A Test-Bed for Misbehavior Detection in Mobile Ad-hoc Networks — How Much Can Watchdogs Really Do?

EPFL IC Technical Report IC/2003/72

Sonja Buchegger, Cédric Tissières, and Jean-Yves Le Boudec
EPFL-IC-LCA
CH-1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

Abstract: Several misbehavior detection and reputation systems have been proposed for mobile ad-hoc networks, relying on direct network observation mechanisms, so-called watchdogs. While these approaches have so far only been evaluated in simulations and restricted to selfish packet dropping, we are interested in the capabilities of a watchdog detection component in a real network. In this paper we present our test-bed implementation of misbehavior detection. Following an evaluation of both the feasibility and detectability of attacks on routing and forwarding in the Dynamic Source Routing (DSR) protocol, we present the design of our test-bed. In order to add detection capabilities, we extend the concept of passive acknowledgment by mechanisms for partial dropping, packet modification, and fabrication detection. We combine DSR with Netfilter and APE to enable detection. We implement both attackers and detection and show their feasibility and limitations.

1 Introduction

We are interested in the attacks on routing and forwarding in mobile ad-hoc networks. Specifically, we want to determine whether and how attacks can be mounted and detected by observation in a real network environment.

Several reputation-based systems to deal with misbehavior in mobile ad-hoc networks have been proposed, all relying on some component to detect misbehavior in the neighborhood of a node. To the best of our knowledge, so far, the evaluation of detection has been restricted to simulations and only to the misbehavior type of not forwarding packets not destined to one self. Even for the detection of this simple attack, some concerns have been raised [10] whether it is unambiguously feasible to classify it as such.

Our approach is to build a test-bed that can be used to test attacks as well as whether they can be detected, and thus study the practicality and feasibility of several reputation-based misbehavior detection systems.

The main contributions of this paper can be summarized as follows.

- We provide a systematic list of attacks on DSR and evaluate the effort and gain for mounting them as well as whether and how they can be detected.
- We extend the notion of passive acknowledgment to enable the detection of attacks.
- We built and present here a test-bed that enables researchers to assess the feasibility and detectability of attacks.
- We implemented and tested several attacks and showed their detection.
- We compared the performance of DSR enhanced by our extended passive acknowledgment detection mechanism to regular DSR. We found that it performs at least as well as explicit acknowledgment, but mitigates the problem of duplicates due to lost acknowledgments.

The remainder of the paper is organized as follows. First, in Section 2 we give some brief background information on Dynamic Source Routing (DSR) and pas-

sive acknowledgment, since we investigate attacks on DSR and detect them by means of an enhanced passive acknowledgment. Then we discuss related work in Section 3. In Section 4 we discuss the feasibility and detectability of attacks on DSR. We describe the design and architecture of our test-bed in Section 5, followed by our detection mechanism by enhanced passive acknowledgment in Section 6 and a description of the attacks we implemented in Section 7. We discuss the use of our test-bed and further work in Section 8, and Section 9 concludes the paper.

2 Background

2.1 Dynamic Source Routing

Dynamic Source Routing is a protocol developed for routing in mobile ad-hoc networks and was proposed for MANET by Broch, Johnson and Maltz [6]. In a nutshell, it works as follows: Nodes send out a ROUTE REQUEST message, all nodes that receive this message forward it to their neighbors and put themselves into the source route unless they have received the same request before. If a receiving node is the destination, or has a route to the destination, it does not forward the request, but sends a REPLY message containing the full source route. It may send that reply along the source router in reverse order or issue a ROUTE REQUEST including the route to get back to the source, if the former is not possible due to asymmetric links. ROUTE REPLY messages can be triggered by ROUTE REQUEST messages or gratuitous. After receiving one or several routes, the source picks the best (by default the shortest), stores it, and sends messages along that path. In general, the better the route metrics (number of hops, delay, bandwidth or other criteria) and the sooner the REPLY arrived at the source (indication of a short path - the nodes are required to wait a time corresponding to the length of the route they can advertise before sending it in order to avoid a storm of replies), the higher preference is given to the route and the longer it will stay in the cache. In case of a link failure, the node that cannot forward the packet to the next node sends an error message toward the source. Routes that contain a failed link, can be 'salvaged' by taking an alternate partial route that does not contain the bad link.

2.2 Passive Acknowledgment (PACK)

During packet forwarding every node is responsible confirming that the packet was received by the next hop. There are three ways to get this acknowledgment, as specified in the DSR draft [6]:

- Link-layer acknowledgment: this is supplied by the MAC layer.
- Passive acknowledgment: this confirmation comes indirectly by overhearing the next node forward the packet.
- Network-layer acknowledgment: this is when nodes explicitly request a DSR acknowledgment from the next hop.

PACK can be used for Route Maintenance when originating or forwarding a packet along any hop other than the last hop. PACK cannot be used with the last hop since it will never retransmit a packet destined to itself. PACK needs two conditions to be applied: nodes have their network interfaces in promiscuous mode, and network links operate bidirectionally.

PACK works as follows. When a node receives a packet to be forwarded to a node other than last hop, the node sends the packet without requesting a network-layer acknowledgment (ACK). If it doesn't overhear the retransmission of the next node within a timeout, the node retransmits the packet again, without network-layer ACK request. After a certain number of trials, a network-layer ACK request must be used instead of PACK for all remaining attempts for that packet.

When a node receives a new packet, it considers it as a PACK if the following checks succeed:

- Source address, destination address, protocol identification and fragment offset fields in the IP header of the two packets must match.
- If either packet contains a DSR Source Route header, both packets must contain one, and the value in the Segments Left field (it indicates the number of hops remaining until the destination) in the DSR Source Route header of the new packet must be less than that in the first packet.

