BOOTSTRAPPING DENY-BY-DEFAULT ACCESS CONTROL FOR MOBILE AD-HOC NETWORKS

Honggang Zhang, Suffolk University, hzhang@ieee.org
Brian DeCleene, BAE Systems, brian.decleene@baesystems.com
Jim Kurose, Don Towsley, Univ. of Massachusetts Amherst, {kurose, towsley}@cs.umass.edu

ABSTRACT

We investigate the bootstrapping of policy-based access control in a deny-by-default mission-critical MANET. In the absence of any initial policies, a deny-by-default system fundamentally prevents all traffic flow. Providing all policies prior to deployment assumes advanced knowledge of all possible future scenarios – an assumption that is often unrealistic in practice; furthermore, policies may change over time. Thus, alternatively, network nodes can be initialized with a small set of initial policies (which we refer to as an axiomatic set of policies) that allow them to obtain additional policies, update outdated policies, and establish connectivity with neighboring nodes – a process that we refer to as bootstrapping. We identify a set of axiomatic policies for bootstrapping a deny-by-default system, propose a bootstrap protocol for neighbor link setup, and study how policies can be propagated within the MANET. Safety and liveness of the proposed bootstrap protocol are formally proved via model checking in SPIN. We also analyze the tradeoff between network vulnerability (the fraction of time that a node's policy is out-of-date) and the overhead incurred by different policy-dissemination approaches.

I. INTRODUCTION

In a deny-by-default system [3][6][13], policy rules explicitly define the various actions that system components are permitted to take. In the absence of a known, explicit policy allowing a given action, that action is implicitly disallowed, i.e., is *denied by default.* In a deny-by-default network, for example, policy may dictate which nodes can forward data, control, and policy traffic to which other nodes. A deny-by-default network architecture [14][2][1][16] contrasts sharply with today's Internet architecture, in which, for example, incoming datagrams are forwarded by default by Internet routers unless blocked by firewalls or ingress filters. A deny-by-default architecture provides enhanced security by limiting the set of possible actions to those that are explicitly allowed.

A crucial challenge in deny-by-default architectures is the manner in which network nodes obtain policy information. In the case of static policy, a node may be pre-configured with the entire set of policies it will need. This assumes advanced knowledge of all possible future scenarios and the set of policies that will be needed – an assumption that is often unrealistic in practice. An alternative is to provide nodes with just enough initial policy that they later can obtain (i.e., *bootstrap*) any additional policy that they may need. In the case that policy changes dynamically, nodes must be able to obtain additional new policy and refresh stored, out-of-date policy.

In the paper, we investigate the bootstrapping of policy-based access control in a deny-by-default mission critical MANET. We identify a small set of initial policies (which we refer to as an *axiomatic* set of policies) that allows a node to obtain additional policies, update outdated policies, and establish a set of capabilities that allow nodes to establish neighbor relationships amongst themselves in a manner that is consistent with policy. We propose a bootstrap protocol for neighbor link setup and prove its correctness (safety and liveness) properties using model checking. Additionally, in the case of dynamic policy changes, we study how policy updates can be propagated within the MANET. Using simple queuing models, we analyze the tradeoff between network vulnerability (the fraction of time that a node's policy is out-of-date) and the overhead incurred by different dynamic policy-dissemination approaches.

The rest of this paper is organized as follows. Related work is discussed in Section II. In Section III, we present preliminaries regarding policy in a deny-by-default network and identify an axiomatic set of policies needed to bootstrap neighbor relationships in such a network. In Section IV, we propose the bootstrap protocol for neighbor link setup and present the formal verification of safety and liveness of the protocol via SPIN [9]. Then in Section V, we analyze MANET vulnerability for three different policy distribution architectures, and insights/guidelines are provided to nodes in MANET to

This material is based upon work under a subcontract #069153 issued by BAE Systems National Security Solutions, Inc. and supported by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare System Center (SPAWARSYSCEN), San Diego under Contract No. N66001-08-C-2013.

978-1-4244-2677-5/08/$25.00 ©2008 IEEE

meet desired levels of security and performance in the bootstrapping process. The paper concludes in Section VI.

