Automated Adversarial Testing of Unmodified Wireless Routing Implementations*

Endadul Hoque, *Member, IEEE,* Hyojeong Lee, *Member, ACM,* Rahul Potharaju, *Member, IEEE,* Charles Killian, *Member, ACM,* and Cristina Nita-Rotaru, *Senior Member, IEEE, Member, ACM*

***Abstract*—Numerous routing protocols have been designed and subjected to model checking and simulations. However, model checking the design or testing the simulator-based prototype of a protocol does not guarantee that the implementation is free of bugs and vulnerabilities. Testing implementations beyond their basic functionality (also known as *adversarial testing*) can increase protocol robustness. We focus on automated adversarial testing of real-world implementations of wireless routing protocols. In our previous work we created Turret, a platform that uses a network emulator and virtualization to test unmodified binaries of general distributed systems. Based on Turret, we create Turret-W designed specifically for wireless routing protocols. Turret-W includes new functionalities such as differentiating routing messages from data messages to enable evaluation of attacks on the control plane and the data plane separately, support for several additional protocols (e.g., those that use homogeneous/heterogenous packet formats, those that run on geographic forwarding (not just IP), those that operate at the data link layer instead of the network layer), support for several additional attacks (e.g., replay attacks) and for establishment of adversarial side-channels that allow for collusion. Turret-W can test not only general routing attacks, but also wireless specific attacks such as wormhole. Using Turret-W on publicly available implementations of five representative routing protocols, we (re-)discovered 37 attacks and 3 bugs. All these bugs and 5 of the total attacks were not previously reported to the best of our knowledge.**

***Index Terms*—Automatic testing, routing protocols, security, wireless communication**

I. 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 [2], and OLSR [3], reactive protocols such as AODV [4] and DSR [5], and hybrid protocols such as DST [6]. Additionally, there have also been efforts to improve the performance of the routing protocols by operating at the data link layer instead of the network layer, a representative example being the BATMAN [7] protocol. Given the increased threats that exist in wireless networks, several secure routing protocols have been designed. Examples include SAODV [8], ODSBR [9], ARAN [10], and Ariadne [11]. Many of the protocols mentioned above, such as AODV, ARAN, OLSR, DSDV, and BATMAN, were implemented and are available from public repositories [12]–[16].

Given the importance of routing as a fundamental component of wireless networks, many protocols have been subjected to model checking the design [17] and to testing the simulator-based implementation [18], [19]. For example, several model checking tools [17] were used to verify wireless routing protocols, and several simulators [18], [19] were used to demonstrate and test wireless routing protocols [2]–[6], [20]. While model checking helps to verify the validity of the design, it does not provide a guarantee that the real-world implementation is free of bugs and vulnerabilities, since implementations contain optimizations not captured by the model, sometimes diverge from the design, and often introduce new bugs. In addition, while simulators provide easier and simpler ways to describe a protocol, they sacrifice some aspects of realism such as the interaction of the protocol with the operating system components.

Fig. 1 shows the popularity of some wireless protocols in the academic community (obtained from Google Scholar) — it is evident that hundreds of researchers use the publicly available implementations for performance comparison across protocols [21]–[23], or to investigate properties of the network stack such as performance of TCP in multihop ad hoc networks [21], [24]. Thus, it is important to ensure that these implementations are robust and do not include faults and security vulnerabilities that may lead them to enter an unsafe state or exhibit degraded performance.

In our previous work, Gatling [25], we showed the importance of performing adversarial testing for message-passing distributed systems. By testing systems implementations beyond just basic functionality (i.e. examining edge cases, boundary conditions, and ultimately conducting destructive testing), we discovered vulnerabilities, many of which were not captured by model checking the design or by simulator-based testing. However, Gatling requires the target protocol to be implemented in the MACE language [26], whereas Max [27] focuses on two-party network protocols to find attacks that can manipulate the victim's execution control flow by relying on the user specified information about a known vulnerability of the implementation to limit the search space and thereby catering itself as more suitable for corner cases.

In this paper, we focus on adversarial testing of implementations of wireless routing protocols. We consider attacks and failures that are created through manipulation of protocol

E. Hoque and C. Nita-Rotaru are with the Department of Computer Science at Northeastern University, Boston, MA and Purdue University, West Lafayette, IN. (email: mhoque@purdue.edu; c.nitarotaru@neu.edu).

H. Lee is with Google, Inc. (email: hjlee0215@gmail.com).

R. Potharaju is with Microsoft (email: rahul986@gmail.com).

C. Killian is with Google, Inc. and Purdue University, West Lafayette, IN. (email: ckillian@cs.purdue.edu).

**A preliminary version of this paper was presented at WiSec 2013 [1].*

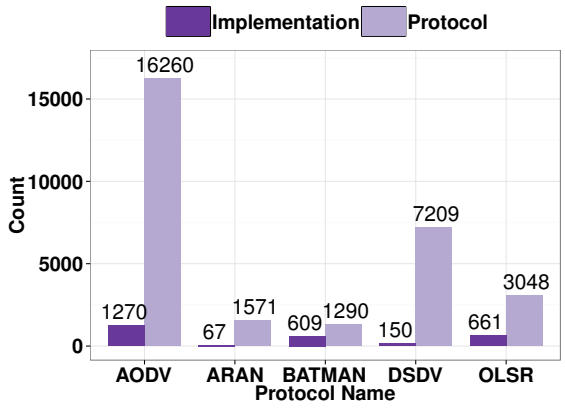


Fig. 1. Comparison of the routing protocols based on popularity (computed by searching on Google Scholar). *Protocol* counts indicate the total number of citations to the original research paper; *Implementation* counts indicate citations to the implementations and the URL of the software.

messages and are specific to wireless routing protocols, having a global impact on the protocol performance. We build on our previous work in automated adversarial testing for distributed systems by leveraging the design of Turret [28] to create an adversarial testing platform for wireless routing protocols. Turret uses a network emulator to create reproducible network conditions and virtualization to run unmodified binaries of systems' implementations. The platform requires the user to provide a description of the protocol messages and corresponding performance metrics. Turret's design is a good starting point for a cost-effective wireless testing environment because it allows a binary to run in its native operating system while limiting the impact of noise and interference on the performance of the system. Our contributions in this work are:

- We present Turret-W, a platform for adversarial testing of wireless routing protocols. Turret-W leverages the design of Turret and includes new functionalities such as the ability to differentiate routing messages from data messages, support for protocols that use homogeneous or heterogenous packet formats, support for protocols that run on geographic forwarding (not only IP), support for protocols that operate at the data link layer instead of the network layer, support for replay attacks, and ability to establish side-channels between malicious nodes. As a result, Turret-W can test not only general attacks against routing, but also wireless specific attacks such as blackhole and wormhole attacks. Our approach is cost effective in comparison with the hardware and manpower costs required by the approach in [23]. In addition, our approach does not pose any restriction on the implementation language like Gatling [25], nor relies on a priori knowledge of any vulnerability like Max [27].
- We demonstrate attack discovery with Turret-W using detailed case studies on five representative wireless routing protocols: a reactive protocol (AODV), a secure reactive protocol (ARAN), and three proactive protocols (OLSR, DSDV, and BATMAN), whose implementations we obtained from public repositories. We found 1 new and 7 known attacks in AODV, 6 known attacks in ARAN, 5 known attacks in OLSR, 4 new and 7 known attacks in DSDV, and 7 known attacks in BATMAN, for a total of 37 attacks. While most of attacks we found are protocol level attacks, one attack in AODV and 4 attacks in DSDV were solely implementation level attacks, and such attacks could have been discovered only by testing the actual implementations under adversarial environments.
- We show that Turret-W also can find bugs, as it provides a testing environment that is realistic and controllable. Unlike attacks, bugs cause performance degradation in benign executions. We discovered 3 bugs in total, 2 in AODV and 1 in ARAN. The bugs in AODV were due to a subtle interplay between AODV code and the operating system kernel.

The rest of the paper is organized as follows. §II provides an overview of the platform we use in this paper, §III describes our methodology, while §IV–§VIII present our five case studies on AODV, ARAN, OLSR, DSDV, and BATMAN, respectively. §IX describes related work and §X summarizes the paper.

II. Platform Overview

Our goal is to test wireless routing implementations, where the network conditions can be reproducible and also isolated from outside world interference. In our previous work [28] we created Turret, a platform for adversarial testing of message passing distributed systems. The design of Turret makes it an appealing choice for testing wireless network protocols because the emulation of the network ensures reproducible performance and limits the noise and interference, while the virtualized approach allows binaries to run in their native environments. However, Turret cannot be directly applied to wireless networks or routing protocols. Below, we first give an overview of Turret, the platform that we built on, and then explain what functionalities we added to support wireless routing protocols. We refer to Turret with our extension as Turret-W.

A. Overview of Turret

Turret is a platform for performance-related attack discovery in unmodified distributed system binaries. Turret uses virtualization (i.e. KVM [29]) to run arbitrary operating systems and applications, and network emulation (i.e. NS-3 [30]) to connect these virtualized hosts in a realistic network setting. Turret requires a description of the message formats that the system relies on, and a set of metrics that capture the performance of the system.

