2007

Specification Synthesis for Monitoring and Analysis of MANET Protocols

Natalia Stakhanova
Iowa State University

Samik Basu
Iowa State University, sbasu@iastate.edu

Zhang Wensheng
Iowa State University, wzhang@iastate.edu

Xia Wang
Iowa State University

Johnny S. Wong
Iowa State University, wong@iastate.edu

Specification Synthesis for Monitoring and Analysis of MANET Protocols

Natalia Stakhanova   Samik Basu   Wensheng Zhang   Xia Wang   Johnny Wong

Department of Computer Science
Iowa State University
Ames, IA 50011 USA
{*ndubrov, sbasu, wzhang, jxiawang, wong*}@iastate.edu

Abstract

This paper introduces an approach to automatic synthesis of the specification models of routing protocol behavior from the observed flow of the network traffic. In particular, our technique generalizes the monitored sequences of routing messages constructing a high-level abstract view of the protocol. The basis of our method is similar to Inductive Logic Programming technique that derives a sound hypothesis from the individual examples. We conduct preliminary experiments on the example of AODV and DSR ad-hoc routing protocols and discuss the effectiveness of the generated specification models in detecting protocol misuses.

1   Introduction

Systems whose execution dynamics are hard to predict or are prohibitively large to analyze statically are typically monitored at run-time to ensure correctness, reliability and security of the computing environment on which the systems are being deployed. Specifically, in the domain of intrusion detection and prevention run-time monitoring has gained prominence and provided viable results. Against this background, we focus on the problem of monitoring and analysis of Mobile Ad-hoc Network (MANET) protocols.

MANET is a network of loosely connected (without any central infrastructure) aggregation of mobile computing nodes where messages are transferred from one part of the network to another via multi-hop wireless links. Each mobile node acts as a router, cooperatively constructing routes according to proactive [24, 5], reactive [25, 13] or hybrid [8] routing protocols, and forwarding messages for other nodes. Without any prior requirement on infrastructure, the MANET helps in fast and effective deployment of distributed computing/communication facility in places which are not congenial to infrastructure deployment. However, the infrastructureless, mobile and cooperative natures also cause the MANET to be more dynamic, indeterministic and vulnerable than conventional infrastructured networks, which makes it harder to monitor and analyze the behavior of the network.

Being inherently decentralized and dynamic, ensuring and enforcing security and normalcy in MANET via run-time monitoring is a challenging task and is the focus of intense research in recent years. Run-time monitoring aims at discovering the abnormalities in the execution by comparing the execution against some specific set of normal and/or abnormal behavior. It can be broadly classified into three categories a)misuse-based (b) anomaly-based and (c) specification-based [29].

The misuse-based technique relies on pre-specified attack signatures, and any execution sequence matching with a signature is flagged as abnormal. An anomaly-based approach, on the other hand, typically depends on normal patterns, and any deviation from normal is classified as malicious or faulty. Unlike misuse-based detection, an anomaly-based approach can detect previously unknown abnormalities. However, it relies on machine learning techniques which can only classify pre-specified behavioral patterns, and suffers from a high rate of false positives [19]. A specification-based technique operates in a similar fashion to an anomaly-based method detecting deviations from the specified legitimate system behavior. However, as opposed to anomaly detection, a specification-based approach requires user guidance in developing a model of valid program behavior in a form of specifications. This process, though tedious and reliant on user-expertise, is more accurate than an anomaly-based technique.

In this context, we propose to develop *models* of MANET protocols via run-time monitoring. On one hand, the technique is close to anomaly detection which relies on automated learning of run-time patterns. On the other hand, our technique aims to generalize the learnt patterns so as to incorporate in the models certain degree of abstraction. In this sense, it is close to *specification models* which are manually synthesized and represent high-level abstract view of the protocol.

Our approach is based on Inductive Logic Programming (ILP) method that induces a hypothesis from individual observations and background knowledge. In our setting we derive an abstract model of protocol behavior from the examples of its executions, where each example represents a sequence of routing messages initiated during a single route discovery. Taking advantage of the commonality of routing traffic, we develop a generalization algorithm for constructing *models* of routing protocols in automated fashion and present a generated specifications for AODV and DSR protocols. The generated *models* are easily readable and can be effective in detecting protocol misuses as shown by our preliminary experiments.

Our model can be effectively employed for:

1. Anomaly-based intrusion detection of protocols. The execution of the protocol can be monitored against the generated model and any deviation will be flagged as an anomaly.

2. In the event, a manual specification of the protocol is available, our model can be used to validate the correctness of the specification, i.e. whether the specification indeed represents the implementation of the protocol. Specifically, manual specification, requiring expert knowledge and careful development, may fail to provide a complete model. Our generated model can potentially identify these holes in the specification and help to develop specifications efficiently.

3. If a specification is not available, our model can also be used as a replacement to analyze the protocol statically via model checking. Often certain minor oversight in the implementation of the protocol may result in incorrect behavior. Model checking techniques have the potential to identify such problems if a manageable specification of the implementation is provided.

Organization. The reminder of the paper is organized as follows: A brief overview of related work is given in Section 2. Section 3 presents an overview and the details of the proposed approach to constructing specifications. Experimental setup and results are given in Section 4. Section 5 concludes the paper and provides direction for future work.

2   Related Work

As specification-based approach showed to be beneficial in many areas including software testing [3, 4] and intrusion detection [15, 17, 11, 12], a significant amount of research has been focused in this direction.

One of the first specification-based models was introduced by Ko [17, 18, 15]. Ko's initial work [17] has been focused on the monitoring of the privileged programs executions using audit trails. While the number of potential program vulnerabilities is unknown, the intended program behavior is limited and can be specified in a concise fashion in a form of specification.

