Adversarial Testing of Wireless Routing Implementations

Md. Endadul Hoque
Purdue University
mhoque@cs.purdue.edu

Hyojeong Lee
Purdue University
hyojlee@cs.purdue.edu

Rahul Potharaju
Purdue University
rpothara@cs.purdue.edu

Charles E. Killian
Purdue University
Google Inc.
ckillian@cs.purdue.edu

Cristina Nita-Rotaru
Purdue University
crisn@cs.purdue.edu

ABSTRACT

We focus on automated adversarial testing of real-world implementations of wireless routing protocols. We extend an existing platform, Turret, designed for general distributed systems, to address the specifics of wireless routing protocols. Specifically, we add functionality to differentiate routing messages from data messages and support wireless specific attacks such as blackhole and wormhole, or routing attacks such as replay attacks. The extended platform, Turret-W, uses a network emulator to create reproducible network conditions and virtualization to run unmodified binaries of wireless protocol implementations. Using the platform on publicly available implementations of two representative routing protocols we (re-)discovered 14 attacks and 3 bugs.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Wireless communication; C.2.m [**Miscellaneous**]: Security

General Terms

Design, Security

Keywords

Adversarial testing; Virtualization; Wireless; Routing

1. INTRODUCTION

Mobile ad-hoc networks allow a set of wireless nodes to communicate with each other without any central infrastructure. As traditional routing protocols do not perform well in a constrained environment such as wireless networks, significant work has been put into designing routing protocols for wireless networks. Examples include proactive protocols such as DSDV [32], reactive protocols such as AODV [33] and DSR [21], or hybrid protocols such as DST [34]. Moreover, due to increased threats that exist in wireless networks, several secure routing protocols have been designed. Examples include: SAODV [38], ODSBR [11], ARAN [36], Ariadne [20]. Several of these protocols such as AODV, ARAN were implemented and are available from public repositories.

Given the importance of routing as a fundamental component of wireless networks, many protocols have been subjected to model checking the design [12] and testing the simulator-based implementation [6,39]. While model checking greatly helps to verify the validity of the model, real-world implementations often bring vulnerabilities that model abstraction does not capture. Also, while simulators provide easier and simpler ways to describe a protocol, the simplicity sacrifices some aspects of realism such as interaction with the operating system components.

Recent research [23,24] showed the importance of performing adversarial testing where software is tested beyond just basic functionality by examining edge cases, boundary conditions, and ultimately conducting destructive testing. Adversarial testing makes protocols more robust to arbitrary and extreme conditions and can lead to discovering vulnerabilities in implementations, many of which might have not occurred in simulator-based implementations.

Previous work related to wireless routing implementations has focused exclusively on performance comparison across protocols [14, 16], or using test-beds to investigate properties of the network stack such as performance of TCP in multihop ad hoc networks [13, 16].

In this paper, we focus on adversarial testing of real-world implementations of wireless routing protocols. We consider attacks and failures that are created through manipulation of protocol messages and are specific to wireless routing protocols, having a global impact on the protocol performance. We use an experimentation environment that allows binaries to run in their native operating systems while limiting the impact of noise and interference on the system performance. Our aim is to automatically discover potential attacks by exploiting vulnerabilities in protocol design and implementation. Our contributions are:

- We extend an existing platform, Turret [23], designed for general distributed systems, to address the specifics of wireless routing protocols. The platform uses a network emulator to create reproducible network conditions and virtualization to run unmodified binaries of wireless protocol implementations. The platform requires the user to provide a description of the protocol messages and corresponding performance metrics. We present Turret-W, an extension of an existing platform, that includes a wireless malicious proxy that differentiates routing messages

WiSec'13, April 17–19, 2013, Budapest, Hungary.

from data messages and supports wireless specific attacks such as blackhole and wormhole, or routing attacks such as replay attacks. Our approach is cost effective as compared to the hardware and manpower costs required by the approaches in [14].