A controller bootstraps the system by starting NS-3 and running application binaries inside the virtual machines. Each of these virtual machines (VMs) acts as an individual node of the distributed system. The VMs communicate with each other with the help of the NS-3 emulator. Specifically, each VM is mapped to a node inside NS3, called a *shadow node*, through a *Tap Bridge* connection (available in NS-3), which connects the inputs and outputs of an NS-3 network device to the inputs and outputs of the VM's network interface (i.e., the corresponding TAP device of the VM) as if the NS-3 network device is a local device to the VM. The controller lets each shadow node know if it will act as a benign node or as a malicious node.

TABLE I
MESSAGE DELIVERY ACTIONS IN TURRET

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 II
MESSAGE LYING 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

A shadow node instructed to act as malicious will activate the *malicious proxy*, a component implemented by Turret on top of the Tap Bridge, to intercept messages generated by the application running inside the virtual machine and modify them according to an *attack strategy*. An attack strategy may consist of two types of malicious actions: *Message Delivery Actions* that affect when and where a message is delivered (see Table I) and *Message Lying Actions* that affect the contents of a message (see Table II). In the case of message lying actions, different fields inside a message can be automatically modified based on the selected attack strategy and the user-provided message formats.

B. Limitations of Turret for Wireless Routing

Turret cannot be directly applied to wireless networks or routing protocols because of several limitations.

Distinguishing between control plane and data plane: While Turret can inject attacks and faults into any message-oriented protocol, it does not differentiate data messages from routing messages. In case of routing, many attacks on the data plane including degradation in the application performance can be amplified if the routing mechanism is disrupted. Thus, a platform intended for routing needs to control independently both the control (routing) plane and the data plane so that it can inject fine-grained attacks based on the type of the control plane messages and coarse-grained attacks based on the service type of the data plane messages. For wireless networks, the separation is also needed to support basic attacks such as *blackhole* in which an attacker will drop all data messages but participate in the routing algorithm correctly.

Parsing homogeneous and heterogeneous packets: Turret expects an intercepted packet to contain only one message pertaining to the target protocol. Whereas routing protocols are typically designed to follow either *homogeneous packet format* (i.e. the routing protocol packs one type of routing message(s) into a single datagram) or *heterogeneous packet format* (i.e. the routing protocol packs different types of routing messages into a single datagram). In both cases, the length of the packet can be fixed or variable. Routing protocols designed for wireless networks generally adopt either packet formats, as communication is expensive in wireless networks.

Supporting non-IP packets: Turret assumes that the target protocol runs on top of Internet Protocol (IP) at the network layer. Thus, the malicious proxy processes each intercepted packet as an IP packet. However, not all existing wireless routing protocols use IP as the packet forwarding protocol at the network layer. For example, some protocols use geographic forwarding [31], [32] where packets are forwarded based on physical proximity. Others such as BATMAN [7] or HWMP [33] operate at layer 2 (data link layer), instead of layer 3 (network layer), use MAC addresses for routing instead of IP addresses and transport routing information encapsulated into raw Ethernet frames. Therefore, it is important to support both non-IP and layer 2 routing packets to enable adversarial testing of such protocols.

Replaying packets: Turret does not provide the functionality to replay packets. Replaying packets is particularly interesting in case of wireless networks since it is a very low cost attack that can easily be launched. Note that packet replaying is different from packet duplication. In a replay attack, an attacker records another node's valid packets and resends them (without modification) later to other benign nodes via legitimate channels only if the packets contain the target control message(s). 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 disrupt the routing plane.

Establishing wormhole tunnels: Turret does not support colluding attacks. However, an attack specific to wireless networks that requires coordination between two attackers and is 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. As wormhole attacks are feasible (basic attack requires only two colluding nodes), it is important to be able to test the impact of wormhole attacks on the routing protocol.

C. Turret-W Description

We modified Turret to address the above limitations. The new platform, Turret-W, is shown in Fig. 2. The controller component coordinates the testing. It generates a topology file for the network emulator using a configuration file provided by the user. The configuration file specifies parameters such as the network topology, number of nodes, and number of malicious nodes. The controller then starts the virtual machines and binds each of them to the underlying network emulation layer. It then loads the routing service at the routing layer and instantiates the application 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 used to estimate the performance of the application running on top of the routing protocol.

Wireless network emulation: Like in Turret, the virtual machines operate on top of a network emulation layer provided by NS-3[1]. We configure NS-3 to emulate WiFi links. We leverage the Tap Bridge connection (available in NS-3) to

[1]Note that Emulab [34], MobiNet [35], Orbit [36] could also conceptually replace NS3. Emulab with fixed wireless provides more realism. However, the approach provides less reproducible results because of unwanted interference on the wireless channel and requires a separate implementation of the malicious version of the target routing protocol for each malicious node.

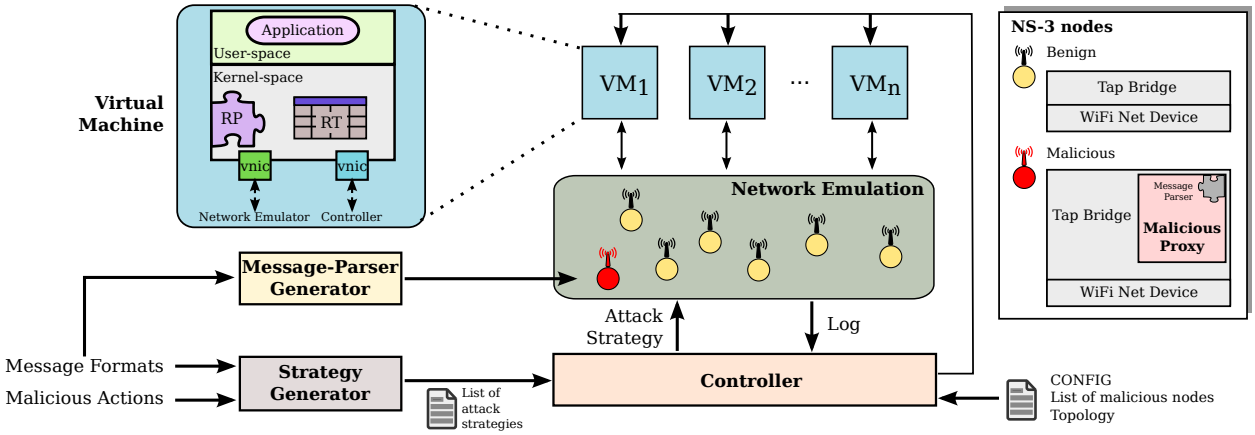


Fig. 2. Turret-W platform (RP:Routing Protocol, RT:Routing Table and VNIC:Virtual Network Interface Card)

connect a VM with its corresponding shadow node so that it enables a NS-3 net device to appear as a local device inside the VM thereby allowing the VM to use this local net device for WiFi transmission. The network emulation layer creates a virtual multi-hop wireless environment to transmit packets from a source to a destination virtual machine.

Attacks specific to wireless routing: We modified the Turret's malicious proxy (implemented on top of the Tap Bridge) to differentiate between messages originating from the routing layer and the application layer based on the port number. Differentiating data messages from routing messages allows Turret-W to implement a blackhole attack wherein a malicious node acts benign at the routing layer but selectively/entirely drops messages originating from the application layer.

We also provide support for a wormhole attack as follows: a wormhole tunnel is implemented as part of the malicious proxy inside NS-3 connecting any two colluding adversaries (precisely, shadow nodes). However, the routing code running in the virtual machines are oblivious of this tunnel, which introduces a new challenge to deal with. If we just forward data messages between the end nodes forming the wormhole, one side effect is that the routing will believe there is no direct link between the two end points of the wormhole. Therefore, to convince the routing services of the end nodes, forming the wormhole tunnel, that they are direct neighbors of each other, we allow these end nodes to exchange their own beacon messages (e.g., HELLO) over the wormhole tunnel. At the same time, the beacon messages forwarded between the ends of the wormhole should be restricted only to those generated by the end nodes that form the wormhole and not their neighbors since that will results in incorrect updates of routing tables. All other routing protocol messages are forwarded by the colluding nodes over the wormhole tunnel 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 Tables I, II, and III.

Homogeneous and heterogenous packets: To inject a malicious action, the malicious proxy needs to be able to parse messages in order to act on different message types and to lie on a particular field of a message. The message-parser reads a message format description and outputs necessary source code that feeds into the malicious proxy. This source code contains a set of API calls (e.g., getMessageType(), getMessageSize() etc.) that expose properties of the message to the malicious proxy. An example message format description (a route request for AODV) is given below:

```
AodvRreq {
  uint8_t type = 1;
  uint32_t dest_addr;
  uint32_t dest_seqno;
  uint32_t orig_addr;
  uint32_t orig_seqno;
  ... }
```

Routing protocols can follow either homogeneous packet format or heterogeneous packet format. For instance, AODV sends a route request message in a single UDP packet and thus, can be said to follow the homogeneous packet format. In contrast, OLSR allows individual messages be piggybacked and transmitted together in one transmission such as a topology control message and a HELLO message can be sent together in a single UDP packet. We modified the message-parser generator so that it can handle both homogeneous and heterogeneous packet formats and thus, enable testing of a wider variety of routing protocols.