3 Related Work

3.1 Detection

In this section we describe several approaches that build on the detection of misbehavior in mobile ad-hoc networks and could thus benefit from our test-bed to evaluate the effectiveness of attacks and their detection.

Watchdog and path rater components to mitigate routing misbehavior have been proposed by Marti, Giuli, Lai and Baker [10]. They observed increased throughput in mobile ad-hoc networks by complementing DSR with a *watchdog* for detection of denied packet forwarding and a *path rater* for trust management and routing policy rating every path used, which enable nodes to avoid malicious nodes in their routes as a reaction.

Intrusion detection for wireless ad-hoc networks has been proposed by Zhang and Lee [19] to complement intrusion-prevention techniques. The authors argue that an architecture for intrusion detection should be distributed and cooperative, using statistical anomaly-detection approaches and integrating intrusion-detection information from several networking layers. They use a majority voting mechanism to classify behavior by consensus. Responses include reauthentication or isolation of compromised nodes.

CONFIDANT proposed by Buchegger and Le Boudec [5] detects misbehaving nodes by means of observation or reports about several types of attacks and thus allows nodes to route around misbehaved nodes and to isolate them from the network. Nodes have a *monitor* for observations, *reputation records* for first-hand and trusted second-hand observations, *trust records* to control trust given to received warnings, and a *path manager* for nodes to adapt their behavior according to reputation.

CORE, a collaborative reputation mechanism proposed by Michiardi and Molva [11], also has a *watchdog* component; however it is complemented by a reputation mechanism that differentiates between subjective reputation (observations), indirect reputation (positive reports by others), and functional reputation (task-specific behavior), which are weighted for a combined reputation value that is used to make decisions about cooperation or gradual isolation of a node.

A context-aware inference mechanism has been proposed by Paul and Westhoff [14], where accusations are related to the context of a unique route discovery process and a stipulated time period. A combination is used that consists of un-keyed hash verification of routing messages and the detection of misbehavior by comparing a cached routing packet to overheard packets.

OCEAN by Bansal and Baker [4] relies exclusively on first-hand observations for ratings. If the rating is below the faulty threshold, the node is added to the faulty list. This faulty list is appended to the route request by each node broadcasting it to be used as an avoid list. A route is rated good or bad depending on whether the next hop is on the faulty list. In addition to the rating, nodes keep track of the forwarding balance with their neighbors by maintaining a chip count for each node.

Cross-feature analysis is proposed by Huang, Fan, Lee, and Yu [7] to detect routing anomalies in mobile ad-hoc networks. They explore correlations between features and transform the anomaly detection problem into a set of classification sub-problems. The classifiers are then combined to provide an anomaly detector. A sensor facility is required on each node to provide statistics information.

3.2 Test-Beds and DSR implementations

We evaluated several existing test-bed environments and implementations of DSR in view of what they provide to enable the detection of misbehavior. The criteria were that it had to be a real network, support promiscuous mode, support DSR, support passive acknowledgment, preferably have logging and scripting facilities, and it had to work on current off-the-shelf hardware such as available network cards.

Specifically, we considered APE [2], MobiEmu [20], the Monarch DSR implementation [9], Click [8] and the pecolab DSR implementation [15], and the piconet DSR implementation [17].

In comparison to the alternatives, the APE testbed combined with the piconet implementation of DSR fulfilled the largest range of our requirements. We integrated them and added capabilities as described in Section 5.

4 Attacks on DSR

In the following we give examples of attacks on DSR and classify them as dropping, modification, fabrication, or timing attacks. We also state their potential detectability.

4.1 Dropping Attacks

Drop all packets not destined to itself or perform only partial dropping. Partial dropping can be restricted to specific types, such as only data packets, or route control packets that contain it, or packets destined to specific nodes. The attacker can also decide to drop only some of the packets listed above. The previous hop can detect dropping by use of passive acknowledgment.

Avoid sending a ROUTE ERROR when having detected an error, to prevent other nodes from looking for alternative routes. Thus, the source of the data packet will not know that this route is disrupted and will not initiate a Route Discovery to find another route. By using fake data packets sometimes, the initiator could confirm the validity of the route if it receives a reply to this fake packet from the destination which cannot interpret the data. To the previous hops using passive acknowledgment this looks like dropping packets and can thus be detected as misbehavior.

4.2 Modification Attacks

By sending forged routing packets, an attacker can create a so-called black hole, a node where all packets are discarded or all packets are lost. If the attacker itself is the black hole and then just drops the packets, this can be detected by the neighbors using passive acknowledgment. If the black hole is a virtual node or outside the network, it is hard to detect. The attacker could also cause the route at all nodes of an area to point into the black hole area when the destination is outside the network. This could be done by sending forged ROUTE REPLY messages for example. The attack of using an unreachable node as a black hole is not easily detectable since the last node on the route that could not reach the destination will send a ROUTE ERROR back. If the attacker drops the ROUTE ERROR, this can be detected. Otherwise, the source node will initiate another route discovery process and the attacker will go undetected.

Attempt to make routes that go through oneself appear longer by adding some virtual nodes to the route. Thus, a shorter route will be chosen avoiding this node. When the attacker receives a ROUTE REQUEST, it replies with a ROUTE REPLY as if the route were already in its route cache, but it adds some virtual nodes to make the route longer. It could also modify (add some virtual nodes) and forward the ROUTE REQUEST. As the ROUTE REPLY comes back, it removes the virtual nodes and forwards the packet. By use of enhanced passive acknowledgment to detect tampering, adding nodes can be detected. In the same way, an attacker can remove itself to be avoided, this can also be detected by passive acknowledgment.