- We demonstrate attack discovery with Turret-W using detailed case studies on two representative wireless routing protocols: a reactive protocol AODV, and a secure reactive protocol ARAN, whose implementations we obtained from public repositories. We found 1 new and 7 known attacks in AODV, and 6 known attacks in ARAN, for a total of 14 attacks. While most of attacks we found are protocol level attacks, one attack in AODV was solely an implementation level attack and such an attack could have been discovered by testing the actual implementation under adversarial environments.
- We show that Turret-W also helps to find bugs because it implicitly supports a testing environment that is realistic and controllable. Bugs are different from attacks in that they can cause performance degradation in benign executions. We discovered 3 bugs in target implementations, 2 in AODV and 1 in ARAN.

The rest of the paper is organized as follows. §2 provides an overview of the platform we use in this paper, §3 describes our methodology and presents two case studies. §4 describes related work and §5 summarizes the paper.

2. PLATFORM OVERVIEW

Our goal is to test real-world implementations, where the network conditions can be reproducible and also isolated from outside world interference. Platforms that leverage virtualization and some form of emulation fit this profile. We selected Turret [23] because it provides these capabilities. However, Turret is not designed for wireless networks, but assumes more general message-passing systems. We first give an overview of Turret, the platform that we built on, and then describe how we extended it to support wireless routing protocols. We refer to Turret with our extension as Turret-W.

2.1 Turret Overview

Turret is a platform for performance attack discovery on unmodified distributed system binaries running in realistic environments. Turret uses virtualization to run arbitrary operating systems and applications, and network emulation to connect these virtualized hosts in a realistic network setting. Turret requires a description of the external API of the message protocol, and a set of metrics that capture the application performance of the system.

Specifically, Turret uses KVM [17] virtualization techniques, to allow several virtual machines, each acting as an individual node, to run on the same physical host. Each node can then run an application and communicate with other nodes through an emulated network, achieved through NS-3 [7] [1]. Turret maps each virtualized node with a shadow node inside NS-3 by creating a *Tap-Bridge* connection between the virtualized node and its corresponding end node.

[1]Note that Emulab [4], MobiNet [26], Orbit [9] could also conceptually replace NS3. Emulab with fixed wireless provides more realism. However, the approach provides less reproducible results because of unwanted disturbance on the wireless channel and requires a separate implementation of the malicious version of the tested routing protocol.

A controller bootstraps the system, (i.e. starts NS-3 and runs application binaries inside the virtual machines), and instructs nodes whether they will act as benign or malicious. A node marked as malicious will then be equipped with a *malicious proxy* which is a part of the Tap-Bridge layer on the NS-3 shadow node. This malicious proxy takes care of intercepting packets and modifying them according to an *attack strategy*. To make this possible, the malicious proxy requires the user to provide it with a list of *messages formats* that the protocol relies on. This way, different fields inside a message can be automatically modified based on the selected attack strategy.

Action	Action Description	Parameter
Drop	Drops a message	Drop probability
Delaying	Injects a delay before it sends a message	Delay amount
Duplicating	Sends the same message several times instead of sending only one copy	Number of duplicated copies
Diverting	Sends the message to a random node instead of its intended destination	None

Table 1: Message delivery actions in Turret

Action	Action Description	Parameter
LieValue	Changes the value of the field with a specified value	The new value
LieAdd	Adds some amount to the value of the field	The amount to add
LieSub	Subtracts some amount from the value of the field	The amount to subtract
LieMult	Multiplies some amount to the value of the field	The amount to multiply
LieRandom	Modifies the value with a random value in the valid range of the type of the field	None

Table 2: Message lying actions in Turret

Turret supports two types of malicious actions: *Message Delivery Actions* which affect when and where a message is delivered (see Table 1) and *Message Lying Actions* which affect the contents of a message (see Table 2).

2.2 Turret Limitations for Wireless Routing

Turret does not focus on routing protocols or on wireless networks and suffers from the following limitations:

Distinguishing between control plane and data plane: While Turret can inject attacks and faults into any message-oriented protocol, it does not differentiate the data layer from the routing layer. As Turret injects attacks based on message types, it can not inject attacks on packets in bulk data transfer application such as ftp. In the case of routing, many data plane attacks or degradation in performance can be amplified if the routing mechanism is disrupted and it does not matter if the application is a message oriented application or a bulk data transfer application. For wireless networks, the separation is needed to support basic attacks such as *blackhole* in which an attacker will drop all data packets but participate in the routing algorithm correctly.