Packet forwarding protocols: Typically routing protocol implementations use Internet Protocol (IP) as the packet forwarding protocol at the network layer. However, developers are free to choose other packet forwarding protocols more suitable for the target network such as geographic forwarding for wireless ad hoc networks. The DSDV implementation [37] for the Click Modular router [38] is using such a protocol. Instead of IP, it is built on top of the Grid service [39] that is based on geographic forwarding. We modify the malicious proxy so that it handles routing messages packed into either IP or non-IP packets, and thus, we enabled the testing of routing protocols that are built on top of non-IP protocols.

Routing at layer 2: Traditionally routing protocols operate at layer 3 (the network layer) on top of IP (or some other packet forwarding protocols). However, several recently developed routing protocols (e.g., BATMAN) operate at layer 2 (the data link layer) where the nodes are attached to a unique Ethernet broadcast domain and are agnostic to the network topology. Moreover, routing in such protocols relies on MAC addresses instead of IP addresses. To enable adversarial testing of routing protocols like BATMAN, our malicious proxy supports injecting malicious actions into routing messages even when they are encapsulated and forwarded as raw Ethernet frames.

TABLE III
MALICIOUS ACTIONS ADDED BY TURRET-W

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 algorithm correctly	None
Wormhole	Creates a wormhole between two colluding nodes and tunnels packets between each other	None
Wormhole with blackhole	Creates a wormhole between two colluding nodes and tunnels routing packets between each other, but drops all data packets	None

Attack strategy generation: The strategy generator is responsible for generating a list of attack strategies that the target protocol should be tested against. For example, consider the following strategies in case of AODV where the malicious proxy is being instructed to duplicate each route request (AodvRreq) message 50 times and drop all the route error (AodvRerr) messages (i.e. 100%):

```
DUP AodvRreq 50
DROP AodvRerr 100
```

Given the message format description of the protocol under test, these attack strategies are generated based on the malicious actions listed in Tables I and II along with a value that decides the severity of that action. This attack strategy generation is inspired from prior work [25], [27], [40]. To support the additional wireless specific attacks listed in Table III, we extended Turret's basic set of malicious actions with replay, blackhole, and wormhole attacks.

Note that Turret-W treats the protocol binary as a black-box and requires no additional information on the protocol, e.g., source code. While such characteristics make Turret-W to be applicable to a wide range of routing implementations, they may not always lead Turret-W to finding sophisticated attacks that can manifest deep in the execution. Finding such attacks often requires more information about the protocol like the protocol state-machine (required by SNAKE [41]) or the instrumented source code (required by MAX [27]).

Support for multiple interfaces: Though Turret-W currently supports routing protocols that rely on a single network interface out-of-box, the platform can easily be extended to support routing protocols that leverage multiple network interfaces [42], [43]. In our current setup, each VM is equipped with only two network interfaces — one dedicated for the target routing protocol and another for other purposes (e.g., controlling the VM). Therefore, to enable testing of routing protocols that leverage multiple interfaces, we could equip the VMs with the necessary number of interfaces and configure the network emulator to detect these interfaces.

III. METHODOLOGY

We demonstrate our platform on real-world implementations of five representative wireless routing protocols: AODV [4], ARAN [10], OLSR [3], DSDV [2], and BATMAN [7]. AODV is a well-known reactive (routes are determined on-demand) routing protocol whereas ARAN is not only reactive but also a secure routing protocol. On the other hand, both OLSR, DSDV, and BATMAN are proactive (routes are determined in advance) routing protocols. For AODV, ARAN, OLSR, and BATMAN we obtained the implementations from their public repositories [12]–[14], [16], while for DSDV, we obtained the implementation available in the Click modular router source [15]. Note that the DSDV implementation runs on geographic forwarding and the BATMAN implementation operates at layer-2 (the data link layer). In addition, the implementations of OLSR and BATMAN are being developed as part of various Linux distributions (e.g., Ubuntu). Next, we describe the attacker model, our experimental setup and the selection of system parameters.

A. Attacker Model

We focus on performance attacks mounted by malicious participants to disrupt the routing service thereby impairing the protocol performance, which is expressed by a *performance metric* that is when evaluated gives an indication of the progress the protocol has made towards completing its goals. To find such attacks, we measure the protocol performance, using the given performance metric, during each execution of the protocol in the presence of malicious participants in the network. The achieved performance is compared against a baseline performance obtained from an execution where all nodes are benign. We define an attack as follows:

Definition 1 - Performance Attack*: When the performance difference between a malicious execution and a benign execution is greater than a threshold, δ, we say that the attack strategy has resulted in a successful attack.*

Here, δ is a system parameter that depends on the protocol under test.

By directly testing real implementations running in their target operating systems, our platform captures the intricate interactions between the protocol being tested and the operating system components. In addition, the isolation and the reproducibility offered by the emulated and virtualization-based environment help us discover bugs that impair the performance of the protocol even in a benign environment. Such bugs cannot be found in a simulation environment. We define a bug as follows:

Definition 2 - Performance Bug*: A performance bug is an implementation-level error that limits the practical utility of the protocol in a benign execution by causing 100% loss of application packets sent by the source.*

B. 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 the experiments, we use 12 VMs, each allocated 128 MB RAM. For AODV, we use Debian 6.0.5 with Linux Kernel 2.6.32 as the guest OS. One of the advantages of our platform is that it allows us to execute binaries to run on their target operating systems. For instance, since ARAN requires an older kernel, we use Fedora Core 1 with Linux kernel 2.4.22 as the guest OS.

Our emulated network is a multihop wireless adhoc network. For the 802.11 MAC layer, we use 802.11a with a bit rate of 6 Mbps and a propagation loss model (called RangePropagationLossModel, available in NS-3) with a range of 100 meters for each link. We perform our experiments using a static grid topology. As an application on the VMs, we run *iperf* [44], a network benchmarking tool. In all the experiments, the performance of the application we report is averaged over ten runs.

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 implementation, we use 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 [9], [45].

As a performance metric, we use *packet delivery ratio* (PDR), i.e., a ratio of the total number of packets (in our case, application packets) received by the destination to the total number of packets sent by the source. PDR is easy to measure irrespective of the underlying routing protocol as it can be computed from the results produced by the application (i.e. iperf). Moreover, this metric does not require any instrumentation of the routing protocol implementation, which supports our goal of testing unmodified routing implementations. For each protocol, we capture the PDR achieved in each malicious execution and compare it with the baseline PDR. Given that we look for attacks that significantly degrade the performance, we argue that the measured baseline PDR can be used as a ground truth since it is always closed to the maximum (i.e., 100%) as per our experimental observation (see § IV- VIII).

We select malicious node(s) randomly and inject malicious strategies during the entire experiment. We vary the total number of adversaries from 1 to 4 (out of the total 12 nodes) exhibiting a homogeneous behavior, i.e., we inject the same attack strategy to each malicious node. For every attack strategy applied to the routing messages, a malicious node drops application packets with a probability of p (a system parameter) to affect the performance of the application.

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 protocol but drops 100% of application packets. The effect of a wormhole is noticeable in terms of application performance when combined with a blackhole attack. Remember that except for blackhole and/or wormhole attacks, we use the packet dropping probability p to drop application packets in all other malicious executions.

The threshold δ, a system parameter, is dependent on the protocol under test. The user can specify the threshold indicating the amount of performance loss he is willing to tolerate. Alternatively, it can be determined from ground truth by recording the observed performances for different attack strategies and select the threshold value that will detect the attack manifested by the weakest adversary from the set of the known attacks where a higher threshold means a more aggressive attacker. We relied on the second approach. We consider blackhole with one attacker as the weakest adversary where the adversary drops all data messages but participates benignly in the routing protocol. Moreover, we know all the protocols we are testing are susceptible to blackhole attacks. Hence, we decide to choose 0.2 (i.e., 20%) as our threshold so that our tool can detect the blackhole attack. Intuitively, any successful attack strategy manifested by a relatively stronger adversary (attacks against both the routing and the data messages) worsens the performance. Therefore, the chosen δ would also be able to detect such attack strategies.

Overhead of Turret-W: Routing protocols usually use timeouts to prevent the use of stale information or provide reliability of transmission. When these timeouts expire, routing protocols take necessary measures such as removing stale entries from routing tables, restarting new route discovery, or entering the recovery state. Turret-W can cause two different types of delays that will not be observed in real environment. First, it can cause a processing delay when the network flow is heavier than the network emulator capacity. Second, a malicious proxy can add delays while injecting malicious actions. The first type of delay is due to the nature of emulation based testing and can be prevented by over-provisioning. However, the impact of the second type of delay needs to be measured. To evaluate the amount of delay introduced by the malicious proxy, we performed experiments with AODV and OLSR protocols for the malicious attacks listed in Table IV. We observed that the delay is in the order of tens of μsec with a median of 40 μsec. Whereas the route expiration timeout used in AODV and OLSR are 5 sec and 6 sec, respectively. This result demonstrates that the computation of the malicious proxy of Turret-W does not have any significant impact on the routing protocols due to the low overhead.