Change the Last Hop External flag in the ROUTE REPLY to make this route less interesting for the initiator of the route discovery. This modification can be detected by enhanced passive acknowledgment.

Salvage routes that are not broken and redirect a data packet to consume bandwidth and energy, or to deviate traffic for malicious purposes. When the attacker receives a data packet, it changes the route of the packet and also sends a ROUTE ERROR to the source to indicate the change of route. Thus, the source will delete the original route of its cache and will use the new route next time. It can potentially be detected when the next hop overhears the ROUTE ERROR containing itself.

To create a routing loop, an attacker could send forged routing packets that cause packets traversing nodes in a cycle without reaching their destination, consuming bandwidth and power. This could be detected if nodes check for loops in the source route not only when forwarding a ROUTE REQUEST. If, however, the attacker manages to use two different addresses for one node, it is not detectable from inspecting the header.

Modify the nodes list in the header of a ROUTE REQUEST or a ROUTE REPLY to misroute packets and to add incorrect routes in the route cache of other nodes. The attacker could add, remove or change any node in the header of the packet, disturbing route discovery and causing nodes to misroute packets. This attack could be detected by the previous node by means of enhanced passive acknowledgment.

Decrease the hop count (TTL) when receiving a packet, so that the packet will never be received by the destination. This attack could be detected by the previous node in route by enhanced passive acknowledgment.

4.3 Fabrication Attacks

An attacker could forge ROUTE ERROR packets causing nodes to incorrectly remove working routes from their route cache. In the worst case, this attack could prevent a node from being able to route any packets. Every time a node receives a ROUTE ERROR, it must remove this route from its route cache and broadcast this information to its neighbors. The difficulty for the attacker is to emit a ROUTE ERROR for a route that exists in the Route cache of the victim. The attacker must take part to the route too, otherwise it could not send this ROUTE ERROR without suspicion. This attack is difficult to detect for the nodes that are not mentioned in the ROUTE ERROR, since it is not possible distinguish a normal gratuitous ROUTE ERROR from a forged ROUTE ERROR.

Send spoofed ROUTE REQUESTs with subsequent sequence query id, so that the next ROUTE REQUESTs from the spoofed node will be discarded by the nodes since they already saw them. No ROUTE REPLY will come back since the destinations do not exist. Thus, when the victim will initiate new ROUTE REQUEST, nodes will discard them because they have already seen the same originating adress associated with the same id. Its detection is limited to the spoofed node when it receives a ROUTE REQUEST supposedly originated by itself and to nodes appearing in the route that have not received the request before.

Forge ROUTE REPLY packets causing nodes to misroute packets and to add incorrect routes in their route cache. The nodes that overhear it must update their route cache. Thus, they will misroute packets and consume energy and bandwidth. This is hard to detect.

Initiate frequent ROUTE REQUEST to consume bandwidth and energy and to cause congestion. The attacker could initiate ROUTE REQUEST for the same destination or for another destination every packet. Since ROUTE REQUEST are broadcast, it costs a lot of bandwidth and energy. In the first case, the event cannot be seen as a normal event. In the second case, there is an uncertainty over the behavior of the node.

4.4 Timing Attacks

Send route replies with a time not proportional to the length of the route. This can give more priority to long routes thus attracting routes to the attacker, or less priority to short routes thus avoiding the attacker. It is easy to mount. It can be observed when nodes wait for several routes to arrive and checking their length before adding them to the route cache.

5 Test-Bed Design

5.1 Overview

Our test-bed consists of several components. Whenever possible, we used components that are already publicly available and serve at least part of our purposes. We then proceeded to integrate the components by means of utilities that we modified to provide the functionalities we need and to glue the parts together.

The resulting architecture can be seen from Figure 1. We describe the use and integration of the main components in more detail in the subsequent sections and just list them briefly in the following.

- A Linux kernel module implementation of DSR called piconet [17] for routing. We modified by adding mechanisms to provide regular passive acknowledgment, our enhanced PACK for detection, and several attacks.
- The APE testbed [2] for scripting and mobility, and to integrate our distribution to be booted from CD.
- Netfilter [16] for capturing packets in promiscuous mode. We patched it so that it could handle packets promiscuously received using a new hook.
- PCMCIA card drivers pcmcia-cs for Linux, which we patched to enable promiscuous mode.

The setup for our experiments consists of 3 Pentium II laptops, 233 MHz, Linux kernel 2.4.19, APE

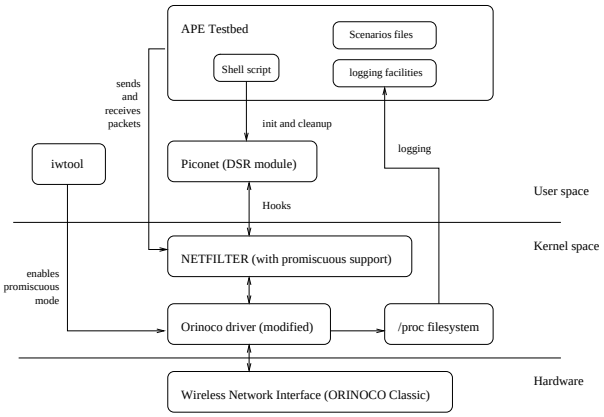

Figure 1: Test-bed Architecture

0.4, Redhat 7.2, and 1 Pentium IV laptop, 2.20 GHz, Linux kernel 2.4.20, Debian 3.0r1(woody). For all laptops we used Orinoco Classic Gold 802.11b cards, 11 Mb/s, driver pcmcia-cs-3.2.1 (orinoco 0.11b driver included).