Another specification-based approach focused on monitoring program behavior has been proposed by Sekar [28, 30]. In addition to the monitoring program executions through system calls, this technique aims to enforce specified legal behavior through isolation of compromised processes. Later, Sekar et al. [29] proposed a specification-based anomaly detection technique. This work aimed at augmenting machine learning techniques with high-level specifications to achieve a high degree of precision in detecting anomalies in software. They manually developed high-level specifications (as finite state machines) of software systems and annotated them using statistical information learnt via machine-learning technique.

While the above techniques are mostly based on the monitoring of the program behavior, SHIM (System Health and Intrusion Monitoring) approach [16] employs specifications in the form of constraints that describe valid system behavior. Checking the validity of these constraints during run-time although does not directly detect a potential attack, catches its manifestation.

One of the major downsides of the specification-based approach is the necessity to develop the system specifications manually. As this process is time-consuming and error-prone, automatic generation of specifications is highly beneficial.

As such, Wagner and Dean [34] employed a static analysis to automatically derive program specifications. They proposed three methods for generating the specifications: *callgraph model*, based on control-flow analysis of code, *abstract stack model* represented as pushdown automaton and *digraph model* based on all possible 2-gram sequences of system calls derived from control-flow graph.

Ko [15] presented a machine-learning approach for developing security specifications automatically based on *Inductive Logic Programming*(IPL) method. His work was focused on generating program behavioral specifications at the system call level using an ILP tool, *Progol* and aimed to reflect security properties of the programs. Although his approach has showed the advantages of automatic development of specifications, it lacked the ability to represent ordering of program operations.

Our work was inspired by this technique. In addition to modeling the ordering of events in program executions, we extend Ko's approach by considering a network setting, specifically, network routing protocols.

While ensuring completeness of the developed specifications is a common difficulty of the specification-based models, only a few approaches have attempted to address this problem. Song at el. [31] proposed a formal framework based on ACL2 for analysis and verification of specifications. Since the system specifications are developed based on certain assumptions, deploying a mechanism to secure these assumptions will improve the security of the system.

More recently, several techniques applied specification-based approach to detect attacks against routing protocols, specifically, AODV [32], OLSR [33, 23] and cryptographic protocols [12]. However, in all these approaches specifications were developed manually.

3 Synthesis of pecification Models for MANET Protocols

The central tenet of our technique is that specifications of (MANET) protocols can be synthesized from the flow of the network traffic. A specification, in this context, is a form of a graph where the nodes/vertices represent the configuration of the protocol and directed-edges between vertices define how the protocol evolves from one configuration to another. Such a specification model will explicitly include all possible *monitored* behavior of the protocol that is safe. Furthermore, we generalize the specification with an attempt to include additional behavior of the protocol that is *not known* to be anomalous. The aim of such a synthesis procedure is to develop a specification of the protocol that can be used to flag anomalous run-time behavior of protocol (based on the specifications of valid protocol behavior), detect intrusive behavior (based on the developed specifications of invalid protocol behavior), validate existing specifications developed manually, and analyze protocol against desired properties.

Run-time monitoring of protocol against its specification of normal behavior has been well-studied [32, 33, 23]. To be best of our knowledge, this is first attempt to synthesize such specifications automatically. In this paper, we will present automatically generated specifications of two MANET protocols. Secondly, if a specification is developed apriory manually (manual specification), our synthesized specification can be used to validate the completeness/correctness of the manual specification. Manually developed specifications, though typically developed by experts, may be incomplete due to un-intentional oversight of the specifier and complexity of the protocol. We claim that automatically synthesized protocol specification can be complimentary to the manually developed ones. Finally, protocol specifications are used to verify conformance to desired properties (e.g. *every request message is eventually followed by a corresponding reply message*) and are specifically important to detect subtle flaws in the protocol that may remain un-noticed [6]. In fact, a typical automated verifier (model checkers [9, 2, 27]) takes as input protocol specifications in the form of a model (Kripke Structure, Labeled Transition Systems—see [10] for details) as synthesized by our technique.

The basis of our specification synthesis (and generalization) is, in principle, similar to *inductive logic programming* (ILP) approach presented in the next section. In the following sections, we proceed with the brief overview of ILP problem followed by the description of our synthesis algorithm and its relationship with ILP.

3.1 Background: Inductive Logic Program

Given a set of examples and background knowledge or facts in the domain of the examples, Inductive Logic Programming (ILP) aims at *inducing* a hypothesis which when interpreted in the context of background knowledge *deduces* the examples [20]. Specifically, ILP is the inverse of natural logical deduction where logical consequences or examples are deduced from the background knowledge. For example, consider a set of edge relations in a graph given as facts

$$
\begin{array}{ll}
\texttt{edge(a,b).} & \texttt{// edge from a to b} \\
\texttt{edge(b,c).} & \texttt{// edge from b to c} \\
\texttt{edge(a,d).} & \texttt{// edge from a to d} \\
\texttt{edge(d,c).} & \texttt{// edge from d to c} \\
\texttt{edge(c,a).} & \texttt{// edge from c to a}
\end{array}
\tag{1}
$$

and the deduction rules

$$
\begin{array}{l}
\texttt{reach(X, Y)} \Leftarrow \texttt{edge(X, Y).// X has a direct edge to Y} \\
\texttt{reach(X, Y)} \Leftarrow \texttt{edge(X, Z)} \wedge \texttt{reach(Z, Y).} \\
\texttt{//X has a direct edge to Z \& Z can reach Y}
\end{array}
\tag{2}
$$

4

In the above, parameters of reach-rules are said to be unbounded. A logic programming engine will evaluate the `reach` rules by "grounding" (substituting) the unbounded parameters to specific values (`a`, `b`, etc in the above example) and deduce that every node in the graph can reach each other.