Scalability of Turret-W: The scalability of Turret-W depends on (a) the scalability provided by the underlying emulator, and (b) the scalability of the routing protocol under test. Turret-W leverages the emulation environment of NS-3 and hence is subject to its limitations such as not being able to support large network sizes in the emulation due to the overhead related to the management of the large number of threads in the NS-3 process [46]. As NS-3 is one of the most widely used network emulators and the performance of network emulation is not within the scope of our work, we choose a reasonable size of network consisting of 12 nodes and focus on networks that can still operate correctly under a reasonable number of malicious nodes (up to 30% of the total nodes).

IV. Case Study 1: AODV

We now describe how we used Turret-W to test AODV [4]. All discovered attacks and bugs are shown in Table IV.

A. Protocol Description

AODV establishes a path on-demand. Specifically, when a source desires to send a message to a destination to which it does not have a valid route, it starts a *route discovery* process by broadcasting a route request (RREQ) message to its neighbors. Each node then forwards the first received RREQ by re-broadcasting it to its neighbors. This process continues until the RREQ reaches the destination or an intermediate node that has a valid route to the destination. In addition to forwarding the RREQ, each intermediate node records in its routing table (*i.e.*, *precursor list*) the address of the neighbor from which it receives the first RREQ, forming a reverse path. Once the RREQ reaches the destination node or an intermediate node with a valid route, the node responds to the RREQ by unicasting a route response (RREP) message to its precursor neighbor, *i.e.*, its neighbor on the reverse path, which in turn relays the RREP via precursor nodes back to the

source node. From then on, the source node keeps unicasting the data to the next hop neighbor as long as the route is valid.

A node maintains connectivity with its neighbors by periodically broadcasting beacon messages (HELLO). Whenever the next hop becomes unreachable, the upstream node of the broken link propagates a route error (RERR) message to each of its upstream neighbors. Following the reverse path, the RERR finally reaches each source node that contains the broken link on the route to its destination. A source then reinitiates the route discovery if a route to the destination is still desired.

Implementation used: We use AODV-UU-0.9.6 implementation publicly available from [12], which is RFC 3561 [47] compliant. The AODV-UU consists of two components — a loadable kernel module (`kaodv`) and a user space daemon process (`aodvd`). The kernel module intercepts and handles network packets by registering hooks (callbacks) with the Linux kernel's network stack. To register such hooks, `kaodv` uses the *Netfilter* framework [48]. The daemon (`aodvd`) uses netlink socket to communicate with `kaodv` and `NETLINK_ROUTE` protocol to communicate with the kernel routing table. We configure the protocol using the default values presented in [12].

B. Discovered Bugs

During the benign testing 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 are multiple hops away from each other, we observed that packets were not being delivered in the benign case. By design, whenever an application sends a packet for a destination to which the route is either invalid or unavailable, `kaodv` should 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. Our investigation revealed that in the AODV-UU implementation, 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 performance where the first packet, *i.e.*, SYN packet is crucial to establish the connection. We also 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 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. Thus, `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 [49]. 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`.

C. Discovered Attacks

Attack caused crashing. We discovered an implementation attack that can cause all neighbors of a malicious node to crash. When a malicious proxy modifies an RREQ message to be an RREP by changing the type of the RREQ message, a recipient processes this altered RREQ message as an RREP message. The base RREP message (i.e., 20 bytes) is smaller in length than a base RREQ message (i.e., 24 bytes) [47]. Therefore, a recipient of the malformed RREQ message processes the message as if it were an RREP with extensions [47], and this causes AODV-UU of the receiver to crash with a segmentation fault. However, the root cause is an interger overflow vulnerability in the AODV-UU code. We show the related code snippet below:

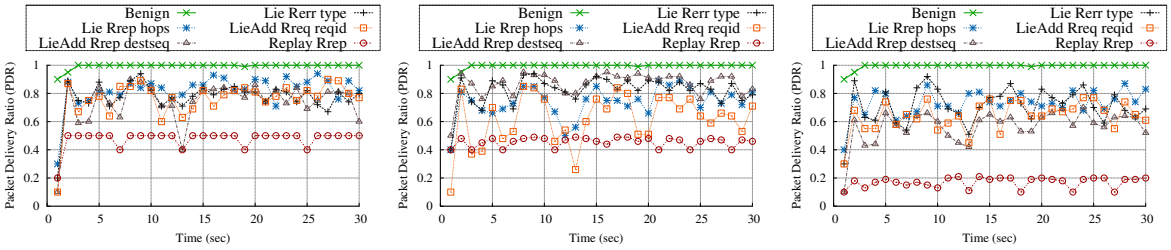
```
1.void NS_CLASS rrep_process(..., int rreplen, ...){
2.    unsigned int extlen = 0;
3.    AodvExtension *ext = rrep + RREP_SIZE;
      ...
4.    while ((rreplen - extlen) > RREP_SIZE) {
        // RREP_SIZE is 20
        ...
5.      /* read ext length from packet */
6.      extlen += EXT_HDR_SIZE + ext->length;
        // EXT_HDR_SIZE is 2
7.      ext = ext + EXT_HDR_SIZE + ext->length;
8.    }
      ...
9.  }
```

`extlen` is defined as an unsigned integer (line 2) and there is no checking if the extension length matches the actual message size. In this case, the received buffer length (`rreplen`) is 24 bytes. Therefore, when the RREQ's originator seq number field value becomes 21 or bigger, this code will assume that the message has two extensions, one with 0 length and the other with length 21 or larger. At line 6, it will first increase `extlen` to be 2, which is the header size, then at the second iteration, it will add 2+21, and thus, `extlen` becomes 25. This results in an integer overflow on the left hand expression of the "while" condition at line 4, and therefore, the loop continues iterating. Later, the code crashes with a segmentation fault. This vulnerability can be fixed by enforcing careful type safety and boundary checking.

Attacks caused by malicious actions. We rediscovered several attacks on AODV-UU based on message delivery and lying actions that decrease the PDR below the accepted threshold. By design, AODV is known to be susceptible to these attacks [8], [10]. In case of our benign experiments, we observe a 98% PDR. Fig. 3(a)-3(c)[2] show the temporal impact of the attacks on PDR as a function of the number of adversaries in the network. The impact of an attack increases as more nodes become malicious in the network.

Replay RREP. By replaying an RREP message received from a node, an adversary can fool its benign neighbors to

[2]Due to space limitations, we omit the figure for 1 adversary from Fig. 3



(a) Two adversaries (b) Three adversaries (c) Four adversaries

Fig. 3. Packet delivery ratio for AODV-UU for the discovered attacks against routing messages

believe that the originator is their one-hop neighbor. The benign neighbors that are at least two hops away from the actual originator believe the adversary is the originator node as they never receive RREP messages directly from the originator. 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 as low as 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 adds a positive value with the destination sequence number of an RREP message, and this causes the recipient to select the route through the malicious node. In the case of 4 adversaries the PDR drops to 56%.

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 as low as 62% as the number of adversaries increases.

Lie RERR type and *Lie RREP hops*. Modifying the type of an RERR to RREQ causes the recipient to discard the packet. We find that adversaries can reduce the performance to 71% by performing this attack. Similarly, when a malicious node sets the hop count of an RREP to 0, the recipient selects the route through the malicious node as the recipient thinks that it can reach the destination by 1 (=0+1) hop. We observe that this attack causes the PDR to drop up to 73%.

Blackhole/wormhole attacks. We first tested AODV-UU against blackhole attackers (malicious nodes that drop all the data packets). We then introduce an additional blackhole node that colludes with the other blackhole node via a private channel to perform a wormhole attack. The PDR drops to 50% with the increase in blackhole nodes, whereas the PDR drops to 40% in case of the wormhole attack.

V. Case Study 2: ARAN

We now describe how we used Turret-W to test the implementation of ARAN presented in [50]. We summarize all discovered attacks and bugs in Table IV.

A. Protocol Description

ARAN [10], [50] is a secure reactive wireless routing protocol. ARAN introduces authentication, message integrity and non-repudiation by utilizing digital signatures on messages. Each node receives a certificate from a trusted certification authority (CA). The protocol consists of a route discovery process utilizing three types of routing messages: route discovery (RDP), route reply (REP), and route error (ERR). In essence, the route discovery process of ARAN is similar to that of AODV. In addition, ARAN guarantees end-to-end authentication. The routing messages are digitally authenticated at every hop, which ensures that only authorized nodes participate at each hop between the source and the destination.