Replaying packets: Turret does not provide the functionality to replay packets. When replaying a packet, an attacker records another node's valid control messages and resends them later to other benign nodes via legitimate channels. This causes other nodes to add incorrect routes to their routing table. Such attacks can be used to impersonate a specific node or simply to disturb the routing plane.

Establishing side-channels (wormholes): Turret does not support colluding attacks because finding an effective

colluding strategy in distributed systems results in a state space explosion. However, an attack specific to wireless networks that requires coordination between two attackers and shown to be very detrimental is the wormhole attack where two colluding adversaries cooperate by tunneling packets between each other to create a shortcut in the network.

2.3 Turret-W Overview

We modified Turret to address the above limitations. The new platform, Turret-W, is shown in Figure 1. The *controller* is the core component that drives the system. It generates a topology file for the network emulation layer using a configuration file provided by the user that specifies various parameters such as the network topology, number of nodes, number of malicious nodes. Then it starts the virtual machines and binds each of them to the underlying network emulation layer. It then loads the routing service (e.g., AODV) at the routing layer and instantiates the application (e.g., *iperf*) at the application layer. It accepts the list of attack strategies created by the strategy generator and injects them into the malicious proxy. Finally, it collects log messages that are useful in post-processing for estimating application performance running on top of the routing protocol.

Wireless network emulation: We configured the network emulator, NS-3, to emulate WiFi links. Like in Turret, virtual machines operate on top of a *network emulation layer* provided by NS-3. In principle, each virtual machine acts as an individual node in the system and is connected to a corresponding shadow node inside NS-3 using a Tap-Bridge connection. A Tap-Bridge connection makes an NS-3 net device appear as a local device inside the virtual machine thereby allowing the virtual machine to use the underlying net device as if it were its own net device for WiFi transmission. The network emulation layer creates a virtual multihop wireless environment to transmit packets from a source to a destination virtual machine.

Supporting wireless routing specific actions: We modified the malicious proxy to differentiate between packets originating from the routing layer and the application layer. We achieve this by utilizing the port number to differentiate between application and routing related packets. This makes it possible to implement a blackhole attack wherein a malicious node acts benign at the routing layer but selectively/entirely drops packets originating from the application layer. We implement the wormhole attack as follows: the private channel is implemented as part of the malicious proxy inside NS-3 and hence the virtual machines are agnostic about the channel. According to the routing protocol, if a node does not hear back from one of its neighbors, the node considers the route to the neighbor as a stale route. Therefore, to convince one colluding node that the other colluding node is its neighbor, we allow the colluding nodes to send their beacon messages (e.g., *hello*) to each other. However, to prevent a colluding node at one end from assuming that the benign neighbors of the other colluding node are its neighbors, the colluding nodes do not forward the beacon messages received from their benign neighbors over the private channel. All other routing protocol messages are forwarded by the colluding nodes over the private channel so that they can perform the wormhole attack in the route discovery process. As a result, Turret-W supports all the malicious actions presented in in Tables 1, 2, and 3.

Attack strategy generation: The strategy generator is responsible for generating a list of attack strategies that a protocol-under-testing should be tested against.

These sequence of strategies are generated based on the malicious actions given in Tables 1 and 2 along with a value that decides the severity of that action. To support the additional wireless specific attacks listed in Table 3, we have extended Turret's basic set of malicious actions with replay, blackhole and wormhole attacks.

Action	Action Description	Parameter
Replay	Records valid control messages from a node and resends them to other benign neighbors	None
Blackhole	Drops all data packets but participates in the routing protocol correctly	None
Wormhole	Creates a wormhole between two colluding nodes and tunnels packets between each other	Types of the control messages to be tunnelled
Wormhole with blackhole	Creates a wormhole between two colluding nodes and tunnels routing packets between each other, but drops all data packets	Types of the control messages to be tunnelled

Table 3: Malicious actions added by Turret-W

3. EXPERIMENTAL RESULTS

We demonstrate our platform on real-world implementations of two well known on-demand wireless routing protocols: AODV [33], and ARAN [36], whose implementations we obtained from [1, 2]. In this section, we describe our experimental setup and discuss our case studies on these protocols. We summarize all the attacks and bugs reported by Turret-W in Table 4.