5.2 Adding PACK to Piconet

The first problem to solve was to put the network interface in promiscuous mode. We use an hack of the orinoco_cs driver provided within the APE test-bed source files [2]. Using this modified driver, we could put the ORINOCO Classic card in promiscuous mode with the help of the `iwtool` command. We also try to use our implementation using the monitor mode with the ORINOCO card, but it fails because of two problems: we could not send any packets when the interface is in monitor mode and the captured packets do not activate the `NF_IP_PROMISCUOUS` hook in our modified netfilter. For more details on monitor mode, see [1] and [12].

When the interface is in promiscuous mode, it keeps all the packets it could overhear on the network. But, netfilter drops the "promiscuous" packets before they could be caught by any hook, so that it was impossible to process these packets within the netfilter framework. Since keeping the same global architecture was the easiest solution, we patched [18] netfilter to make it able to handle promiscuous traffic. This patch adds a `NF_IP_PROMISCUOUS` hook that catches all packets promiscuously received. With this improvement at hand, it was feasible to implement PACK over piconet.

We first add `prom_handler` which is called whenever the `NF_IP_PROMISCUOUS` catches a packet. After a check that ensures the originator belongs to the same subnet, `proc_pack_check` is called. This function parses the packet in order to find if it has a source route option, and in this case, retrieves the value of the segs_left field. Then, it looks for a packet that fulfills the tests for a packet to be a passive acknowledgment, as described in the previous section (i.e. source address, destination address, etc. must correspond). If it finds one, the packet is removed from the PACK queue. The packet promiscuously received is then dropped since it was not destined to the node itself.

When a packet is forwarded or originated, there is a check to know whether the next hop is the destination. In this case, the explicit network-layer acknowledgment is used with the function `ack_q_add`. Otherwise, we use the function `pack_q_add` instead the previous one, taking care to change the size of the packet when building it since it has no more ack request option in the header. `pack_q_add` is used when a node sends a normal packet (`dsr_send`), a fragmented packet (`dsr_fragment_send`), a route reply (`send_rt_reply`), a route error (`send_rt_error`) and when forwarding a packet that includes a source route option (`proc_sr_rt_opt`).

The function `pack_q_add` first retrieves the segs_left field from the header if it exists, so that this value can easily be found later when parsing the queue looking for a PACK. Then, it builds a clone of the packet that will be kept and sets a timer that expires after PASSIVE_ACK_TIMEOUT ms. When this occurs, `pack_timeout` is called. This function first checks if the maximum number of retransmissions is reached. If not, it resends the packet. Else, it adds an ack request option in order to use network-layer acks instead of PACK. To do that, the packet is first expanded using `skb_copy_expand`, then we fill the ack request option and add this packet to the ack queue. The packet is then processed as described in the initial implementation of piconet. The older packet waiting in the queue to be PACKed is removed.

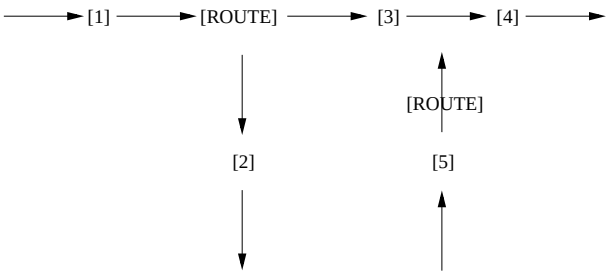

Figure 2: Netfilter architecture

5.3 Netfilter

Netfilter [16] provides a set of hooks in various points in the IPV4 protocol stack as shown in Figure 2. Packets enter on the left side of the diagram. They first pass some sanity checks (i.e. not truncated, IP checksum correct) and then are passed to the netfilter `NF_IP_PRE_ROUTING` [1] hook.
Next they enter the routing code, which decides whether the packet is destined for another interface, or a local process.
If the packet is destined for the machine itself, the netfilter framework is called again for the `NF_IP_LOCAL_IN` [2] hook, before being passed to the process (if any).
If it is destined to pass to another interface instead, the netfilter framework is called for the `NF_IP_FORWARD` [3] hook.
The packet then passes a final netfilter hook, the `NF_IP_POST_ROUTING` [4] hook, before being put on the network again.
The `NF_IP_LOCAL_OUT` [5] hook is called for packets that are created locally.

Now, we can see when each hook is activated. Kernel modules can register to listen to these hooks by using the `nf_register_hook` function. The module must define the priority of function within the hook, so that each function listening to this hook are called by order of priorities. When a function is called, it could then interact with the packet and manipulate it. The module can then tell netfilter to do one of these five things:

1. `NF_ACCEPT`: continue traversal as normal.
2. `NF_DROP`: drop the packet.
3. `NF_STOLEN`: we have taken over the packet; don't continue traversal.
4. `NF_QUEUE`: queue the packet.
5. `NF_REPEAT`: call this hook again.

`NF_ACCEPT` is used whenever we need to let a packet continue its way as if the module were not loaded: for example, if a node sends a packet that is addressed to a node that is not on its subnet (e.g. on a wired LAN). We use `NF_ACCEPT` to let the packet follow the standard kernel routing rules. When a node receives a packet that is destined to itself, it processes it and removes the DSR header, then it uses `NF_ACCEPT` to let the packet follow its way to the upper layers. `NF_DROP` is used quite often. for example, when a node receives a bad packet, it simply discard it by returning this `NF_DROP`, or, when it gets a packet promiscuously, it processes it and then releases it with `NF_DROP` since this packet is not destined to itself. `NF_STOLEN` is only used one time: when the kernel sends a packet, a node intercept it in the `LOCAL_OUT HOOK` and modifies its routing if needed. At the end, it uses `NF_STOLEN` to tell the kernel that it will send the packet itself and so, the kernel has nothing more to do with it. `NF_QUEUE` and `NF_REPEAT` are never used in our implementation.