Implementation used: We rely on the implementation *arand*-0.3.2 (referred below as ARAND), publicly available from [14]. This user space routing daemon built for Linux kernel 2.4 relies on the Ad hoc Support Library (ASL) [51] that provides an interface to the kernel functionalities required by any on-demand ad hoc routing protocol. ASL takes care of adding/deleting routes in the kernel routing table and notifying ARAND to initiate a route discovery for a destination in case of an unavailable route. The ARAND daemon also utilizes the functionality provided by the `route_check` kernel module of ASL to delete stale routes. For the cryptographic functionalities, it uses OpenSSL [52]. We use the default values for parameters as used in [14].

B. Discovered Bug

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 attempts to forward the response directly to the source node (i.e., the originator of the route discovery) instead of the correct next hop node that is on the reverse route to the source. If the intermediate node is more than one hop away from the source node, this response message cannot be delivered to the source, and thus, the route discovery fails. We fix this bug by letting the intermediate node use the correct next hop address to forward the route response. This bug is due to an implementation mistake that exists inside the `aran_processREP()` function defined in `aran.c` and manifests in topologies having nodes that are at least 3 hops away from each other.

C. Discovered Attacks

Attacks caused by message forwarding actions. We rediscovered several attacks on ARAND based on malicious delivery that have a significant impact on the performance. By design, ARAN is known to be susceptible to these attacks [9], [53]. We observe a 99% PDR when no attacks take place. We

then measure the changes in the PDR achieved by ARAND as a function of the number of adversaries. Due to space limitations, we omit these graphs.

Divert REP, *Drop ERR* and *Delay REP*: 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. Diverting REP messages disrupts the completion of route discovery whereas dropping ERR messages keeps the source unaware of the broken link and thus, prevents the source from re-initiating a route discovery for the destination. Four malicious nodes can drop the PDR to below 30% by diverting REP messages and to 40% by dropping ERR messages. On the other hand, delaying a REP message at an intermediate malicious node can reduce the PDR, but the impact is less significant as compared to diverting REP messages.

Drop RDP. An intermediate malicious node can drop a route discovery (RDP) message instead of re-broadcasting. 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).

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%.

VI. Case Study 3: OLSR

We now describe how we used Turret-W to test OLSR [3]. All discovered attacks are shown in Table IV.

A. Protocol Description

OLSR [3] is a proactive routing protocol based on the traditional link-state algorithm where each node maintains topology information about the network by periodically exchanging link-state messages. OLSR minimizes the size of each control message and the number of rebroadcasting nodes during each route update by employing a multipoint relaying strategy. During every topology update, each node in the network selects a set of neighboring nodes, called *multipoint relays*, to retransmit its packets. To select the multipoint relays, each node periodically broadcasts a list of its one hop neighbors using HELLO messages. From the list of nodes in the HELLO messages, each node selects a subset of its one hop neighbors, which cover all of its two hop neighbors. Each node, then, disseminates information about the subset, i.e., the set of multipoint relays, using *topology control* (TC) messages that are retransmitted only by the multipoint relays of the node. Other nodes receiving these TC messages process them but do not retransmit. Each node eventually determines an optimal route (e.g., with minimum hops) to every known destination using the topology information and updates its routing table. During data transmission, this routing table is leveraged to determine route to a destination.

Implementation used: We use olsrd-0.6.3 (referred below as OLSRD) publicly available from [13], which is RFC 3626 [60] complaint. This implementation is a routing daemon that employs the `ioctl()` system call to communicate with the kernel and utilizes the `NETLINK_ROUTE` protocol to manipulate the kernel routing table. Unlike the above reactive protocols, it does not have any kernel module that intercepts the network packets from the network subsystem. The daemon communicates with other nodes over UDP and interacts with the kernel only when necessary, e.g., to add/delete a route to/from the kernel routing table, to enable IP forwarding, etc. We use the default values for parameters as used in [61].

B. Discovered Attacks

Attacks caused by malicious actions. We rediscovered several attacks in OLSRD based on message delivery and lying actions that have a significant impact on the application performance. By design, OLSR is known to be susceptible to these attacks [54]–[56]. We observe a 100% PDR in a benign scenario. We measure the impact of the attacks on PDR as a function of the number of adversaries in the network. Due to space limitations, we omit the graphs from this section.

Replay HELLO. When a node receives a HELLO message from another node, it adds the node to its neighbor list and starts broadcasting a new HELLO message. Based on the HELLO messages, nodes learn about their one hop neighborhood and select their multipoint relays that forward TC messages. By replaying a HELLO received from a neighbor, a malicious node can disrupt the routing service of its benign neighbors that are not direct neighbors of the originator of the HELLO. We observe the PDR to be around 80% on average, regardless of the number of attackers in the network.

Drop TC 100%. A TC message traverses the entire network via multipoint relays. TC messages are important because a node considers all the received TC messages to infer the network topology and thus, establishes a route to every other node. Therefore, an attack on TC messages is more damaging in that it will lead to inconsistencies in routing table of benign nodes. We observe that dropping TC messages results in at most 50% drop in PDR. Note that while selecting malicious nodes randomly in our experiments, we do not add any constraints on the selection procedure.

Lie Pkt Seq. By design, OLSR follows the heterogeneous packet format where each packet is ordered by a sequence number. Before sending out a packet, a malicious proxy can replace the sequence number of the packet with a fake value (e.g. 0). This malformed packet causes disruption in route calculation. Four malicious nodes can drop the PDR of OLSRD up to 69%.

Blackhole/wormhole attacks. We measured the PDR obtained by OLSRD at the presence of three different configurations of blackhole and wormhole attackers: one blackhole attacker, two independent blackhole attackers, a colluding pair blackhole attackers connected through a private channel. With the increase in blackhole nodes the PDR decreases as low as 50%. The combination of the wormhole and blackhole attackers makes the attack more significant as the PDR drops to around 30%.

TABLE IV
ATTACKS AND BUGS (RE-)DISCOVERED BY TURRET-W. ATTACKS/BUGS WITH (*) MEANS NEWLY DISCOVERED.

Protocol Impl.	Discovery Type	Name	Description
AODV-UU 0.9.6 [12], Reactive, Updated: Apr 13, 2011	Attack*	Lie RREQ type 2	Lie about RREQ message type by setting to 2 (RREP) (causes crashing)
	Attack [10]	Lie RERR type 1	Lie about RERR message type by setting to 1 (RREQ)
	Attack [8], [10]	Lie RREP hop 0	Lie about the hop count in route response to be 0
	Attack [10]	LieAdd RREQ reqid 10	Increment the route request id of route request by 10
	Attack [8], [10]	LieAdd RREP destsq 10	Increment the destination sequence number of route response by 10
	Attack [8], [10]	Replay RREP	Replay both route response and hello messages
	Attack [8]	Blackhole	Drop all data packets
	Attack [8], [10]	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 [14], Reactive, Updated: Jan 31, 2003	Attack [53]	Drop RDP 100%	Drop each route request message
	Attack [53]	Delay REP 2s	Delay forwarding of route response message by 2 seconds
	Attack [53]	Divert REP	Divert route response message
	Attack [53]	Drop ERR 100%	Drop route error message
	Attack [9]	Blackhole	Drop all data packets
	Attack [9]	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
OLSRD 0.6.3 [13], Proactive, Updated: Jun 5, 2011	Attack [54]–[56]	Replay HELLO	Replay a HELLO message received from a neighbor
	Attack [54]–[56]	Drop TC 100%	Drop all topology control messages
	Attack [54]–[56]	Lie Pkt Seq 0	Lie about the sequence number in olsr pkt to be 0
	Attack [54], [55]	Blackhole	Drop all data packets
	Attack [54], [55]	Wormhole + Blackhole	Colluding malicious nodes drop all data packets
DSDV [15], Proactive, Updated: Sep 24, 2011	Attack*	Lie HELLO seq 255	Lie about own sequence in HELLO messages with 255 (cause crashing)
	Attack*	Lie HELLO dstseq 255	Lie about the dest. sequences in HELLO messages with 255 (cause crashing)
	Attack*	Lie HELLO hopcount 255	Lie about the hopcount in HELLO messages with 255 (cause crashing)
	Attack*	Lie HELLO dstseq 254	Lie about the dest. sequences in HELLO messages with 254 (cause crashing)
	Attack [57]	Replay HELLO	Replay all HELLO messages received from neighbors
	Attack [57]	Drop HELLO 100%	Drop all HELLO messages
	Attack [57]	Divert HELLO	Divert own HELLO messages
	Attack [58]	LieAdd HELLO seq 10	Increment the own sequence number of HELLO messages by 10
	Attack [58]	LieAdd HELLO dstseq 10	Increment each destination sequence number in HELLO messages by 10
	Attack [11], [57]	Blackhole	Drop all data packets
	Attack [57]	Wormhole + Blackhole	Colluding malicious nodes drop all data packets
Batman-adv 2014.1.0 [16], Proactive, Updated: Mar 13, 2014	Attack [7], [59]	Replay OGM	Replay an OGM message received from a neighbor
	Attack [7]	Lie OGM TQ 255	Lie about the transmit quality in OGM to be 255
	Attack [7]	Replay Unicast4Addr	Replay an Unicast4Addr message received from a neighbor
	Attack [7]	Lie Unicast type 0	Lie about the type of an Unicast message to be 0
	Attack [7]	Lie Unicast4Addr type 0	Lie about the type of an Unicast4Addr message to be 0
	Attack [59]	Blackhole	Drop all data packets
	Attack [59]	Wormhole + Blackhole	Colluding malicious nodes drop all data packets

VII. CASE STUDY 4: DSDV

We now describe how we used Turret-W to test DSDV [2]. All discovered attacks are shown in Table IV.