3.1 Experimental Setup

All our experiments are performed on a Dual-Quad core Intel(R) Xeon(R) CPU E5410@2.33GHz with 8 GB RAM host machine. We use Ubuntu 10.04.4 LTS to serve as the host OS. In all experiments, we use 12 VMs, each allocated 128 MB RAM. For AODV, we use Debian 6.0.5 with Linux Kernel 2.6.32-5-686 as the guest OS. One of the advantages of our platform is that it allows us to support exactly the environment for which a binary was compiled. For instance, since ARAN requires an older kernel, we use Fedora Core 1 with Linux kernel 2.4.22-1.2115.nptl.i386 as the guest OS.

Our emulated network is a multihop wireless adhoc network. For the 802.11 MAC layer, we used 802.11a with bit rate of 6 Mbps. We performed our experiments using a static grid topology. As an application, we run *iperf* [5], a network benchmarking tool, on the VMs. In all the experiments, the performance of the application we report are averaged over ten runs.

As we are interested in performance attacks, we obtain a performance baseline using *benign testing*, where we randomly select pairs of source and destination nodes and transfer a stream of UDP packets between them for 30 seconds. Since we do not intend to stress the protocol or its implementation, we select a lower data rate of 128 Kbps so that the impact of attacks can be easily observed – a low packet delivery ratio implies an attack [11, 31].

In *adversarial testing*, we randomly select malicious nodes and inject malicious strategies during the entire execution. For all attack strategies applied to routing messages, the malicious node drops application packets with a probability of p (= 0.3 in our case). The purpose of using $p = 0.3$ is two-fold: 1) it helps differentiate if the effect of the attack is route discovery failure or packet dropping, and 2) since

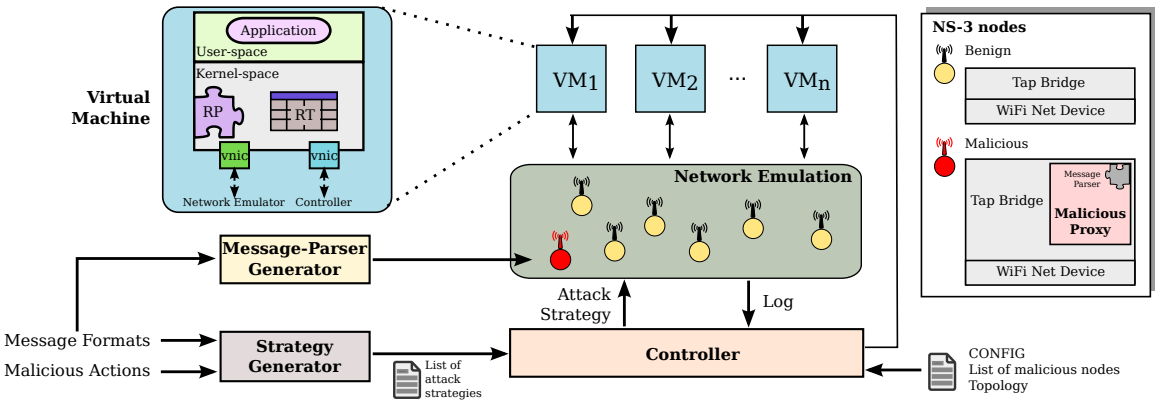

Figure 1: Turret-W (RP:Routing Protocol, RT:Routing Table and VNIC:Virtual Network Interface Card)

we focus on the insider attackers, $p = 0.3$ characterizes the less aggressive behavior which is typical behavior of insider attackers. We vary the total number of adversaries in the network from 1 to 4 exhibiting a homogeneous behavior, i.e., we inject the same attack strategy to each malicious node.

To demonstrate the effect of blackhole attacks and wormhole attacks, we perform experiments with three different configurations of adversaries: blackhole with one adversary, blackhole with two adversaries, combination of wormhole and blackhole with two colluding adversaries. When a blackhole attack strategy is injected, an adversary participates benignly in the routing protocols but drops 100% of application packets. The effect of wormhole is the most noticeable in terms of application performance when we combine the blackhole strategy with the wormhole strategy. Note that except the cases of blackhole and/or wormhole attacks, we use the packet dropping probability $p = 0.3$ in all other malicious executions as mentioned above.