In contrast to the above, ILP aims to identify the `reach` rule as a hypothesis from a given set of examples. Specifically, if the facts in Equation 1 are given as background knowledge along with a set of examples describing the reachability between the vertices in the graph, ILP can generate the reach-rules of Equation 2 as hypothesis. In other words, the hypothesis when evaluated in the context of given background knowledge includes all the examples.

Formally, if B denotes the background knowledge, E denotes the set of examples and H is the generated hypothesis, then

$$\forall e \in E : \ B \wedge H \models e \ \text{ Completeness property} \tag{3}$$

The above states that all examples can be deduced (modeled by, $\models$) from the hypothesis *and* the background knowledge. This is referred to as completeness property of the hypothesis. Note that, completeness is defined with respect to the examples in E.

In addition to examples, a set of negative examples may also be provided and the requirement imposed on the hypothesis is that it must not lead to given negative examples. If $\bar{E}$ be the set of negative examples:

$$\forall \bar{e} \in \bar{E} : B \wedge H \not\models \bar{e} \ \text{ Consistency property} \tag{4}$$

As with completeness, consistency is also defined only with respect to the negative examples in $\bar{E}$. In the current context, the synthesized specification can be viewed as an hypothesis which will be shown to be both complete and consistent with respect to monitored correct behavior (examples) and known anomalous behavior (negative examples) respectively.

3.2 Protocol Specification from Routing Flows

We model the specification of the protocol, under consideration, using the set of *request-reply* routing flows. A flow is represented by a sequence of routing messages initiated by *route request*. Each message in a sequence represents a sequence state and includes the information corresponding to the routing message (depending on the routing protocol this information can include source and destination IP addresses, hop count, sequence numbers, etc.). An example of such flow in AODV protocol can be $\langle$RREQsent $\to$ RREPsent $\to$ RREPreceived$\rangle$ denoting sending of a request, followed by sending of the corresponding reply, followed by receipt of the reply. The corresponding specification model will contain the above sequence. Formally, a specification model is defined as follows:

Definition 1 (Specification Model) *A specification model $M = (S, s^0, T, A)$ where S is a set of states, $s^0 \in S$ is the start state, T is a set of directed edges $(S \times A \times S)$ and A is the set of edge labels.*

In the above, S are states of the form `RREQsent` and T contains transitions of the form `RREQsent` $\xrightarrow{a}$ `RREPsent` where $a \in A$ is information corresponding to `RREQsent`.

The specification model can be viewed as a *hypothesis* that is generated from a set of example sequences E taking into consideration a set of negative example sequences $\bar{E}$. The construction procedure ensures the following model property.

$$\forall s_1^e \xrightarrow{a_1} s_2^e \xrightarrow{a_2} s_3^e \ldots \xrightarrow{a_{n-1}} s_n^e \in E : \exists s_1 \xrightarrow{b_1} s_2 \xrightarrow{b_2} s_3 \ldots \xrightarrow{b_{n-1}} s_n \in M : s_1 = s^0 \ \wedge \ \forall i : s_i^e = s_i \ \wedge \ b_i = a_i \tag{5}$$

5

```
 1:  void main(){
 2:    for all flow sequences ∈ E {
 3:       Seq_m = verticalMerge(Seq_k, M);
 4:          if (∃ e ∈ Ē |Seq_m ⊨ e )
 5:             horizontalMerge(Seq_k , M);
 6:          else horizontalMerge(Seq_m , M);
 7:    }
 8:    return;
 9: }
10:
11:
12:  Seq_m verticalMerge(Seq_k , M) {
13:    int i=l;
14:     while (i≠k) {
15:          if(s_i=s_{i+1}) {
16:             if (s_i → s_i ∉ M) {
17:                createTransition(s_i,s_i);
18:             } // end of if-then
17:             setParameters(s_i, s_{i+1}, M);
18:             if (i+1 ≠ k) {
19:                createTransition(s_i,s_{i+2});
20:                setParameters(s_i, s_{i+2}, M);
21:             } // end of if-then
22:             remove(s_{i+1});
23:          } // end of if-then
24:          else i=i+1;
25:    } // end of while
26:     return Seq_k;
27: }
```

```
28:  void horizontalMerge(Seq_k, M){
29:  int i=l;
30:    while (i≠k) {
31:       if(s_i ∈ M){
32:          if(s_{i+1} ∈ M){
33:             if (s_i → s_{i+1} ∉ M) {
34:                createTransition(s_i ∈ M,s_{i+1} ∈ M);
35:                setParameters(s_i, s_{i+1}, M);
36:                if (∃ e ∈ Ē | M ⊨ e ) {
37:                   removeTransition(s_i → s_{i+1} ∈ M);
38:                   return;
39:                }
40:             } // end of if-then
41:          } // end of if-then
42:          else{
43:             createState(s_{i+1} ∈ M);
44:             createTransition(s_i ∈ M,s_{i+1} ∈ M);
45:             setParameters(s_i, s_{i+1}, M);
46:             if (∃ e ∈ Ē | M ⊨ e ) {
47:                removeTransition(s_i → s_{i+1} ∈ M);
48:                removeState(s_{i+1} ∈ M);
50:                return;
51:             }
52:          } // end of if-else
53:       } // end of if-then
54:       else {
55:          createState(s_i ∈ M);
56:          if (∃ e ∈ Ē | M ⊨ e ) {
57:             removeState(s_i ∈ M);
58:             return;
59:          }
60:       } // end of if-else
61:       i=i+1;
62:    } // end of while
63:    return;
64: }
```