A. Protocol Description

The DSDV (destination-sequenced distance-vector) routing protocol is based on the Bellman-Ford family of algorithms that utilize distance vectors to calculate paths, between any two nodes in the network, along which data can be exchanged. DSDV is a table-driven proactive routing protocol, and therefore, each node maintains a routing table consisting of entries for every possible destination (not just the neighbors) along with the cost to reach the destination. As a cost metric, the protocol uses hop-count that is the number of hops a packet has to travel to reach its destination.

Each node periodically advertises its own routing table to its neighbors using HELLO messages. In addition, any changes to the routing table are propagated to other nodes as quickly as possible. These updates may lead to routing loops within the network. To avoid routing loops, each routing update from the node is tagged with a sequence number. Each node is free to choose an even number as the starting sequence number for the routing updates where the node is listed as the destination, but the node increments the sequence number by 2 for each periodic update. A sequence number defines the freshness of the route to the destination. Note that one node cannot change the sequence number tagged with such routing updates made by others. However, in case of a broken/expired link to one of its neighbor, the node can increment the sequence number by 1 and trigger an update mechanism. The nodes receiving this update check the sequence number and if it is an odd number, they remove the corresponding entry from their routing table. Moreover, DSDV uses settling time to dampen the route fluctuations due to node mobility.

Implementation used: We use the DSDV implementation presented in [37] that is developed as part of the Grid project, which is built on the Click modular router [38], and is publicly available from [15]. We call it DSDV-Click. This implementation of DSDV can run either at the user-space using the Click user-space process or the kernel-space using the Click Linux kernel module. We chose the former due to its nature of high portability and easy debugging. At user-space, the Click process loads a network tunnel (`tun`) device, which the process considers as a file descriptor (e.g., `/dev/tun0`) and the operating system considers as a network interface (e.g., `tun0`). The Click process exchanges packets with the operating system's network stack using this tunnel device. We use the default values for parameters as used in [15].

B. Discovered Attacks

Attacks caused crashing. We discovered 4 implementation dependent attacks in DSDV-Click that cause all the neighbors of a malicious node to crash.

Lie HELLO seq or *dstseq with odd values*: We found that there can be multiple sequence numbers in a HELLO message. A node places its own sequence number (we refer to it as *seq*)

as well as the sequence number of each destination (we refer to it as *dstseq*) that it is aware of into its HELLO messages. Whenever a node receives such a HELLO message, it checks if each advertised route is active. If so, each of the received sequence numbers must be an even number. Therefore, by simply lying on one or more of these sequence numbers, *i.e.*, by setting a positive odd number, a malicious node can cause each of its neighbors to fail an assertion check and crash.

Lie HELLO hopcount with 255: While advertising routes to other destinations, the originator node also includes *hopcount* (i.e., the number of hops to reach each of them from the originator) into its HELLO messages. We found an attack where an adversary can exploit the integer overflow vulnerability associated with the *hopcount* field, which is one byte in length. The adversary maliciously advertises routes with a value of 255 as the *hopcount*. Whenever one of the adversary's neighbors receives such advertisements and decides to update its routing table, the node adds 1 to the received *hopcount*. This addition overflows the field causing the node itself to crash due to an assertion failure.

Lie HELLO dstseq with even values: Turret-W helped us discover another crashing attack that is very subtle and delicate in terms of its execution. In this attack, the malicious node always modifies the route advertisements with a positive even number as the destination sequence number (*dstseq*), which apparently looks correct according to the protocol. However, a positive even number as *dstseq* is not correct for an advertisement of an expired route. Therefore, whenever the malicious node sends advertisements about the recently expired routes with a positive even number as *dstseq*, an assertion check on the neighbors causes them to crash.

Attacks caused by malicious actions. Like other protocols, we also found several attacks in DSDV-Click that impair the application performance. By design, DSDV [2] is known to be susceptible to these attacks [11], [57], [58]. We measure the changes in PDR achieved by the application as a function of the number of adversaries in the network where each node employs the DSDV-Click as the underlying routing protocol. In the benign case, we observe a 100% PDR. Due to space limitations, we omit the graphs.

LieAdd HELLO seq and *LieAdd HELLO dstseq*. Recall that, in DSDV, each node maintains a routing table consisting of entries for all possible destinations (not only neighbors) and periodically advertises its routing table to its neighbors using beacon messages (i.e., HELLO). Each of these messages contains the sequence number (*seq*) of the node itself along with zero or more entries for other destinations that the node is aware of at that very moment. Each additional entry includes the received sequence number (*dstseq*) of the corresponding destination. A sequence number tagged with a route defines the freshness of the route— a higher sequence number indicates a more recent route. Therefore, whenever a node receives a HELLO message from another node with the destination sequence number higher than what it is aware of, the node selects this new route. A malicious node can exploit this fact and add a positive even number to the destination sequence number contained in a HELLO message, and this causes the receiving nodes to select the route through the malicious node. Note that instead of a positive even number, if the adversary chooses to add a positive odd number, the attack will cause the neighbors to crash (as explained earlier) since DSDV expects the sequence numbers defined by the originators to be positive even numbers.

According to our experimental results, adding positive even numbers to the *dstseq* field is more damaging than performing the same attack on the *seq* field. We can attribute this to the fact that by modifying the *seq* field the adversary just offers a more recent route to itself whereas by modifying the *dstseq* fields the adversary offers more recent (but not legitimate) routes to other destinations containing itself on these paths. Our experiment results show that the achieved PDR can drop from 62% to 20% with the increase in the number of adversaries when the adversaries perform such attacks on the *dstseq* fields. However, in case of such attacks on the *seq* field, we observe the PDR to drop from 86% to 72%.

Drop HELLO and *Divert HELLO*. The DSDV protocol requires nodes to exchange only HELLO messages as control packets pertaining to the routing service to establish a routing table. Therefore, when a malicious node drops all of its own HELLO messages, no other nodes within the network will ever be aware that the malicious node is active. As a result, the source node selects a path longer than the shortest one if the malicious node is on that shortest path. Similarly, when a malicious node sends its own HELLO messages to randomly selected nodes instead of broadcasting the messages, only a few nodes will know about the existence of this node. However, every node eventually learns the route to the malicious node due to the route advertisement mechanism of the DSDV protocol. The cost metric of these routes may not be the real optimum value. Consequently, the source may end up using a longer path than the original shorter one. In both the cases, we observe the PDR drops roughly from 95% to 65% with the increase in the number of adversaries.

Replay HELLO. In this attack, an adversary re-broadcasts the HELLO messages received from the neighboring nodes without any modification. As a result, any two benign neighbors of the adversary that are multiple hops away from each other (in reality) consider themselves as 1-hop neighbors. Moreover, these false links never expire as long as the attack continues. In this case, we observe the PDR changes from 88% to 50% as the number of adversaries increases.

Blackhole/wormhole attacks. To test DSDV-Click in the presence of blackhole/wormhole attacks, we followed the same approach as for the other protocols. In case of one blackhole attacker, we observe a PDR of 80% whereas the PDR drops to 63% when we introduced another blackhole adversary. In case of the wormhole attack, the PDR drops to 49%.

VIII. Case Study 5: BATMAN

We now describe how we used Turret-W to test BATMAN [7]. All discovered attacks are shown in Table IV.

A. Protocol Description

BATMAN is a proactive routing protocol for multi-hop wireless adhoc networks. Unlike link-state protocols, it does not determine the whole path to the destination, nor does it requires the global view of the network topology to route packets. Instead, it requires each node to maintain only the best

next hop to every other node in the network using collective intelligence, similar to a distance-vector protocol. Therefore, information about any topological change in the network does not need to be instantly spread throughout the network.

Each node periodically broadcasts an originator message (OGM) to inform its existence to its neighbors. The neighbors then rebroadcast the message to their neighbors and so on and so forth. Therefore, every node is aware of the existence of every other node in the network but records only the list of direct neighbors that it has received such messages from. The best next-hop to each destination is selected based on a metric called Transmit Quality (TQ), which measures the probability of a successful transmission of a packet on the link between the node and the next-hop. As a result, each node only knows who to handover the data (encapsulated in Unicast messages) destined to a node that is multiple hops away. The data is handed over to the best next-hop neighbor, which in turn repeats the mechanism until reaches its destination.

BATMAN utilizes a distributed ARP table (DAT) to enable nodes to perform faster ARP lookup operations. In essence, DAT mechanism creates an ARP cache distributed across the nodes by storing ARP entries as the ARP requests/responses travel through the network. Unlike traditional ARP requests, given an IPv4 address, a node can identify the group of nodes that may contain the related ARP entry by utilizing a distributed hash function. Instead of broadcasting, requests are sent as unicast messages (Unicast4Addr). If there is no response to the request, the requester node can fallback to the traditional ARP mechanism and broadcast the ARP request.