5.4 Initial Piconet Implementation

5.4.1 Sending a packet

Piconet uses the netfilter framework to intercept the packets and manipulate them to implement the DSR protocol. Referring back to Figure 2, piconet uses the `PRE_ROUTE` [1] and the `LOCAL_OUT` [5] hooks. Additionally, the `POST_ROUTE` [4] hook is used for the DSR to IP gateway. In the next subsections, we explain the internals of piconet by following the journey of a packet through the whole implementation.

When we send a packet, this packet is intercepted by the `LOCAL_OUT` hook of our module. The function `local_out_handler` is called. Some preliminary tests check if the packet is destined to our subnet or if it is not a multicast for example. Then, the function tries to build a route entry that can be add to the packet.

The route table is first parsed using

lookup_route. If no route is found, then we send a route request by using send_rt_req. First, this function interacts with the route request cache (i.e. set the timer,...). Then, finish_send_rt_req is called.

Like all the other functions that output packets, that function first allocates some memory to have enough place to build our packet. Then, it matches the IP header struct and fills IP fields. Next, it adds a DSR header struct and fills the common DSR header. Now, it is time to add the DSR options to the packet. In this case, we only add a dsr_rt_req_opt, but if we send a normal data packet, we could add a dsr_src_rt_opt or maybe a dsr_ack_req_opt if we want network-layer acknowledgments. The important point is to be sure to allocate the right amount of memory for the packet.

When the packet is built, there are two different possibilities. Maybe, we do not receive a route reply for the moment and send_q_add is called to add the packet to the skb queue and we set a timer, so that the request could stop after a timeout expires. If we have already a route to send the packet, dsr_send is called.

This function adds the DSR header and builds the packet in the same way we did for the route request above. In this implementation, an explicit network-layer ack was used since it was the easiest solution. Therefore, we add an ack request option to every packet built in dsr_send. We also add the packet in an ack queue, that keeps a clone of all the packets waiting to be acknowledged, by using ack_q_add.

This function builds a clone of the packet, sets a timer and adds the clone to a list. If the timer expires, ack_timeout is called. That function manages the number of timeouts and retries.

5.4.2 Receiving a packet

When a packet enters the stack, the PRE_ROUTE hook calls pre_route_handler. It first checks if the packet implements the DSR protocol. Next, it parses the header in order to find all the options. Each time an option is found (PAD1, PADN, ROUTE_REQ, ROUTE_REPLY, ROUTE_ERROR, ACK_REQ, ACK, SRC_ROUTE), a corresponding function is called.

proc_rt_req_opt is called for a route request option. This function adds the reverse route to the originator in the route cache and then determines if we are the destination of the route request. If yes, it sends a route reply with send_rt_reply. Else, it checks whether we are not already in the route to avoid loops and if it is the first time we process this route request. In this case, the route request is rebroadcast using rebcast_rt_req.

proc_rt_reply_opt is called for a route reply option. This function only adds the route contained in the packet to the forward route cache.
proc_rt_error_opt is called for a route error option. It only removes the route from the route cache using remove_route.
proc_ack_req_opt is called for an ack request option. It sends an ack reply.
proc_ack_reply_opt is called for an ack reply option. This function first adds the neighbor address to the forward route to speed up the route discovery. Then, it finds and remove the packets from the ack queue.

Finally, proc_src_rt_opt is called when a source route option is found. It begins with some checks to determine if we are the destination of the packet or the gateway. In this case, no more processing is done. Else, it decreases the segs left field and adds route to source and to destination to the forward and reverse route table. After the forward address is determined, it is time to route the packet correctly. It bypasses the kernel routing with ip_route_input, otherwise the kernel will send it directly to the destination address of the IP header since this node is on the same subnet.

When all the options are processed, the function pre_route_handler removes the DSR header from the packet if the packet is destined to us and passes it to the upper layer (through LOCAL_IN).

5.5 Our Use of the APE Test-Bed

The APE Test-Bed provides some facilities to lead real world multi-hop wireless tests:

- Deployment of the tests is facilitated by the possibility to use a bootable CD-ROM or a package on a Linux or Windows machine.
- Scripted scenarios enable people to physically

carry out the experiments without prior instruction. Instructions are displayed on the laptops so that the tests could be easily reproducible.

- Possibility to add more routing protocols using scripts that initialize and cleanup sessions.
- Centralization of logs is done in a Master/Slaves architecture. This simplifies the post-analysis of the logs (e.g. synchronization).
- Visualization of node placements and movements can be done using a Java interface. This tool uses the radio signal strength (superspy) to build the map of nodes.
- Analysis tools are also provided to retrieve some basics metrics like virtual movement, data loss rate or path optimality.
- Mobility can be emulated by the mackill function which blocks out MAC addresses.
- It is extensible and based on a Linux environment.

More details can be found in [13].

We were able to build a personalized APE distribution quite easily to add the functionalities we require. First, we need to combine the sources of APE, a new kernel (2.4.19 in our case), and the sources of PCMCIA-CS (3.2.1 in our case). Then, we apply a patch for the kernel, so that it is able to use the mackill module which we use to disable the communication between two nodes at the MAC layer. This way, we can also emulate a loss of connection without having to move the nodes. Then, we apply a patch for the pcmcia-cs package that adds the so-called superspy and the promiscuous mode to the orinoco driver, as a prerequisite for the PACK function.