Figure 1: Pseudo-code for *vertical merge* and *horizontal merge.*

The above ensures that all sequences in the example set are also present in the synthesized model (completeness—see Equation 3). Furthermore,

$$\forall s_1^{\bar{e}} \xrightarrow{a_1} s_2^{\bar{e}} \xrightarrow{a_2} s_3^{\bar{e}} \ldots \xrightarrow{a_{n-1}} s_n^{\bar{e}} \in \bar{E} : \forall s_1 \xrightarrow{b_1} s_2 \xrightarrow{b_2} s_3 \ldots \xrightarrow{b_{m-1}} s_m \in M : s_1 = s^0 \ \wedge \ \exists i : s_i^e \neq s_i \ \vee \ b_i \neq a_i \tag{6}$$

The above ensures that the negative examples are not included as any sequence in the synthesized specification model; either the states are not identical or the edge-labels are inconsistent. This follows from the required consistency property of the model (Equation 4).

3.2.1   Algorithm for Specification Synthesis from Examples

Figure 1 presents an algorithm for developing specification model (Definition 1) from sequences of examples. The algorithm consists of two major procedures. `verticalMerge` merges states in *one* sequence leading to generation of loops in the result. The resultant sequence is a generalized version of the original sequence—the former is a *substring* of the latter. The procedure `horizontalMerge` on the other hand merges sequences resulting from the vertical merging procedure. This leads to (partial) sharing of sequence-states/transitions between multiple sequences.

Procedure `verticalMerge` (Figure 1) takes as input the given network flow sequence Seq_k and a specification model M and returns a new sequence Seq_m where repetitions in Seq_k are generalized
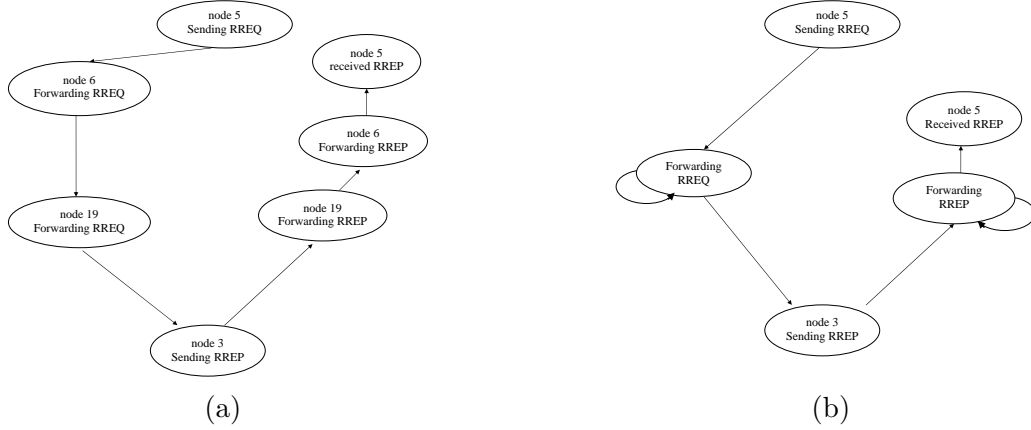
Figure 2: Example of `verticalMerge` procedure: (a) before the procedure (b) after the procedure

as loops (*unbounded repetitions*).

For all states in the sequence Seq_k, `verticalMerge` iteratively checks whether two consecutive states should be merged (Lines 14-15). Once the states to be merged are found a new transition that represents a self-loop is created and relationships between corresponding parameters in this transition are set (Lines 16-18). The subprocedure `setParameters()` identifies the relationships $(=, <, >, \neq)$ between all parameters associated with two given states. One of the merged states is removed and all its out-going transitions are created as out-going transitions of the other. (Lines 18-22).

An example of procedure `verticalMerge` is given in Figure 2[1]. Consider a sequence

$$\left\langle \begin{array}{l} \texttt{RREQ(sent by node 5)} \rightarrow \texttt{RREQ(fwd by node 6)} \rightarrow \\ \texttt{RREQ(fwd by node 19)} \rightarrow \texttt{RREP(sent by node3)} \rightarrow \\ \texttt{RREP(fwd by node 19)} \rightarrow \texttt{RREP(fwd by node 6)} \rightarrow \\ \texttt{RREP(rcvd by node 5)} \end{array} \right\rangle$$

in Figure 2(a). States representing forwarding state (`RREQfwd` and `RREPfwd` nodes) can be merged together into (generalized) states `RREQfwd` and `RREPfwd` with self-loops. The resulting generalized sequence is

$$\left\langle \begin{array}{l} \texttt{RREQ(sent by node 5)} \rightarrow \texttt{RREQfwd(self-loop)} \rightarrow \\ \texttt{RREP(sent by node 3)} \rightarrow \texttt{RREPfwd(self-loop)} \rightarrow \\ \texttt{RREP(rcvd by node 5)} \end{array} \right\rangle$$

as shown in Figure 2(b).

To ensure the *Consistency property* defined in Equation 4 we check whether the resultant generalized sequence from `verticalMerge` becomes a superstring of any of the negative examples $\in \bar{E}$ (Line 4). If this is the case, then the generalization is not performed and the original input sequence Seq_k is inserted into a specification model M through `horizontalMerge` procedure which takes as input a sequence Seq_k (either in generalized or in original form) and a specification model M.