As a performance metric, we use *packet delivery ratio* (PDR), i.e., a ratio of the total number of packets received by the receiver to the total number of packets sent by the sender. PDR is easy to measure and does not require any instrumentation in routing protocol implementations. We plan to investigate other security metrics for future work. We formally define an attack as follows:

***Definition 1 - Performance Attack**: When the difference in the performance metric for a benign execution to that of a malicious execution exceeds by a threshold, δ, we say that the attack strategy has resulted in a successful attack.*

Here, δ is a system parameter and is dependent on the system-under-test. In our experiment we decide to choose 0.2 as our threshold.

One of the by-products of using our system is the ability to find bugs that have an impact on performance. We define a bug as follows:

***Definition 2 - Performance Bug**: An implementation-level fault that produces a degradation of performance during a benign execution.*

3.2 Case Study: AODV

We now describe how we used Turret-W to test AODV [33]. We omit the graphs due to lack of space.

Implementation used: We use AODV-UU-0.9.6 implementation, publicly available from [1] that is RFC 3561 [10] compliant. The AODV-UU consists of two components — a loadable kernel module (`kaodv`) and a user space daemon process (`aodvd`). The kaodv module intercepts and handles network packets by registering hooks with the Linux kernel using the netfilter framework. We use the default values for configuration related parameters presented in [1, 10].

During the benign testing (to obtain a baseline to compare against) of AODV-UU, we discovered two unknown implementation bugs caused by a subtle interplay between the AODV-UU code and the kernel.

Bug 1. Kernel interaction order. In an attempt to measure TCP streaming performance between a source and a destination that is multiple hops away, we observed that packets were not getting delivered in the benign case. We investigated the cause and identified this bug. By design, whenever an application sends a packet with a destination to which the route is either invalid or unavailable, `kaodv` is supposed to hold the packet and notify `aodvd` to perform a route discovery. After finishing the route discovery, `aodvd` should notify the kernel to update the routing table and the `koadv` module to release the withheld packet.

In the implementation, however, the order of notification upon completion of a route discovery was incorrect, i.e., in the reverse order. This bug could not have been discovered if we had not attempted to measure TCP streaming performance where the first packet, *i.e.*, SYN packet is crucial to establish the connection. However, we observed packet loss when initially using UDP, but like others, we attributed this to the lossy behavior of UDP inside the wireless channel. We fix the bug by reversing the order of the two notifications.

Bug 2. Route packets harder. In the process of obtaining a baseline using a network benchmark tool *iperf*, we observed performance degradation over time despite the route being available and valid in the routing table.

When the kernel transport layer hands-over any locally generated packet to the IP layer, `kaodv` receives the control of the packet via a hook registered with *netfilter*. So, `kaodv` is responsible for returning a value to netfilter so that netfilter can decide what to do – accept/drop/ignore the packet or call the hook again.

When `kaodv` receives the control for a packet and already has a valid route, `kaodv` notifies netfilter to continue processing the packet by returning `NF_ACCEPT`. On receiving `NF_ACCEPT`, netfilter sends the packet down the network stack without performing any further iptables tests [19]. As a result, netfilter does not send the packet to the correct next hop node on the route to the destination. We fix this bug by invoking `ip_route_me_harder()` inside `kaodv` before returning `NF_ACCEPT`.

Attack causing crashing. We discovered an implementation attack that can cause all neighbors of a malicious node to crash. When a malicious proxy modifies an RREQ