A routing protocol has to be implemented as a kernel module in order to be integrated in the APE test-bed, we do this with the piconet DSR module. Then, we define a script used to initiate and cleanup the module. This architecture makes APE very extensible and modular. We also add some new scenarios and modify the configuration file to match our requirements. Finally, after compiling the whole package (kernel and pcmcia-cs and our own modifications included), we make a bootable CD-ROM and a zip package.

If we use the zip package, the installation is very simple. We just need to extract it in the root directory `/`, and there is a script file that must be run to modify our bootloader. More details can be found in [3].

After installing APE and booting with this distribution, we start experiments by using the command `start_test`. It opens a menu in which we choose the scenario, the node representing the machine, and the protocol we want to use. When the experiments are completed, data gathering is done using a script.

6 Enhanced Passive Acknowledgment: More Watchdog Capabilities

In addition to the normal use of PACK, we benefit from the promiscuous mode to add more tests to detect attacks. Since the packets sent are logged in a queue waiting to be acknowledged by PACK, it is straightforward to check some additional fields to detect misbehavior in the flow of packets.

Thanks to the bi-directionality of the link-layer (IEEE 802.11b), a node is able to find out whether the next node forwards its packet if both nodes are still in the range of one another. This is possible because the node receives the packet in promiscuous mode when the next node forwards it. If it does not overhear the packet forwarded, it means that the next hop either did not forward it or that it did forward it but it was not overheard because the next-hop node moved out of range just after receiving the packet to be forwarded. With the PACK retransmission mechanism, the node waiting for the PACK resends the packet. If it does not get acknowledged, it emits a route error claiming that the next node is unreachable.

When a node promiscuously overhears the forwarded packet, it can additionally check whether it has been modified, and if so, whether the modifications result from a normal behavior or not.

The DSR draft [6] gives the fields we must check in order to consider the packet we receive as a PACK. By checking the four fields of the IP header, we can identify a packet uniquely so that we are sure we overheard one retransmission of the packet we forward. Next, the DSR

draft requires that if both packets have a source route option, then the segments left value in the overheard packet must be less than in the logged packet. This last check assures that the overheard packet is fresher than the logged one.

In practice, however, most Linux versions now sometimes set the IP identity field to zero for security reasons. This means for the use of passive acknowledgment, that if we want to identify packets uniquely, we have to use other pieces of information. We propose two solutions to this problem. The first is to generate a random identity number in the case when it has been set to zero. The second is to use the data contained in the packet to uniquely identify it, without modifying the IP identity. We only need to identify the packet uniquely if there is a need for retransmission and there would be several packets eligible. For our purpose of detection of partial dropping, it suffices to know that a packet was dropped that belonged to a particular path, regardless of which packet it was exactly in the sequence.

In order to implement the added watchdog capabilities to detect some attacks or events, we enhanced the passive acknowledgment we added to piconet so that every packet is completely checked for changes when we identify it as a passive acknowledgment. Thus, if the attacker changes one of the four IP fields we use to identify a PACK the regular PACK was not able to use our detection capabilities. We check the following fields and log if one of them changes:

- IP header: The TTL value must be decremented by only one.
- Route reply option(s): All fields.
- Route error option(s): All fields.
- Source route option: If the Salvage value is unchanged, all fields except Segs Left (we only check that this value decreases). If the Salvage flag changed, we only check Type, Last Hop External, First Hop External and Segs Left (must have decreased).
- Forged route error: a node can detect it, if the unreachable address in the route error option is its own.

This new functionality detects the changes well. It detects all the attacks we implemented that are based modifications in the header, as described in Section 7.

7 Attacks Implemented in the Test-Bed

7.1 Choice

After testing the PACK implementation we added to piconet, we focused our attention on attacks that could be detected by adding more watchdog capabilities in our implementation. We kept three types of attacks: header modification, partial dropping, and sending forged route error messages.

7.2 Header modification

7.2.1 Selfish attacks

First, we modify the PACK piconet in order to implement some selfish attacks that will help the attacker saving power. We keep three different modifications.

1. Last Hop External: We change this flag in the route reply option to make this route less interesting for the initiator of the route discovery. If it receives more than one route, it must prefer the ones that have this flag set to zero. This is done just by changing the value of `rtreply->lasthopx` in the `proc_rt_reply_opt` function if we are not the destination of the packet. We do the same for the Last Hop External and First Hop External fields in the source route option.
2. Removing itself from the route reply option: If a node removes its own address from the route reply option, it will not take part of the route and save power. To implement this, we add some code in the `proc_rt_reply_opt` function that looks for the address of the node itself, removes it and appends the addresses following it. It changes the blank line at the end of the route reply option to a PADN option.
3. Route error modification: If a node finds a route error option in the header, it modifies it in a self-

ish way. It changes the error source address to the address of the next-hop and the address of the unreachable node to its own, so that the next hops will remove it from their route cache. We add this attack in `proc_rt_err_opt` since it will modify a packet that includes a mandatory source route option.

We investigate how these modifications work in a real environment:

1. Last Hop External: Since piconet does not deal with this flag when determining the best route, this attack has no effect on the routing.

2. Removing oneself from the route reply option: This attack works in our simple test environment. Every time the source receives the modified route reply, the data packet it sends does not reach its destination because of the false route. If another route to the destination exists, then the route is changed to avoid the attacker.

3. Route error modification. The modification works and the receiver has to delete the route, thus avoiding the attacker.

7.2.2 Malicious attacks

Then, we add some others attacks that can be mounted by altering the header. These attacks will not help the attacker saving power, but only disrupting routes. We test the following:

1. Source route option altering: a node changes its address in the source route option so that the next hops will add an incorrect route in its route cache. This attack is implemented in the `proc_src_rt_opt` function.