Implementation used. We use Batman-adv-2014.1.0 (referred below as Batman-adv) implementation publicly available from [16]. This implementation is a kernel-space implementation running at the data link layer where both the routing information and the data traffic are encapsulated and forwarded as raw Ethernet frames. Hence, the network communication does not depend on IP. The protocol emulates a virtual network switch connecting all the nodes as if the nodes are link local, and therefore unaware of the network topology.

B. Discovered Attacks

Attacks caused by malicious actions. We rediscovered several attacks on Batman-adv based on message delivery and lying actions that decrease the PDR below the accepted threshold. By design, the BATMAN protocol is known to be susceptible to these attacks [7], [59]. In case of our benign experiments, we observe a 97% PDR. We then measure the impact of the attacks on PDR as a function of the number of adversaries in the network. Due to space limitations, we omit the graphs from this section.

Reply OGM. By replaying the originator messages (OGMs) received from a node, an adversary can induce its benign neighbors to consider the originator as a direct neighbor since OGMs are used to announce the existence of nodes in the network. This disrupts the routing service substantially since these replayed OGMs propagate through the network thereby affecting the best next-hop selection at the nodes that are closer to the attacker than to the originator. This attack is more damaging than others since replaying OGMs causes these pseudo-links never to expire. We observe that the PDR decreases from 24% to 6% with increasing adversaries.

Lie OGM TQ. When sending an OGM, the originator initializes the transmit quality (TQ) field with its maximum value of 255. Prior to re-broadcasting an OGM, the forwarding node sets the TQ field with a value that is the TQ of the received OGM times its measured TQ towards the last hop node via which it received the OGM. As a result, the TQ field of an OGM indicates the probability of successful transmission of a packet towards the originator along the path the OGM has traversed. A malicious node exploits this fact by setting the TQ field of all outgoing OGM to 255 thereby enticing the neighbors to select itself as the best next-hop neighbor towards the originator. Our experiment results show that the PDR drops from 77% to 54% as the number of adversaries increases.

Replay Unicast4Addr. When the source has to retrieve the MAC address of the destination, it computes the group of nodes that may contain the related ARP entry and sends Unicast4Addr messages. In this attack, an adversary replays all the Unicast4Addr messages containing either ARP request or response. Though this attack cannot directly disrupt the routing table, it can overload the network with Unicast4Addr packets when the number of adversaries increases in the network because the adversaries collectively create a ripple effect by replaying each received Unicast4Addr message. Moreover a Unicast4Addr message is quite smaller in length compared to a message carrying data traffic. As a result, this ripple effect affects the forwarding of the data traffic through the network. In our experiments, we observe the PDR drops from 77% to 47% as the number of adversaries increases.

Lie Unicast type and *Lie Unicast4Addr type*. The source encapsulates the data traffic in Unicast messages and hands over to the best next-hop neighbor and so does the next-hop neighbor until the data reaches the destination. By lying on the type field of a Unicast message, the adversary disrupts the data forwarding as the modified Unicast message is not interpreted as the data message. We observe the PDR drops from 70% to 9% with the increase in the number of adversaries. On the other hand, when the adversary modifies the type field of a Unicast4Addr message, it can disrupt the ARP request for a while. However, after a timeout, the requester falls back to traditional ARP mechanism and broadcasts the ARP request, which eventually reaches the destination or some intermediate nodes that can reply with the related ARP entry. Therefore, in case of this attack, we observe that the attack is only effective when the number of adversaries in the network is larger than 2 causing the PDR to drop to 63%.

Blackhole/wormhole attacks. We test Batman-adv against blackhole/wormhole attacks in the same way as we did for other protocols. We observe that the PDR drops from 60% to 47% as the number of blackhole attacker increases from 1 to 2. When these two attackers collude to create a wormhole, the PDR drops to 42%.

IX. Related Work

Model checking techniques [62]–[64] have been used to verify the correctness of protocol models. Once the model is specified in a high-level modeling language, its correctness is verified mathematically. Many works extended such methods to consider the wireless environment [65]–[68].

While model checking techniques have been helpful to show the correctness of the model of a protocol, the high-level descriptions abstract away many details of the actual implementation resulting in missing vulnerabilities in the abstract model, which may manifest in the actual implementation. Exploration based model checking techniques [69]–[72] apply model checking directly on implementations. Specifically, CMC [70] has been applied on different implementations of the AODV protocol, but requires the implementations to be ported to its specialized runtime environment.

Without denying the benefit of model checking, our work is orthogonally different since we focus on bugs/attacks that impair the performance of the protocol in actual executions of the implementation. In addition, one can argue to establish ground truth using model checking or using formally verified reference implementation like [73]. However, note that being able to model check liveness and performance properties is a challenging problem, and to the best of our knowledge, existing model checking techniques cannot check performance properties. Also, to the best of our knowledge, there are no verified reference implementations for the protocols we tested.

Systematic fault injection is another popular method to improve software robustness [74], [75]. Unlike model checking or symbolic execution, fault injection focuses on exceptional behavior of software by injecting faults. However, such works do not consider adversarial environments as ours where we inject malicious faults that are tailored to imitate attackers.

Several network emulation tools have been developed, for example, NIST Net [76] catering wired networks and Emulab [34], Orbit [36] catering wireless networks. Such tools designed for wireless networks could conceptually replace the NS-3 network emulator and the virtualization-based nodes, but would require the user to provide a separate (and malicious) implementation of the routing protocol under test and that is for each adversary in the network. Whereas, we do not require any such malicious version of the protocol under test.

There have been some recent effort on finding attacks automatically in implementations [25], [27], [28], [40]. Kothari et al. [27] automatically find attacks that manipulate control flow by modifying messages using static analysis by relying on a priori knowledge about vulnerability. Stanojevic et al. [40] automatically search for gullibility in two-party protocols by leveraging a variety of techniques: packet-dropping and packet header modifications. Lee et al. [25] automatically discover performance attacks caused by insiders in distributed systems without requiring instrumented implementation. All these works except [28] require the implementation to be written in specific languages.

X. 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, ARAN, OLSR, DSDV, and BATMAN. In total, we (re-)discovered 37 adversarial attacks capable of either crashing the benign nodes or reducing their performance by disrupting the routing service and 3 implementation bugs that impair the protocol performance in benign environment.

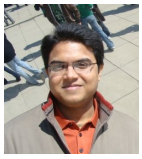
Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant Number CNS-1223834. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

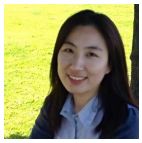
References

[1] E. Hoque, H. Lee, R. Potharaju, C. Killian, and C. Nita-Rotaru, "Adversarial testing of wireless routing implementations," in *WiSec*, 2013.
[2] C. Perkins and P. Bhagwat, "Highly dynamic DSDV for mobile computers," *ACM Sigcomm CCR*, 1994.
[3] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot, "Optimized link state routing protocol for ad hoc networks," in *IEEE Mtc*, 2001.
[4] C. E. Perkins and E. M. Royer, "Ad-hoc On-Demand Distance Vector Routing," in *IEEE WMCSA*, 1997.
[5] D. Johnson and D. Maltz, "Dynamic source routing in ad hoc wireless networks," *Mobile computing*, 1996.
[6] 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.
[7] A. Neumann, C. Aichele, M. Lindner, and S. Wunderlich, "Better Approach To Mobile Ad-hoc Networking (B.A.T.M.A.N.)," http://tools.ietf.org/html/draft-wunderlich-openmesh-manet-routing-00.
[8] M. Zapata and N. Asokan, "Securing ad hoc routing protocols," in *WiSE*, 2002.
[9] 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.
[10] K. Sanzgiri, B. Dahill, B. Levine, C. Shields, and E. Belding-Royer, "A secure routing protocol for ad hoc networks," in *IEEE ICNP*, 2002.
[11] Y. Hu, A. Perrig, and D. Johnson, "Ariadne: A secure on-demand routing protocol for ad hoc networks," *WN*, 2005.
[12] "AODV-UU," http://sourceforge.net/projects/aodvuu.
[13] "OLSRD," http://www.olsr.org.
[14] "ARAN," http://prisms.cs.umass.edu/arand, accessed Nov 2012.
[15] "Click modular router," http://www.read.cs.ucla.edu/click.
[16] "Batman-adv," https://goo.gl/xersqM, accessed May 2014.
[17] F. De Renesse and A. Aghvami, "Formal verification of ad-hoc routing protocols using spin model checker," in *IEEE Melecon*, 2004.
[18] "Network Simulator 2." http://www.isi.edu/nsnam/ns/.
[19] X. Zeng, R. Bagrodia, and M. Gerla, "Glomosim: a library for parallel simulation of large wireless networks," *Sigsim*, 1998.
[20] S. Woo and S. Singh, "Scalable routing protocol for ad hoc networks," *Wireless Networks*, vol. 7, no. 5, 2001.
[21] A. Gupta, I. Wormsbecker, and C. Wilhainson, "Experimental evaluation of TCP performance in multi-hop wireless ad hoc networks," in *Mascots*, 2004.
[22] G. Anastasi, E. Ancillotti, M. Conti, and A. Passarella, "Experimental analysis of a transport protocol for ad hoc networks (TPA)," in *MSWiM*, 2006.
[23] 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 *MSWiM*, 2004.
[24] S. M. ElRakabawy and C. Lindemann, "A practical adaptive pacing scheme for TCP in multihop wireless networks," *ToN*, 2011.
[25] H. Lee, J. Seibert, C. Killian, and C. Nita-Rotaru, "Gatling: Automatic attack discovery in large-scale distributed systems," in *NDSS*, 2012.
[26] C. Killian, J. Anderson, R. Braud, R. Jhala, and A. Vahdat, "Mace: language support for building distributed systems," *ACM SIGPLAN Notices*, vol. 42, no. 6, 2007.
[27] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," *ACM Sigcomm CCR*, 2011.
[28] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru, "Turret: A Platform for Automated Attack Finding in Unmodified Distributed System Implementations," in *ICDCS*, 2014.
[29] I. Habib, "Virtualization with kvm," *Linux Journal*, 2008.
[30] "Network Simulator 3," http://www.nsnam.org.