Protocol Impl.	Discovery Type	Name	Description
AODV-UU 0.9.6 [1] Updated: April 13, 2011	Attack*	Lie RREQ type 2	Lie about RREQ message type by setting to 2 (RREP) (causes crashing)
	Attack [35]	Lie RERR type 1	Lie about RERR message type by setting to 1 (RREQ)
	Attack [35, 38]	Lie RREP hop 0	Lie about the hop count in route response to be 0
	Attack [35]	LieAdd RREQ reqid 10	Increment the route request id of route request by 10
	Attack [35, 38]	LieAdd RREP destsq 10	Increment the destination sequence number of route response by 10
	Attack [35, 38]	Replay RREP	Replay both route response and hello messages
	Attack [38]	Blackhole	Drop all data packets
	Attack [35, 38]	Wormhole + Blackhole	Colluding malicious nodes drop all data packets
	Bug*	Kernel interaction order	Notifies the two components about the route discovery in a wrong order
	Bug*	Route packets harder	Returning `NF_ACCEPT` from hooks causes Netfilter not to check iptables
ARAND 0.3.2 [2] Updated: Jan 31, 2003	Attack [25]	Drop RDP 100%	Drop each route request message
	Attack [25]	Delay REP 2s	Delay forwarding of route response message by 2 seconds
	Attack [25]	Divert REP	Divert route response message
	Attack [25]	Drop ERR 100%	Drop route error message
	Attack [11]	Blackhole	Drop all data packets
	Attack [11]	Wormhole + Blackhole	Colluding malicious nodes drop all data packets
	Bug*	Wrong postal address	Intermediate nodes forward REP to the source instead of the next hop

Table 4: Attacks and bugs (re-)discovered by Turret-W. Attacks/bugs with (*) means newly discovered.

message to be an RREP, a recipient processes this altered RREQ message as an RREP message. The base RREP message (20 bytes) is smaller in length than a base RREQ message (24 bytes) [10]. Therefore, a recipient of the malformed RREQ assumes it as an RREP with extensions [10] which causes AODV-UU to crash. The malicious node becomes successful in exploiting an *integer overflow* and *buffer overflow* vulnerability in the AODV-UU code. This is solely an implementation level attack that cannot be discovered in a benign environment, however it can cause significant damage. It can be fixed by cautious type and boundary checking with an awareness of possible adversarial attempts.

Attacks caused by basic malicious actions. We rediscovered several attacks on AODV-UU based on message delivery and lying actions which decrease the PDR below the accepted threshold. Note that the AODV protocol itself is susceptible to these attacks [35, 38]. In case of our benign experiments, we observe a 98% PDR.

Replay RREP. By replaying a RREP message received from a node, an adversary can fool its benign neighbors to believe that they are the neighbors of the node which, in reality, are at least two hops away. This attack is more damaging than others because replaying the periodic hello messages causes these pseudo-links never to expire. We observe the PDR drops from 75% to 17% as the number of adversaries increases.

LieAdd RREP destsq. Whenever a node receives a control packet from another node with the destination sequence number higher than what it has in its routing table, the node selects the route via this other node. A malicious node can add a positive value with the destination sequence number of an RREP message which causes the recipient to select the route through the malicious node. We observe that this attack results in at most 56% drop in PDR.

LieAdd RREQ reqid. Each RREQ message is uniquely identified by the *request identifier* in conjunction with the originator's IP. For each new route request, the request identifier is incremented by one. No node ever responds to an older RREQ message. A malicious node tricks the destination to respond to an RREQ with a future request identifier so that the source will be left with only one available route, i.e., through the malicious node. We observe that this attack causes the PDR to drop from 78% to 62% with the increase in the number of adversaries.

Blackhole/wormhole attacks. To evaluate the performance of AODV-UU in the presence of a blackhole attacker, we allow an intermediate malicious node to drop all the data packets. Later, we introduce an additional blackhole node that can collude with the other blackhole node via a private channel to perform a wormhole attack. The PDR drops from 70% to 50% with the increase in blackhole nodes, whereas the PDR drops to 40% in case of the wormhole attack.

3.3 Case Study: ARAN

We now describe how we used Turret-W to test ARAN [36].

Implementation used: We rely on the implementation of ARAN called `arand`-0.3.2, publicly available from [2]. This user space routing daemon built for Linux kernel 2.4 relies on the Ad hoc Support Library (ASL) [3]. For the cryptographic functionalities, it uses OpenSSL [8]. We use the default values for parameters as used in [2].

Bug. Wrong postal address. We discovered an implementation bug during the benign experiments in the setting of a multi-hop wireless network. By design, a route discovery request should be flooded via broadcast and the response should be delivered via unicast following the reverse path. However, in the implementation, upon receiving a response, an intermediate node tries to send the packet directly to the source node instead of its correct next hop node. When the intermediate node is not a direct neighbor of the source node, the route discovery will fail. We fix this bug by using the correct next hop address.