2. Error destination address altering: A node changes the Error Destination Address in the route error option to discard route errors. When the destination of the route error will receive the packet, it will not be processed and the route will not be deleted.

How these modifications work in a real environment:

1. Source route option altering: This attacks works in our simple test environment. The answer of the destination of the modified packet never arrives. So that, this node must initiate a new route discovery process since he has no other route to destination.

7.3 Partial Dropping

This attack consists of dropping an arbitrary packet at a constant rate. The attacker will drop this packet whenever it is resent. To implement this attack, we add a new `drop_q` that keeps a log of the packet we drop. Whenever a packet is caught by the `NF_IP_PRE_ROUTING` hook, we first check if this packet has already been dropped using the `check_drop` function. In this case, we drop it again. Then, the packet enters the `drop_packet` function that checks if the packet must be dropped or not. In this case, we add the packet in the `drop_q` queue so that we could identify it later when it is resent.

This attack works well in our tests. We use a rate of one drop every ten packets. The previous hop detects the drop when the PACK timeout expires. It resends the packet that will be dropped again by the attacker and emits a route error after the explicit ACK timeout. Without link-layer acknowledgments, we have no reliable way to detect if the packet was lost because the next hop went out of range or dropped it intentionally. A heuristic, however, is that if subsequently a packet originating from the next hop is overheard, the node is in the range.

7.4 Fabrication of Forged Route Errors

An attacker could forge ROUTE ERROR packets causing nodes to incorrectly remove optimal routes from their Route cache. In the worst case, this attack could prevent a node from being able to route any packets. In our test implementation, we just emit a forged route error whenever the identification value in the IP header is a multiple of 3 and the packet includes a source route option.

The attack works well in our environment. The source of the packet removes this route from its route cache and starts a new route discovery process. This at-

tack can be detected when the next hop overhears the forged ROUTE ERROR that corresponds to a packet it just received. If the attacker does not forward the packet, it will be detected by the previous hop using passive acknowledgment.

8 Test-Bed Discussion and Future Work

Contrary to concerns raised against the watchdog to correctly detect packet dropping [10], the attacks we implemented were indeed detected successfully by use of our enhanced passive acknowledgment. The concerns were that for instance the partial dropping attack could lead to false conclusions in the case of ambiguous or receiver collisions. In all of our experiments, even with very high traffic load, we never experienced a single collision. Another potential objection to the effectiveness of a watchdog for the detection of dropping is that nodes could use power ranges just large enough to reach the previous hop but not the intended next hop if it is further away. This is very difficult to achieve, the power range adaptation in current off-the-shelf cards is not very precise, additionally nodes would constantly have to find out their distance to their neighbors that are potentially mobile.

Since we rely only on acknowledgments, passive or explicit, to send error messages and we currently have no link-layer notification in case a link breaks, a node moving out of range cannot be distinguished from a node that drops packets instead of forwarding them. This has to be taken into account when fixing thresholds for misbehavior detection.

The implemented attacks and their detection worked in all the experiments, therefore it would make little sense to show graphs on that. What is more illustrative is the performance of the network with our enhanced passive acknowledgment in place and compare it to the regular implementation with explicit acknowledgment, to see whether it has an impact on throughput, loss, and delay. The enhanced passive acknowledgment takes more computation due to the effort of overhearing, comparing and added checks for modification. On the positive side, however, passive acknowledgment does not need to send extra packets for acknowledgment and thus reduces the traffic. As exemplified by Figure 3 showing small packet size and 4 with large packet size, we found that the network performance was as least as good as when using regular explicit acknowledgment, sometimes even better. Even at very high traffic load, the computational overhead did not have any detrimental influence, and using passive acknowledgment mitigates the problem of duplicates that arise due retransmissions of packets that successfully arrived but the acknowledgments were lost.

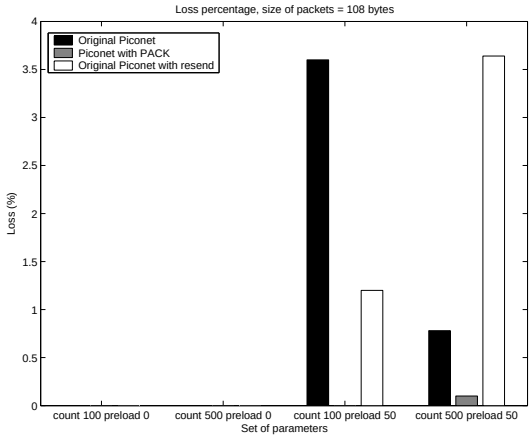

Figure 3: Percentage of lost packets for a number of pings (“count”), packet size 100B

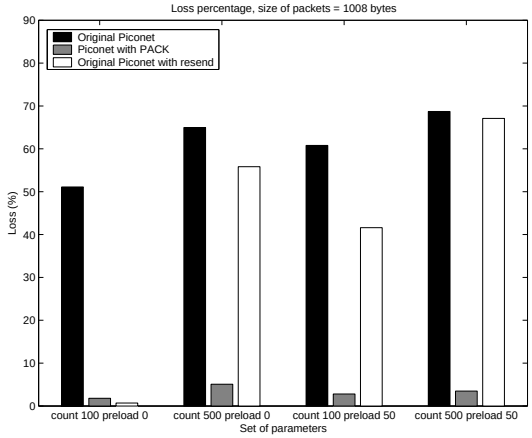

Figure 4: Percentage of lost packets for a number of pings (“count”), packet size 1000B

In the experiment shown, we had the laptops topology aligned in a row to enforce multi-hop forwarding. We varied the packet size, the number of pings, and the preload, i.e. how many packets are sent in a first burst.