For each transition in sequence Seq_k, `horizontalMerge` decides whether a new insertion of this transition and corresponding states in M are necessary (Lines 31-33). Each new insertion of the transition is followed by the adjustment of the parameters associated with this transition (Lines 34-35, 43-45). Lines 36-38, 46-50 and 56-58 ensure that none of the negative examples is included into a specification model M.
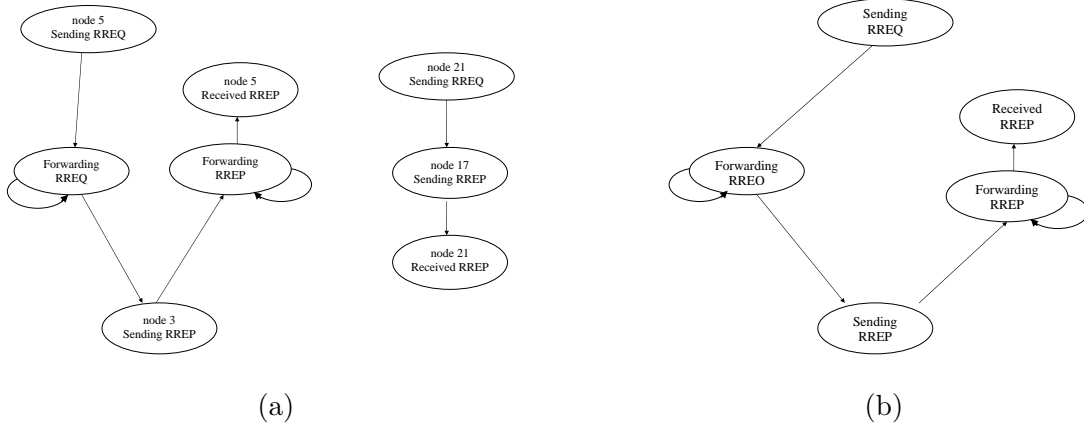
7

Figure 3: Example of `horizontalMerge` procedure: (a) before the procedure (b) after the procedure

For an example of `horizontalMerge` (Figure 3[1]), consider two generalized sequences

$$\left\langle \begin{array}{l} \texttt{RREQ(sent by node 5)} \rightarrow \texttt{RREQfwd(self-loop)} \rightarrow \\ \texttt{RREP(sent by node 3)} \rightarrow \texttt{RREPfwd(self-loop)} \rightarrow \\ \texttt{RREP(rcvd by node 5)}\,\text{and} \\ \texttt{RREQ(sent by node 21)} \rightarrow \texttt{RREP(sent by node 17)} \rightarrow \\ \texttt{RREP(rcvd by node 21)} \end{array} \right\rangle$$

in Figure 3(a). After performing *vertical merge* the resulting specification model is

$$\left\langle \begin{array}{l} \texttt{RREQsent} \rightarrow \texttt{RREQfwd (self-loop)} \rightarrow \texttt{RREPsent} \rightarrow \\ \texttt{RREPfwd(self-loop)} \rightarrow \texttt{RREPrcvd} \end{array} \right\rangle$$

as shown in the Figure 3(b).

4   Case Study

We demonstrate the validity of our approach on the example of The Ad hoc On-demand Distance Vector (AODV) and the Dynamic Source Routing (DSR) routing protocols.

AODV protocol   AODV is on-demand routing protocol which means it allows source node to establish routes only for nodes in active communication. The protocol uses two phases: *route discovery* that is done with route request (RREQ) and route reply (RREP) messages and *route maintenance* represented by route error (RERR). When a node desires to set up a route to a destination node, it broadcasts a RREQ message with a unique ID and its current sequence number. Node that receives a RREQ message might respond with route information (RREP) if it is itself is a destination or it has an active route to the sought destination node, otherwise RREQ hop count is increased by 1 and the message is broadcasted further. Duplicate RREQ message are dropped.

When node replies with a RREP message, destination sequence number is increased by 1, while source sequence number remains the same. The use of sequence numbers in AODV ensures a loop-free routes. RREP messages are unicasted back to the node originating RREQ.

If a link break is detected along the established routing path, route error (RERR) message is generated. More details on AODV protocol can be found in [26].

---

[1]Parameters associated with transitions are not shown for brevity.

DSR protocol Similar to AODV, DSR is protocol on-demand and establishes routes through route discovery and route maintenance phases [14]. Route discovery is initiated by a source node through a broadcast of RREQ identified by unique id. Each intermediate node alone the path of RREQ lists its address in the message, thus allowing node to determine the shortest paths to multiple destinations. Upon arrival of RREQ, destination node sends a RREP message to the route initiator with a copy of accumulated route record from RREQ. Route maintenance is done through RERR messages that are sent if broken link is detected [14].

Constructing specifications To construct specifications for these protocols we used traces of valid protocol behavior obtained through *ns2* tool [1]. The validity of specifications of routing protocols also depends on the attributes associated with each event in the route request-reply flow (each state in a sequence). To identify the parameters needed to distinguish valid behavior, different feature selection techniques [7] can be applied. In our experiments we used all parameters provided by *ns2* simulator. However, final specification model contains only a subset of those features, the rest was removed due to the irrelevancy and space constraints. Figure 4 presents the generated specifications of valid behavior for AODV and DSR protocols.

Generated AODV specification model contains seven states: `sending RREQ(RREQs)`, `forwarding RREQ(RREQfwd)`, `sending RREP(RREPs)`, `forwarding RREP(RREPfwd)`, `received RREP (RREPrec)`, `sending ERROR(RERRs)`,`Drop`. The model starts with a state `Sending RREQ` indicating the source node sending out a `RREQ` message. The transition from one state to another occurs only when the corresponding conditions specified in the text box are satisfied. As such when a `RREQ` reaches an intermediate node and the conditions specified in the text box titled `RREQs→ RREQfwd` are satisfied, the model enters state `Forwarding RREQ`; on the other hand, if the conditions specified in the text box titled `RREQs→RREPs` are satisfied, it enters a state `Sending RREQ`. From the state `Sending RREQ` the model has four valid transitions to the states: `Forwarding RREQ`,`Received RREP`, `Sending ERROR` and `Drop`.