Attacks caused by message forwarding actions. We rediscovered several attacks on `arand` based on malicious delivery that have intense impact on the application performance. Note that the ARAN protocol itself is susceptible to these attacks [11, 25]. In case of our benign experiments, we observe 99% PDR.

Divert REP and *Drop ERR*: By diverting a route reply (REP) message and by dropping a route error (ERR) message, a malicious node can cause the most damage among these attacks. Both these messages are sent via unicast by design, and therefore, if an intermediate malicious node drops or diverts these messages , the upstream nodes on the route remain unaware of the on-going attack. Four malicious nodes can drop the PDR to below 30% by diverting REP messages and to 40% by dropping ERR messages.

Drop RDP. An intermediate malicious node can drop a route discovery (RDP) message instead of re-broadcasting. However, this attack causes a slow decrease in PDR because every intermediate node re-broadcasts the RDP packet and therefore, even if a malicious node does not forward the RDP, the destination eventually receives the RDP message from other benign node(s). The PDR can drop below 60% with the increase in adversaries.

Blackhole/wormhole attacks. We evaluate `arand` in the presence of blackhole/wormhole attackers in the network. In the presence of one blackhole attacker, the PDR drops to 80%. Adding another blackhole node drops the PDR to 42%. However, when two blackhole nodes collude with each other to perform a wormhole attack, the PDR drops to 28%.

4. RELATED WORK

Model-checking [18, 28] and theoretical verification [29] techniques have been used to verify the correctness of models or designs. While theoretical techniques have been helpful to show the correctness of the model, protocols and some characteristics of the environment, the high-level descriptions often do not exactly match the real-world implementations. Many vulnerabilities are introduced during the process of implementation due to the complexity and performance optimization that was not originally considered during the design phase. Hence, verifying correctness and robustness of implementations is another type of challenge [29].

To that end, exploration based model checking techniques [30] and systematic fault injections [15, 27] test real implementations under various conditions. However these works do not consider adversarial environments.

There have been some recent effort on finding attacks automatically in implementations [22–24, 37]. Kothari et al. [22] automatically find attacks that manipulate control flow by modifying messages using static analysis. Stanojevic et al. [37] automatically search for gullibility in two-party protocols by dropping packets and modifying packet headers. Lee et al. [24] automatically discover performance attacks caused by insiders in distributed systems without requiring implementation modification. However [22] requires a priori knowledge about vulnerability. All these works except [23] require the implementation to be written in specific languages.

Turret [23] is closely related to our work as it tries to find attacks on unmodified distributed system implementations in realistic environments. However, it focuses on general distributed systems, and does not consider wireless environments. We extend Turret for wireless environments and add features to support wireless routing protocols such as separation of the data layer from the routing layer so that attackers can independently attack both the layers, and side channels that attackers can use to collude.

5. CONCLUSION

Given the importance of routing as a fundamental component of wireless networks, it is critical to subject their implementations to adversarial testing before deployment. To aid developers in this task, we develop Turret-W, an adversarial testing platform for wireless routing protocol implementations with minimal physical resources. We demonstrate our system by evaluating publicly available real world implementations of AODV and ARAN. In total, we (re-)discovered 14 adversarial attacks capable of either crashing the benign nodes or deteriorating their performance by disrupting the routing service and 3 implementation bugs that have an impact on the application performance.