The figures show an average over ten runs, the standard deviation was very small in all cases, the bars are absent when no loss occurred. We compared the original piconet implementation which uses explicit acknowledgments without retransmission, to versions modified by us, namely explicit acknowledgments with retransmission, and passive acknowledgment. Note that the loss rates might depend on the idiosyncrasies of the machines and drivers used, so we do not claim generality of these results. In the same vain, we observed that both the round-trip time of pings and the total time taken for batches of pings are reduced using passive acknowledgment, we are currently investigating the reasons for this, such as the role of the time it takes to send explicit acknowledgments and premature rerouting attempts in the case of no retransmissions.

In our experiments we set the timer for the passive acknowledgment to 100 ms. The timer is set when sending a packet and expires only if the packet has not been overheard being sent by the next-hop node. We found in all our experiments that the actual time to overhear was below 10 ms, even in the case of high traffic load. We therefore deem the expiry time of 100 ms more than sufficient, it can even be reduced if necessary.

We have implemented both attacks and their detection. In order to render misbehaved nodes harmless, this detection has to be followed by a response, the most effective being isolation. Our test-bed can be extended by mechanisms to disseminate the detection information gained by use of our enhanced passive acknowledgment. This information can then serve as an input to a reputation system to serve as a basis for decision making on a suitable response. The response itself can then also be added to our test-bed to evaluate its effectiveness in a real environment.

Although we intend to use the test-bed to implement our own reputation system based mechanism, we envision its use also for the community to evaluate different protocols. We are in the process of making our code and detailed methodology public, so that the test-bed can be used to investigate both potential attacks and countermeasures.

9 Conclusions

In the quest for a real network evaluation, we modified and integrated several components to form a test-bed suitable for the investigation of the feasibility of both misbehavior attacks and their detection on mobile ad-hoc networks. We built the test-bed, implemented several attacks, and demonstrated their effectiveness. We enhanced the passive acknowledgment mechanism, where nodes overhear the transmission of neighboring nodes to verify the reception of packets, to allow for the detection of a range of attacks. We built this extended passive acknowledgment for detection into the test-bed and evaluated its capabilities.

Watchdogs, as detection components for mobile ad-hoc networks have been called, as implemented in our extended passive acknowledgment mechanism can indeed detect a number of attacks on mobile networks such as packet dropping, and several types of packet modification and fabrication. The capabilities of watchdogs are most limited in the case of packet fabrication.

We propose our test-bed for the use of the community to evaluate attacks, detection, reputation, and response mechanisms.

References

[1] Airsnort homepage. http://airsnort.shmoo.com/, August 2003.

[2] Ad hoc protocol evaluation testbed. http://apetestbed.sourceforge.net, November 2002.

[3] How to build, install and run the ape testbed. http://apetestbed.sourceforge.net/ape-testbed.ps, November 2002.

[4] Sorav Bansal and Mary Baker. Observation-based cooperation enforcement in ad hoc networks. Technical Report, 2003.

[5] Sonja Buchegger and Jean-Yves Le Boudec. Performance Analysis of the CONFIDANT Protocol: Cooperation Of Nodes — Fairness In Dynamic Ad-hoc NeTworks. In *Proceedings of*

IEEE/ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC), Lausanne, CH, June 2002. IEEE.

[6] Y. Hu D. Johnson, D. Maltz. The dynamic source routing protocol for mobile ad hoc networks (dsr). http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-09.txt, April 2003.

[7] Y. Huang, W. Fan, W. Lee, and P. S. Yu. Cross-feature analysis for detecting ad-hoc routing anomalies. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003), Providence, RI*, pages 478–487, May 2003.

[8] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[9] David A. Maltz, Josh Broch, and David B. Johnson. Experiences designing and building a multihop wireless ad hoc network testbed. Technical Report CMU-CS-99-116, CMU School of Computer Science, March 1999.

[10] Sergio Marti, T.J. Giuli, Kevin Lai, and Mary Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Proceedings of MOBICOM 2000*, pages 255–265, 2000.

[11] Pietro Michiardi and Refik Molva. CORE: A collaborative reputation mechanism to enforce node cooperation in mobile ad hoc networks. Sixth IFIP conference on security communications, and multimedia (CMS 2002), Portoroz, Slovenia., 2002.

[12] Linux and lucent wireless cards. http://www.goonda.org/wireless/lucent, June 2003.

[13] E. Nordstrom. Ape - a large scale ad hoc network testbed for reproducible performance tests. http://www.csd.uu.se/courses/course-material/xjobb/docs-reports/Nordstrom-2002.pdf, June 2002.

[14] Krishna Paul and Dirk Westhoff. Context aware inferencing to rate a selfish node in dsr based adhoc networks. In *Proceedings of the IEEE Globecom Conference*, Taipeh, Taiwan, 2002. IEEE.

[15] Boulder Pervasive Communications Laboratory, University of Colorado. The click dsr router project. http://pecolab.colorado.edu/.

[16] H. Welte R. Russel. Linux netfilter howto. http://www.iptables.org/documentation/HOWTO/netfilter-hacking-HOWTO.html, July 2002.

[17] A. Song. piconet ii, a wireless ad hoc network for mobile handeld devices. http://piconet.sourceforge.net.

[18] S. Zander. http://www.fokus.gmd.de/research/cc/glone/employees/sebastian.zander/private/netfilter-prom-patch.tgz, November 2001.

[19] Yongguang Zhang and Wenke Lee. Intrusion detection in wireless ad-hoc networks. In *Proceedings of MOBICOM 2000*, pages 275–283, 2000.

[20] Yongguang Zhang and Wei Li. An integrated environment for testing mobile ad-hoc networks. In *Proceedings of IEEE/ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC)*, Lausanne, CH, June 2002. IEEE.