of 320 B are allocated (272 B if packet and flow deviation calculation is disabled). Observing 10,000 hosts used 4,128 KiB of RAM. This makes OpenLIDS very suitable to be run on devices with low memory resources, especially as typically only hosts in the wireless mesh network are observed. The memory requirements are much lower than with the signature-based approach of Snort.

To determine whether OpenLIDS added any additional delay to traffic or degraded throughput, we compared the data obtained from measurements with `thrulay`. We did not observe a significant difference in RTT or maximum throughput with OpenLIDS in place. As described in Section 4.2.2, the maximum forwarding capacity is affected in the presence of a high rate of new connections. Because we recommend to limit the rate of new connections in general (see Section 2.3), we consider this drawback as acceptable.

4.3 Detecting Malware with OpenLIDS

It is not enough to build a resource efficient intrusion detection system; it must also be effective at detecting attacks. The algorithms we propose for detecting malicious behaviour are largely based upon existing approaches. Results from other work [6, 38, 23] suggest, that we will be able to detect hosts generating spam. The authors of the failed connection attempt analysis algorithm successfully tested it with a variety of Internet worms [35]. To demonstrate that our implementation of this algorithm is effective, we conducted some experiments using collected traces from the recently discovered Conficker worm. A key feature of an IDS is that it does not generate false positives – we used the traces collected from Wray and two peer-to-peer applications to determine whether our implementation generates false positives.

4.3.1 Detecting the Conficker Worm

We analysed the detection capabilities of OpenLIDS with the Conficker (a.k.a Downadup) worm. Conficker is a recent network propagating worm, which has spread widely.

It exploits a vulnerability in the Windows RPC service, performs password attacks against network shares, and infects removable media. To exploit the RPC vulnerability, it scans random IP addresses on TCP port 445, which makes it detectable by OpenLIDS. The variant we used is detected by Symantec Norton AntiVirus as Downadup.B and was first discovered on 30 December, 2008 [5].

To obtain a trace file containing the malicious traffic generated by Conficker, we infected a virtual machine with the worm. The virtual machine was connected to an isolated wired network, and all traffic except outgoing HTTP and DNS connections was blocked. As a suspected way of evading detection, Conficker adapts its scanning rate based upon an estimate of available bandwidth. To simulate networks with different bandwidth capabilities, we throttled the available bandwidth on the network uplink router to 96 Kbps, 496 Kbps or left it unlimited at 100 Mbps. For each bandwidth setting, the full traffic was recorded.

To test the detection capabilities of OpenLIDS, we replayed each trace file five times in the same wireless test environment we used for our earlier experiments. To average the effect of the periodic counter resets that occur every minute on OpenLIDS, a random delay between 0 and 59 seconds was introduced between starting it on the wireless router and starting to replay the trace. The scanning rate and the average time to detection, after the first scan-packet was sent, are shown in Table 2. The scanning rate shown includes a retransmission for every SYN packet.

Bandwidth	Scan Rate	Detection Time
96 Kbps	6.5 pps	121.0 s
496 Kbps	17.4 pps	74.0 s
unlimited	67.0 pps	46.5 s

Table 2: Time to detect the Conficker worm with OpenLIDS with different bandwidth availability.

The results in Table 2 show that at even very low scanning rates, OpenLIDS is able to successfully detect the Conficker activity. Clearly, as the transmission rate increases, the detection times reduce. Thirty seconds of the delay in detecting the worm relates to waiting for connection timeouts. Lower timeout values would increase detection speed.

4.3.2 False Positives

To give an indication of the false positive rate of the detection mechanisms that are deployed on OpenLIDS, we replayed the fourteen traces from the Wray village WMN that have the highest data rates at their reals speed. Because the captured data only includes the first 54 B of each IP packet, we verified that no information required for detection was truncated (e.g., TCP ports of the original packet in ICMP unreachable messages). As the captured length is not sufficient to analyse DNS queries, we captured full DNS queries in Wray for twenty days in early Summer 2008. No MX queries were seen in these traces.

Replaying the Wray traces did not cause any alerts to be triggered, but caused some state transitions in the failed connection detection algorithm described in Section 3.1.1. The cause of these transitions were peer-to-peer applications. To improve confidence in OpenLIDS' ability not to generate false positives in light of peer-to-peer traffic, we ran experiments with *transmission* [34], a BitTorrent client, and *aMule* [1], an eDonkey and Kademlia client. In both cases more than 80 MB of a single file was downloaded and more than 270 sources were available. The experiment was carried out on a 1 Mbps DSL line with NAT being applied. Transmission did not cause any state changes. aMule caused a transition to state DST_HOST with subsequent transition to DST_PORT for UDP, and no alerts were generated. If a particularly aggressive peer-to-peer application used a standard set of ports, a false alert could be triggered.