DSR specification model follows a similar pattern with one additional state included: `Forwarding ERROR(RERRfwd)`. In this case, a transition `RERRs→ RERRfwd` indicates an event when an `RERR` message is forwarded to upstream nodes.

Shown specification model presents basic features of the protocols' behavior and thus can be easily understand and analyzed by humans. For comprehensive evaluation of the protocols more detailed specifications including features like packet salvaging, route shortening in DSR etc., can be constructed.

In addition to valid behavior, we have also considered possible attacks against AODV protocol based on the analysis by [22]. The analysis considered attacks in two dimensions: *atomic* misuses that present a manipulation of one routing message and *compound* misuses that are combinations of several atomic attacks. Since atomic attacks are essential events lying in the basis of any intrusive behavior in AODV [22], we focused on atomic misuses. Traces of AODV misuses were generated using simulation code provided by [21].

Examination of the misuse specifications showed that they largely follow the specifications of valid protocol behavior with minor deviations representing different misuses. For brevity we chose to introduce a *route disruption* attack.

Figure 5(a) presents a *route disruption* attack through a fake RREQ message. An attacker pretends to rebroadst a RREQ initiated from a destination node to the originating RREQ node with a fake source IP address in the IP header. Upon arrival of fake RREQ, target node will unicast a RREP message which will be dropped due to non-existent IP address. As a result the originating RREQ node will include non-existing IP address in the route to the destination node.

Thus a route will be broken [22]. While it is impossible to recognize a non-existing IP address without a full knowledge of nodes on the network, a fake RREQ message can be distinguished by monitoring each request-reply flow using the specification chart and the consistency of AODV features, specifically for this attack, source and destination IP addresses throughout request-reply flow. *Route disruption* can be also caused by a fake RREP message without the corresponding RREQ. An attacker unicasts a fake RREP with non-existing source IP address after forwarding an original RREQ. This would lead to the same results as in case with fake RREQ message. This misuse can be also detected through monitoring the consistency of flow messages each starting with the corresponding RREQ (Figure 5(b)).

5   Conclusion

In this paper we present an approach to learning specifications from the individual traces of program executions on the example of routing protocols. Based on ILP theory our approach constructs an abstract model of protocol behavior by generalizing the examples of route request-reply flows.

The constructed models of protocols' behavior describe the protocols' message exchange and the relationships among protocol-relevant attributes as reflected in the network traffic. While properties hidden from the network layer are not represented in these models, they can be incorporated into specifications using experts knowledge on system environment. We feel that approach brings significant benefits to intrusion detection field. As illustrated by the experiments, constructed specifications can be used in detecting protocol misuses. However, the approach is not designed as a stand-alone intrusion detection tool, although can be used to complement the existing intrusion detection systems.

One of the challenges in the specification-based approaches is to ensure the completeness and consistency of the developed specifications. Our proposed algorithm for synthesizing specifications provides a validation mechanism for ensuring these properties in the derived models. Nevertheless, a formal evaluation of generated specifications with regards to their completeness and consistency is one of the future avenues of our research work.

References

[1] The network simulator ns2. Online. http://www.isi.edu/nsnam/ns.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking 10^{20} states and beyond. In *Proceedings of LICS*, 1990.

[3] J. Chang and D. J. Richardson. Structural specification-based testing: automated support and experimental evaluation. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 285–302, London, UK, 1999. Springer-Verlag.

[4] Y. Chen, R. L. Probert, and D. P. Sims. Specification-based regression test selection with risk analysis. In *CASCON '02: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research*, page 1. IBM Press, 2002.

[5] T. Clausen, P. Jacquet, A. Laouiti, P. Muhlethaler, A. Qayyum and L. Viennot. Optimized link state routing protocol. *IEEE INMIC*, 2001.

[6] Y. Dong, X. Du, G. Holzmann, and S. A. Smolka. Fighting livelock in the i-Protocol: A case study in explicit-state model checking. *Software Tools for Technology Transfer*, 4(2), 2003.

[7] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

[8] Z. Haas. A new routing protocol for the reconfigurable wireless networks. *The IEEE Int. Conf. on Universal Personal Communications*, October 1997.

[9] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[10] M. R. A. Huth and M. D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[11] M. D. Jean-Philippe Pouzol. Formal specification of intrusion signatures and detection rules. In *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, 2002.

[12] S. P. Joglekar and S. R. Tate. Protomon: Embedded monitors for cryptographic protocol intrusion detection and prevention. *J. UCS*, 11(1):83–103, 2005.

[13] D. Johnson. Routing in ad hoc networks of mobile hosts. *The Workshop on Mobile Computing Systems and Applications*, pages 158–163, December 1994.

[14] D. B. Johnson, D. A. Maltz, and Y.-C. Hu. The dynamic source routing protocol for mobile ad hoc networks (dsr), 2004.

[15] C. Ko. Logic induction of valid behavior specifications for intrusion detection. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 2000.

[16] C. Ko, P. Brutch, J. Rowe, G. Tsafnat, and K. N. Levitt. System health and intrusion monitoring using a hierarchy of constraints. In *RAID 2001*, pages 190–204, 2001.

[17] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, 1994.

[18] C. Ko, M. Ruschitzka, and K. N. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *IEEE Symposium on Security and Privacy*, pages 175–187, 1997.

[19] A. Lazarevich, L. Ertoz, A. Ozgur, J. Srivastava, and V. Kumar. A comparative study of anomaly detection schemes in network intrusion detection. In *Proceedings of SIAM Conference on Data Mining*, 2003.