6. REFERENCES

[1] AODV-UU. `http://sourceforge.net/projects/aodvuu/`.
[2] ARAN. `http://prisms.cs.umass.edu/arand/`.
[3] ASL. `http://sourceforge.net/projects/aslib/`.
[4] Emulab - network emulation testbed. `http://www.emulab.net/`.
[5] Iperf. `http://sourceforge.net/projects/iperf`.
[6] Network Simulator 2. `http://www.isi.edu/nsnam/ns/`.
[7] Network Simulator 3. `http://www.nsnam.org/`.
[8] OpenSSL toolkit. `http://www.openssl.org/`.
[9] Orbit. `http://www.orbit-lab.org`.
[10] RFC 3561. `http://tools.ietf.org/html/rfc3561`.
[11] B. Awerbuch, R. Curtmola, D. Holmer, C. Nita-Rotaru, and H. Rubens. ODSBR: An on-demand secure byzantine resilient routing protocol for wireless ad hoc networks. *TISSEC*, 2008.
[12] F. De Renesse and A. Aghvami. Formal verification of ad-hoc routing protocols using spin model checker. In *IEEE Melecon*, 2004.
[13] S. M. ElRakabawy and C. Lindemann. A practical adaptive pacing scheme for TCP in multihop wireless n/ws. *ToN*, 2011.
[14] R. S. Gray, D. Kotz, C. Newport, N. Dubrovsky, A. Fiske, J. Liu, C. Masone, S. McGrath, and Y. Yuan. Outdoor experimental comparison of four ad hoc routing algorithms. In *Procs. of MSWiM*, 2004.
[15] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. Fate and Destini: a framework for cloud recovery testing. In *NSDI*, 2011.
[16] A. Gupta, I. Wormsbecker, and C. Wilhainson. Experimental evaluation of TCP performance in multi-hop wireless ad hoc networks. In *Mascots*, 2004.
[17] I. Habib. Virtualization with kvm. *Linux Journal*, 2008.
[18] G. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.
[19] N. Horman. Understanding and programming with netlink sockets. `http://www.smacked.org/docs/netlink.pdf`, 2004.
[20] Y. Hu, A. Perrig, and D. Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. *WN*, 2005.
[21] D. Johnson and D. Maltz. Dynamic source routing in ad hoc wireless networks. *Mobile computing*, pages 153–181, 1996.
[22] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. Finding protocol manipulation attacks. *Sigcomm CCR*, 2011.
[23] H. Lee, C. Killian, C. Nita-Rotaru, and J. Seibert. A Platform for Finding Attacks in Unmodified Implementations of Intrusion Tolerant Systems. *Poster at OSDI*, 2012.
[24] H. Lee, J. Seibert, C. Killian, and C. Nita-Rotaru. Gatling: Automatic attack discovery in large-scale distributed systems. *NDSS*, 2012.
[25] Q. Li, M. Zhao, J. Walker, Y.-C. Hu, A. Perrig, and W. Trappe. SEAR: A secure efficient ad hoc on demand routing protocol for wireless networks. *Security Comm. Networks*, 2(4):325–340, 2009.
[26] P. Mahadevan, A. Rodriguez, D. Becker, and A. Vahdat. Mobinet: a scalable emulation infrastructure for ad hoc and wireless networks. *Sigmobile CCR*, 2006.
[27] P. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM ToCS*, 2011.
[28] K. McMillan. Symbolic model checking: an approach to the state explosion problem. Technical report, 1992.
[29] M. Musuvathi, D. Engler, et al. Model checking large network protocol implementations. In *NSDI*, 2004.
[30] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: Pragmatic approach to model checking real code. *Sigops*, 2002.
[31] S. Paris, C. Nita-Rotaru, F. Martignon, and A. Capone. Efw: A cross-layer metric for reliable routing in wireless mesh networks with selfish participants. In *Infocom*, 2011.
[32] C. Perkins and P. Bhagwat. Highly dynamic DSDV for mobile computers. *ACM Sigcomm CCR*, 1994.
[33] C. E. Perkins and E. M. Royer. Ad-hoc On-Demand Distance Vector Routing. In *IEEE Mcsa*, 1997.
[34] S. Radhakrishnan, G. Racherla, C. Sekharan, N. Rao, and S. Batsell. Dst-a routing protocol for ad hoc networks using distributed spanning trees. In *IEEE WCNC*, 1999.
[35] K. Sanzgiri, B. Dahill, B. Levine, C. Shields, and E. Belding-Royer. A secure routing protocol for ad hoc networks. In *IEEE ICNP*, 2002.
[36] K. Sanzgiri, D. LaFlamme, B. Dahill, B. Levine, C. Shields, and E. Belding-Royer. Authenticated routing for ad hoc networks. *IEEE JSAC*, 2005.
[37] M. Stanojevic, R. Mahajan, T. Millstein, and M. Musuvathi. Can you fool me? towards automatically checking protocol gullibility. In *HotNets*, 2008.
[38] M. G. Zapata and N. Asokan. Securing ad hoc routing protocols. In *ACM WiSE*, 2002.
[39] X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: a library for parallel simulation of large wireless networks. *Sigsim*, 1998.