4.4 Discussion

We have evaluated OpenLIDS and shown that it is able to reliably analyse TCP streams and network traffic from a real WMN at high throughput without any packet drops. Conversely, Snort and Bro fully exhausted the CPU resources and suffered from significant packet drops. Not performing deep packet inspection avoids having to do stream reassembly and therefore to use the connection tracking system of the Linux kernel which yields much better performance.

For new connections, this approach is not as efficient as expected, because generating and receiving connection tracking events is costly. Performance for new connections is comparable to Bro. Packets creating new connections are typically only a fraction of the traffic and efficiency is high under normal conditions. Using the kernel connection tracking system has the disadvantage that time-out values cannot be adjusted arbitrarily.

Our implementation is able to detect a wide range of attacks with generic and cheap detection metrics. We have shown it to be successful in detecting a real worm which scans at relatively low rates. We have further shown that no false positives were generated when analysing the traffic from a real WMN of the 14 busiest hours in an 18 days period.

5. RELATED WORK

We demonstrated that typical wireless mesh devices have resource issues when performing intrusion detection at high traffic rates. Similar resource problems exist with intrusion detection systems that aim to analyse traffic on high-speed networks. To make intrusion detection scale in this context, a number of approaches distribute traffic among multiple detection sensors that operate in parallel [11, 3]. Each sensor analyses only a subset of the packets on the network. Scalability is achieved by adding more sensors as data rates increase. Adding detection sensors in a WMN context is not a desirable option – typically, hardware costs need to be kept to a minimum.

Another approach to improve performance is by optimising the rule lookup for signature-based intrusion detection. WIND [28] implements an adaptive algorithm that systematically profiles attack signatures and network traffic to reorganise the rules more efficiently. Their approach uses more protocol fields to reject rules as soon as possible, and improves the performance of Snort by factor 1.35 to 1.65. Yoshioka et al. [39] use hash tables to find matching rules. Their approach reduces the memory requirements significantly and improves performance. These approaches only improve the rule lookup but not the packet pre-processing, which we have shown to have performance problems on its own. As signature evaluation would further decrease performance, and memory requirements for pattern matching are substantial, we consider signature-based approaches as not suited for wireless mesh networks.

Load-based adaptation of IDS behaviour has been proposed as a way of addressing resource constraints. Lee et al. [12] argue that an IDS should provide the best value under operational constraints. This means that rather than missing a critical attack while being overloaded, detection of less important attacks should be suspended to make the required resources available. By assigning benefits and costs to detection metrics, they calculate the optimal set that should be active under the current resource constraints. Similarly, to avoid CPU exhaustion and resulting packet drops, Dreger et al. [8] dynamically control load on a system by modifying packet capture filters. They statically define an ordered set of capturing filter as *load-levels*. The filter of a higher load-level imposes a greater load on the IDS than its predecessor, by including more traffic for analysis. The IDS monitors the current CPU load and switches to a lower load-level if it reaches an upper threshold, or to a higher load-level if it reaches a lower threshold.

Reducing packet capturing in the presence of an attack imposes the risk of missing exactly that attack. A minimal solution is to capture only TCP control packets. This is sufficient for the failed connection attempt analysis with TCP, but leads to a loss of useful information (e.g. average packet size and deviation) and is not applicable to UDP and ICMP. To reduce the load of the IDS, it is therefore preferable to disable protocol analysers, reduce the amount of preprocessing activity (e.g. disable stream re-assembly) and possibly switch to capture only packet headers. In this case it is possible to perform host-based cheap anomaly detection, as we presented here. The functionality of OpenLIDS could therefore build the lowest load-level in such a dynamic resource-aware intrusion detection architecture. This would cover resource efficient detection in high load scenarios where more complex analysis would become unreliable. With more computational resources available, the detection capabilities of OpenLIDS can be complemented with additional, possibly deep-packet inspecting, metrics.

There is some previous research on intrusion detection for wireless mesh networks. Much of this work addresses either architectural problems [40, 10, 30] or attacks on the WMN itself [4], and does not consider issues relating to the limited resource availability of mesh devices. We believe our work to be complimentary to this.

6. CONCLUSIONS

In this paper, we have presented an analysis of a typical wireless mesh networking device performing common intrusion detection tasks. It was shown that these resource constrained devices are unable to reliably perform traffic preprocessing, which is necessary to perform deep packet inspection. Furthermore, because of memory and CPU resource constraints, they cannot effectively be used for signature-based detection. To address these problems, we implemented OpenLIDS, a lightweight anomaly-based intrusion detection system. It has simple host-based metrics to detect scanning behaviour, resource consumption attacks, spam email distribution and IP address spoofing – common malware behaviour. Furthermore, it provides additional information about the characteristics of a detected attack, which can help to choose appropriate remedies. The implementation of the detection metrics is lightweight enough to reliably detect attacks at high data rates. Via experimental results, OpenLIDS was shown to be able to detect a current malware variant, and produced no false positives with traffic traces from a WMN deployment.