[20] S. H. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[21] P. Ning. Attacks against the aodv protocol. Online, 2005. http://discovery.csc.ncsu.edu/software/MisuseAODV.

[22] P. Ning and K. Sun. How to misuse AODV: A case study of insider attacks against mobile ad-hoc routing protocols. In *Ad Hoc Networks*, volume 3, pages 795–819, 2005.

[23] J.-M. Orset, B. Alcalde, and A. R. Cavalli. An efsm-based intrusion detection system for ad hoc networks. In *ATVA*, pages 400–413, 2005.

[24] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[25] C. Perkins and E. Royer. Ad hoc on-demand distance vector routing. *The 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[26] C. E. Perkins, E. M. Royer, and S. R. Das. Ad hoc on demand distance vector (AODV) routing. Routing IETF, Internet Draft, 2003.

[27] C. Ramakrishnan, I. Ramakrishnan, S. Smolka, et al. XMC: A logic-programming-based verification toolset. In *Proceedings of CAV*. Springer, 2000.

[28] R. Sekar, Y. Cai, and M. Segal. A specification-based approach for building survivable systems. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 338–347, 1998.

[29] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: a new approach for detecting network intrusions. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, 2002.

[30] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings 8th Usenix Security Symposium*, 1999.

[31] T. Song, J. Alves-Foss, C. Ko, C. Zhang, and K. Levitt. Using ACL2 to Verify Security Properties of Specification-based Intrusion Detection Systems. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2003)*, July 2003.

[32] C.-Y. Tseng, P. Balasubramanyam, C. Ko, R. Limprasittiporn, J. Rowe, and K. Levitt. A specification-based intrusion detection system for aodv. In *SASN '03: Proceedings of the 1st ACM workshop on Security of ad hoc and sensor networks*, pages 125–134, 2003.

[33] C.-Y. Tseng, T. Song, P. Balasubramanyam, C. Ko, and K. N. Levitt. A specification-based intrusion detection model for olsr. In *RAID '05:Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection*, pages 330–350, 2005.

[34] D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

Top diagram (AODV protocol):

```
RREQs->RREQfwd
HC: <
destSN: =
destIP: =
srcSN: =
srcIP: =
destIP(IP header)<> destIP
srcIP(IP header)=srcIP
```

```
RREQfwd -> RREQfwd
HC: <
destSN: =
destIP: =
srcSN: =
srcIP: =
destIP(IP header)<> destIP
srcIP(IP header)<>srcIP
```

```
RREQfwd -> RREQdrop
HC: <
destSN: =
destIP: =
srcSN: =
srcIP: =
```

```
RREQfwd -> RERRs
HC: <
destSN: =
destIP: =
srcSN: =
srcIP: =
```

```
RREQfwd -> RREPs
HC: >
destSN: <
destIP: =
srcSN: NA
srcIP: NA
```

```
RREQs -> RREPs
HC: <
destSN: <
destIP: =
srcSN: NA
srcIP: NA
srcIP(IP header)= destIP
```

```
Legend:
state1 -> state2
par: <
(explaination: par (state1) < par2 (state2))
HC – hop count
destIP – IP address of AODV destination node
destIP(IP header) - IP address of destination node in IP header
destSN – sequence number of destination node
srcIP – IP address of AODV source node
srcIP(IP header) - IP address of source node in IP header
srcSN – sequence number of source node
```

```
RREPs -> RREPrec
HC: >
destSN: <
destIP: =
srcIP(IP header)= destIP
```

```
RREPs -> RREPfwd
HC: <
destSN: =
destIP: =
srcIP(IP header)= destIP
```

```
RREPfwd->RREPrec
HC: =
destSN: =
destIP: =
```

```
RREPfwd -> RREPfwd
HC: <
destSN: =
destIP: =
srcIP(IP header)= destIP
```

```
RREPs -> RERRs
HC: <=
destSN: <=
destIP: =
```

```
RREPfwd -> Drop
HC: =
destSN: =
destIP: =
srcSN: =
srcIP: =
```

States: Sending RREQ, Forwarding RREQ, Sending RREQ, Received RREP, Forwarding RREP, Drop, Sending ERROR

Bottom diagram (DSR protocol):

```
RREQs -> RREQfwd
time: <
destIP: =
RREQ_SN: =
srcIP: =
RREPlength: =
numNodesTraversed: <
```

```
RREQfwd -> RREQfwd
time: <
destIP: =
RREQ_SN: =
srcIP: =
RREPlength: =
numNodesTraversed: <
```

```
RREQfwd -> RREPs
time: <
destIP: <>
RREQ_SN: =
srcIP: <>
RREPlength: <
numNodesTraversed: <
```

```
RREQs-> RREPs
destIP: <>
RREQ_SN: >
srcIP: <>
RREPlength: <
numNodesTraversed: <
```

```
Legend:
state1 -> state2
par: <
(explaination: par (state1) < par2 (state2))
destIP – IP address of DSR destination node
RREQ SN – sequence number of included in RREQ
srcIP – IP address of DSR source node
RREPlength – length of RREQ message
numNodesTraversed - number of nodes traversed
```

```
RREPs -> RREPrec
destIP: =
srcIP: =
RREPlength: =
```

```
RREPs -> Drop
time: <
destIP: =
RREQ_SN: =
srcIP: =
RREPlength: =
numNodesTraversed: =
```

```
RREPf -> Drop
time: <
destIP: =
RREQ_SN: =
srcIP: =
RREPlength: =
numNodesTraversed: =
```

```
RREPs -> RREPfwd
time: <
destIP: =
srcIP: =
RREPlength: =
numNodesTraversed: =
```

```
RREPfwd -> RREPrec
time: <
destIP: =
srcIP: =
RREPlength: =
numNodesTraversed: =
```

```
RREPfwd -> RREPfwd
time: <
destIP: =
srcIP: =
RREPlength: =
numNodesTraversed: =
```

```
Drop -> RERRs
time: <
destIP: =
srcIP: =
RREPlength: >
numNodesTraversed: <
```

```
RREPs -> RERRs
time: <
destIP: =
srcIP: =
RREPlength: >
numNodesTraversed:=
errorSrc: <>
errorDest: <>
```

```
RREPfwd -> RERRs
time: <
destIP: =
srcIP: =
RREPlength: =
numNodesTraversed:=
errorSrc: <>
errorDest: <>
```

```
RERRs -> RERRfwd
time: <
destIP: =
srcIP: =
RREPlength: =
numNodesTraversed:=
errorSrc: =
errorDest: =
```