[31] G. G. Finn, "Routing and addressing problems in large metropolitan-scale internetworks," DTIC Document, Tech. Rep., 1987.
[32] B. Karp and H.-T. Kung, "GPSR: Greedy perimeter stateless routing for wireless networks," in *MobiCom*, 2000.
[33] M. Bahr, "Update on the hybrid wireless mesh protocol of IEEE 802.11s," in *MASS*, 2007.
[34] "Emulab - network emulation testbed," http://www.emulab.net/.
[35] P. Mahadevan, A. Rodriguez, D. Becker, and A. Vahdat, "Mobinet: a scalable emulation infrastructure for ad hoc and wireless networks," *Sigmobile CCR*, 2006.
[36] "Orbit," http://www.orbit-lab.org.
[37] B. A. Chambers, "The grid roofnet: a rooftop ad hoc wireless network," Ph.D. dissertation, Massachusetts Institute of Technology, 2002.
[38] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM TOCS*, vol. 18, no. 3, 2000.
[39] "Grid project," http://pdos.csail.mit.edu/grid/.
[40] M. Stanojevic, R. Mahajan, T. Millstein, and M. Musuvathi, "Can you fool me? towards automatically checking protocol gullibility," in *HotNets*, 2008.
[41] S. Jero, H. Lee, and C. Nita-Rotaru, "Leveraging state information for automated attack discovery in transport protocol implementations," in *DSN*, 2015.
[42] P. Kyasanur and N. H. Vaidya, "Routing and link-layer protocols for multi-channel multi-interface ad hoc wireless networks," *ACM SIGMOBILE MC2R*, 2006.
[43] Y. Peng, Y. Yu, L. Guo, D. Jiang, and Q. Gai, "An efficient joint channel assignment and qos routing protocol for ieee 802.11 multi-radio multi-channel wireless mesh networks," *JNCA*, 2013.
[44] "Iperf," http://sourceforge.net/projects/iperf.
[45] 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.
[46] A. Alvarez, R. Orea, S. Cabrero, X. G. Pañeda, R. García, and D. Melendi, "Limitations of network emulation with single-machine and distributed ns-3," in *SIMUTools*, 2010.
[47] "RFC 3561," http://tools.ietf.org/html/rfc3561.
[48] "Netfilter," http://www.netfilter.org/.
[49] N. Horman, "Understanding and programming with netlink sockets," http://www.smacked.org/docs/netlink.pdf, 2004.
[50] K. Sanzgiri, D. LaFlamme, B. Dahill, B. Levine, C. Shields, and E. Belding-Royer, "Authenticated routing for ad hoc networks," *JSAC*, 2005.
[51] "ASL," http://sourceforge.net/projects/aslib.
[52] "OpenSSL toolkit," http://www.openssl.org/.
[53] 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*, vol. 2, no. 4, 2009.
[54] T. Clausen and E. Baccelli, "Securing olsr problem statement," *IETF INTERNET-DRAFT, draft-clausen-manet-solsr-ps-00. txt*, 2005.
[55] C. Adjih, D. Raffo, and P. Mühlethaler, "Attacks against olsr: Distributed key management for security," in *OLSR Interop/Workshop*, 2005.
[56] C. Adjih, T. Clausen, P. Jacquet, A. Laouiti, P. Muhlethaler, and D. Raffo, "Securing the olsr protocol," in *IFIP Med-Hoc-Net*, 2003.
[57] Y. Hu, D. Johnson, and A. Perrig, "SEAD: Secure efficient distance vector routing for mobile wireless ad hoc networks," *Ad Hoc Networks*, vol. 1, no. 1, 2003.
[58] T. Wan, E. Kranakis, and P. Van Oorschot, "Securing the destination-sequenced distance vector routing protocol (S-DSDV)," in *Information and Communications Security*. Springer, 2004.
[59] E. G. Graarud, "Implementing a Secure Ad Hoc Network," Master's thesis, Norwegian University of Science and Technology, 2011.
[60] "RFC3626 - Optimized Link State Routing Protocol (OLSR)," http://www.ietf.org/rfc/rfc3626.txt.
[61] "Olsrd source package in Debian," https://goo.gl/tzUKaE.
[62] G. Holzmann, "The model checker SPIN," *TSE*, vol. 23, no. 5, 1997.
[63] K. McMillan, "Symbolic model checking: An approach to the state explosion problem," Ph.D. dissertation, CMU, PA, USA, 1992.
[64] D. Dill, A. Drexler, A. Hu, and C. Yang, "Protocol verification as a hardware design aid," in *ICCD*, 1992.
[65] S. Nanz and C. Hankin, "A framework for security analysis of mobile wireless networks," *Theoretical Computer Science*, vol. 367, no. 1, 2006.
[66] S. Chiyangwa and M. Kwiatkowska, "A timing analysis of aodv," *FMOODS*, 2005.
[67] K. Bhargavan, D. Obradovic, and C. Gunter, "Formal verification of standards for distance vector routing protocols," *JACM*, vol. 49, no. 4, 2002.
[68] I. Zakiuddin, M. Goldsmith, P. Whittaker, and P. Gardiner, "A methodology for model-checking ad-hoc networks," *MCS*, 2003.
[69] P. Godefroid, "Model checking for programming languages using verisoft," in *POPL*, 1997.
[70] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill, "CMC: Pragmatic approach to model checking real code," in *OSDI*, 2002.
[71] M. Musuvathi and D. Engler, "Model checking large network protocol implementations," in *NSDI*, 2004.
[72] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *ASE*, vol. 10, no. 2, 2003.
[73] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. Zinzindohoue, "A messy state of the union: Taming the composite state machines of tls," in *S&P*, 2015.
[74] P. Marinescu and G. Candea, "Efficient testing of recovery code using fault injection," *ACM ToCS*, 2011.
[75] 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.
[76] M. Carson and D. Santay, "Nist net: a linux-based network emulation tool," *ACM SIGCOMM CCR*, vol. 33, no. 3, pp. 111–126, 2003.



Endadul Hoque is a Postdoctoral Research Associate at Northeastern University. He obtained his PhD in Computer Science from Purdue University in 2015, MS degree Marquette University, in 2010 and BS degree from Bangladesh University of Engineering & Technology (BUET) in 2008. His research interests include verification and testing for network protocols and distributed systems.



Hyojeong Lee is a softwar engineer at Google. She obtained her PhD degree in Computer Science from Purdue University in 2014, BS and MS degrees from Hongik University, Korea. Her research focus is providing frameworks and methods to automatically test distributed system implementations.

Rahul Potharaju is a scientist at Microsoft. He earned his PhD degree in Computer Science from Purdue University in 2014 and MS degree from Northwestern University in 2009. His research focuses on analyzing the reliability aspects of datacenters from both a hardware and software perspective using a measurement-driven approach.

Charles Killian is a software engineer at Google, Inc. and an adjunct assistant professor in the department of Computer Science at Purdue University. He received his BS degree from NC State University, MS degree from Duke University and PhD degree in Computer Science from University of California, San Diego. He is a recipient of a 2011 NSF CAREER award and regularly serves on program committees for conferences on distributed systems and computing. His research interests include techniques for building, testing, and debugging a wide range of distributed and non-distributed systems and networks.

Cristina Nita-Rotaru is a Professor of Computer Science in the College of Computer and Information Science at Northeastern University and an adjunct professor at Purdue University. She received BS and MS degrees from Politechnica University of Bucharest, Romania, in 1995 and 1996, and a PhD degree in Computer Science from Johns Hopkins University in 2003. She served on the technical program committee of over 80 conferences and workshops in networking, distributed systems, and security. She received the NSF CAREER award in 2006. Her research interests include security and fault-tolerance for distributed systems and network protocols.