As further work, we intend to deploy OpenLIDS in an operational wireless mesh network to further validate our experimental results. Using the kernel connection tracking system has performance advantages under normal conditions, but is not as efficient for high rates of new connections. Also, it is not possible to arbitrarily adjust timeout values. Improving the performance of generating connection tracking events and increasing the flexibility of timeout specification are issues for further work.

7. REFERENCES

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

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

[3] I. Charitakis, K. Anagnostakis, and E. Markatos. An Active Traffic Splitter Architecture for Intrusion Detection. In *MASCOTS 2003. 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems*, pages 238–241, 2003.

[4] T. Chen, G. Kuo, Z. Li, and G. Zhu. Intrusion Detection in Wireless Mesh Networks. *Security in Wireless Mesh Networks*, page 145, 2008.

[5] Conficker Worm. `http://www.symantec.com/security_response/writeup.jsp?docid=2008-123015-3826-99`, December 2008.

[6] A. Decker, D. Sancho, L. Kharouni, M. Goncharov, and R. McArdle. Pushdo / Cutwail Botnet. `http://us.trendmicro.com/imperia/md/content/us/pdf/threats/securitylibrary/study_of_pushdo.pdf`, July 2009.

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

[8] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational Experiences with High-Volume Network Intrusion Detection. *Proceedings of the 11th ACM conference on Computer and communications security*, pages 2–11, 2004.

[9] J. Ishmael, S. Bury, D. Pezaros, and N. Race. Deploying Rural Community Wireless Mesh Networks. *IEEE Internet Computing*, 12(4):22–29, 2008.

[10] O. Kachirski and R. Guha. Effective Intrusion Detection Using Multiple Sensors in Wireless Ad Hoc Networks. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, 2003.

[11] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 285–293, 2002.

[12] W. Lee, J. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang. Performance Adaptation in Real-Time Intrusion Detection Systems. *Lecture Notes in Computer Science*, pages 252–273, 2002.

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

[14] Madwifi. `http://www.madwifi.org`, May 2008.

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

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

[17] Netfilter Project. `http://www.netfilter.org`, May 2008.

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

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

[20] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23):2435–2463, 1999.

[21] V. Paxson. Bro Intrusion Detection System Hands-On Workshop: Bro Overwiew. Technical report, Lawrence Berkeley National Laboratory, 2007.

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

[23] A. Ramachandran and N. Feamster. Understanding the Network-Level Behavior of Spammers. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 36, pages 291–302. ACM New York, NY, USA, 2006.

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

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

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

[27] S. Shalunov. Thrulay – Network capacity tester. `http://shlang.com/thrulay/`, August 2008.

[28] S. Sinha, F. Jahanian, and J. Patel. Wind: Workload-aware intrusion detection. In *Symposium on Recent Advances in Intrusion Detection (RAID'06)*, Hamburg, Germany, September 2006.

[29] Smurf Attack. `http://www.cert.org/advisories/CA-1998-01.html`, February 2007.

[30] D. Sterne, P. Balasubramanyam, D. Carman, B. Wilson, R. Talpade, C. Ko, R. Balupari, C. Tseng, T. Bowen, M. Res, et al. A General Cooperative Intrusion Detection Architecture for MANETs. In *Third IEEE International Workshop on Information Assurance*, pages 57–70, 2005.

[31] J. Stewart. Storm Worm DDoS Attack. `http://www.secureworks.com/research/threats/storm-worm`, February 2007.

[32] B. Stone-Gross, C. Wilson, K. Almeroth, E. Belding, H. Zheng, and K. Papagiannaki. Malware in IEEE 802.11 Wireless Networks. In *Proceedings of the Passive and Active Measurement Conference (PAM)*, 2008.

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

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

[35] A. Wagner, T. Dübendorfer, R. Hiestand, C. Göldi, and B. Plattner. A Fast Worm Scan Detection Tool for VPN Congestion Avoidance. *Lecture Notes in Computer Science*, 4064:181, 2006.

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

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

[38] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming Botnets: Signatures and Characteristics. 2008.

[39] A. Yoshioka, S. Shaikot, and M. Kim. Rule Hashing for Efficient Packet Classification in Network Intrusion Detection. In *Proceedings of 17th International Conference on Computer Communications and Networks. ICCCN 2008*, pages 1–6, August 2008.

[40] Y. Zhang and W. Lee. Intrusion Detection in Wireless Ad-Hoc Networks. *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCom'2000)*, 2000.