States: Sending RREQ, Forwarding RREQ, Sending RREP, Received RREP, Forwarding RREP, Drop, Sending RERR, Forwarding RERR
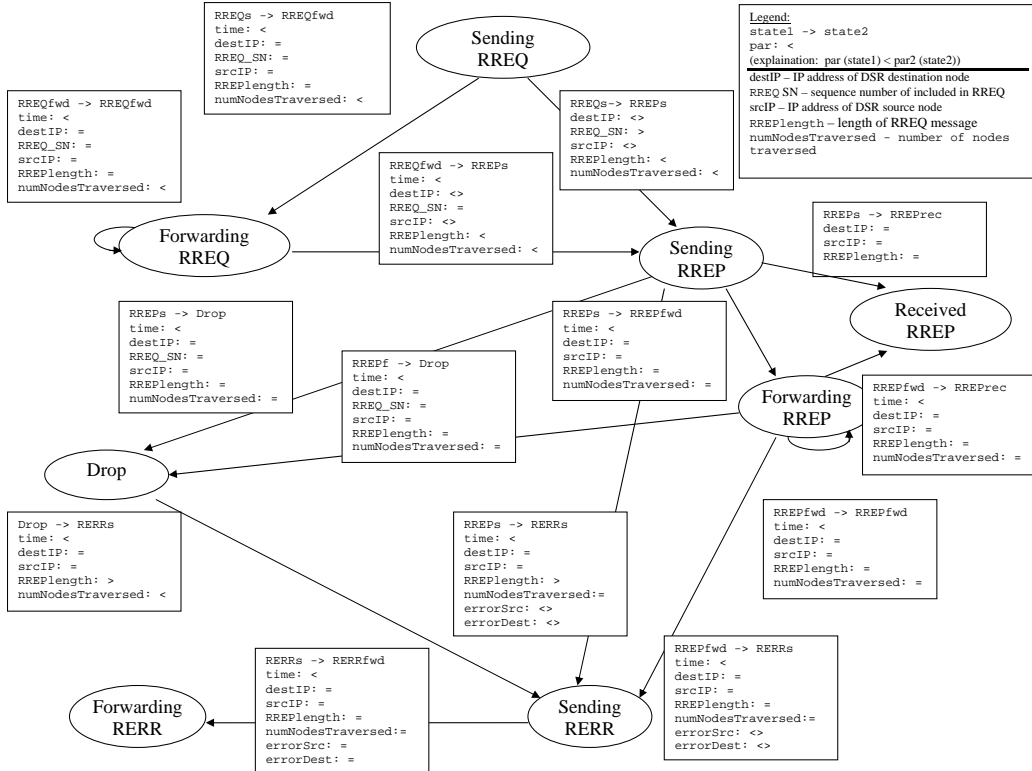
Figure 4: Valid behavior of AODV protocol(above), DSR protocol(below)

RREQ sent -> RREQ fwd
HC: <
pktID: =
destSN: =
destIP: =
srcSN: =
srcIP: =
time: <

Sending
RREQ

RREQ fwd->RREQ fwd
HC: <
pktID: =
destSN: =
destIP: =
srcSN: =
srcIP: =
time: <

Forwarding
RREQ

Drop
RREP

RREQ fwd->RREQ
rebroadcast
HC: >
pktID: =
destSN: =
destIP: <>
srcSN: <>
srcIP: <>
time: <

RREQ rebroadcast-> RREP
sent
HC: =
pktID: =
destSN: <
destIP: =
srcSN: NA
srcIP: NA
time: <

Sending
RREP

RREP sent-> RREP drop
HC: =
pktID: =
destSN: =
destIP: =
srcSN: NA
srcIP: NA
time: =

RREQs -> RREQfwd
HC: <
pktID: =
destSN: =
destIP: =
srcSN: =
srcIP: =
time: <

Sending
RREO

Forwarding
RREQ

RREQfwd-> RREPs
HC: >
pktID: =
destSN: <
destIP: =
srcSN: NA
srcIP: NA
time: <

Sending
RERR

RREPrec -> RERRs
HC: =
pktID: =
destSN: <
destIP: =
srcSN: NA
srcIP: NA
time: <

Received
RREP

RREPs-> RREPrec
HC: =
pktID: =
destSN: =
destIP: =
srcSN: NA
srcIP: NA
time: <

Sending
RREP

Legend:
state1 -> state2
par: <
(explaination: par (state1) < par2 (state2))
HC – hop count
pktID – unique identifier of packet
destIP – IP address of AODV destination node
destSN – sequence number of destination node
srcIP - – IP address of AODV source node
srcSN – sequence number of source node

(a)   (b)

Figure 5: (a)Route disruption attack (fake RREQ message) in AODV protocol (b)Route disruption attack (fake RREP message) in AODV protocol