II. RELATED WORK

An off-by-default, IP-layer access control protocol is proposed in [2] to protect the Internet from malicious traffic. [16] proposes an architecture based on capabilities for limiting the impact of DoS attacks on the Internet. DoS attacks are limited by requiring that traffic sent to a receiver must first be explicitly allowed by that receiver; a network-layer connection is needed for obtaining this permission first. [1] argues against using capabilities, as they are themselves susceptible to denial-of-capability attacks. Our work differs from these works in that we study a policy-based deny-by-default access control for MANETs and the process for bootstrapping this system. The STRONGMAN architecture [12] is a policy-based access control system, but not based on a deny-by-default principle. Regarding policy-based MANETs, DRAMA [7] studies policy management, and [15] studies policy-based interactions (focusing on policy specification) in dynamic wireless networks. None of these studies address the bootstrapping problem. [10] proposes three approaches to policy dissemination. Our work on policy dissemination (once neighbor relationships have been established and routing is in place), differs from previous works in that policy dissemination itself is subject to policy, and our analysis quantifies the tradeoff between vulnerability and performance in dissemination. These results can, in turn, be used for policy configuring and management. IPsec [11] allows establishing a security relationship among nodes, but not for the purpose of policy. To bootstrap security associations for routing in MANETs, [18] presents a technique (independent of any trusted security service) to implement a secure binding between an IP address and a key. [19] discusses various problems in bootstrapping coalition MANETs, in order to identify novel solution methodologies.

III. PRELIMINARIES FOR NETWORK POLICY

In this section, we briefly introduce the definitions of network policies for a deny-by-default network and identify an axiomatic set of policies used in the bootstrap protocol outlined in the following section. Note that the focus of this paper is not on policy specification nor the enforcement of policy within the data plane; instead we focus on the process of signaling between neighboring nodes, in a manner that is consistent with policy, to establish neighbor relationships. Once neighbor relationships have been established, routing and other higher-level network functions can then be put in place.

We define a policy rule as consisting of four components: *the Proposition (P), the Policy Duration, the Policy Refresh Interval, and the Policy Source (S). Proposition (P)* is a conditional test on an element such as a data message, internal node state, or identity (of the node itself or a neighbor) and evaluates to either TRUE or FALSE. For example, a *P* may be written as *(Node=Alice) & (BeaconSrc=Bob)*, where *Node* is the node at which the proposition is being tested, and *BeaconSrc* stands for the beacon-sending node. The *Policy Duration* ($[t_s; t_e]$) specifies the start and end time of a policy. The *Policy Refresh Interval* (t_r) specifies an optional maximum duration that a policy is allowed to be considered viable without receiving confirmation that the policy has not changed. The *Source (S)* policy component indicates the originator of the policy. For ease of exposition, we will not consider policy duration, the policy refresh interval or the policy source when considering the bootstrap protocol; see [17] for the more general case; we will consider these aspects of policy in Section V when we study policy propagation.

Since our focus is on the bootstrap process, we assume that policies are consistent, and that a mechanism (e.g., [12]) exists for determining the integrity, validity and non-repudiation of any policy that a node receives.

Axiomatic Policies. Axiomatic policies are defined as the initial set of policies that are installed *a priori* and allow a deny-by-default system to establish neighbor relationships. The key challenge here will be to define a small set of primitive policy elements (e.g., whether a node is allowed to advertise its existence via beacon messages, whether it can receive additional policy from a given neighbor). Once neighbor relations have been established, higher-level functionality, such as routing can be established. Axiomatic policies do not preclude a node from being initialized with additional policies for performance reasons. The existence and size of this set of axiomatic policies are key measures of the practicality of deny-by-default security. We conjecture the following policies form the axiomatic set:

- *PolicySource* policy, a rule that defines which nodes are authorized sources of new policies and policy updates/refresh. We assume that all nodes are preloaded with a set of trusted policy servers, and our policy integrity assumption prevents false policy insertion/edits.
- *MayBeacon* policy specifies whether a node is allowed to advertise its existence.
- *MayShare* policy of a node with respect to another node specifies whether this node may share a policy with the other node.

- *PolicyAccept* policy of a node specifies the set of nodes from which this node may accept policy updates. Note that our assumption of policy consistency assures that nodes having the same policies arrive at common policy state.

We next demonstrate that these policies enable any two nodes to set up a neighbor link via our proposed bootstrap protocol.

IV BOOTSTRAPPING A NEIGHBOR LINK

In this section, we propose a bootstrap protocol that allows two nodes to establish a neighbor relationship when that relationship, and the process for bootstrapping that relationship, is consistent with policy. The bootstrap protocol only relies on the axiomatic policies defined in the previous section. When two nodes "establish a neighbor relationship", we mean that the final state of both nodes is that they have each agreed that they can send and receive messages to/from each other.

Although the axiomatic policies are required to bootstrap the system, they do not define whether a particular neighbor relationship is allowed. Therefore, a node will typically also be initialized with policy that defines a set of one or more nodes with whom it is allowed to run the bootstrap protocol. If this initial set is empty, a node will be unable to join the network or obtain additional policy unless there is a path between this node and a policy server. If the set is non-empty, the node may run the bootstrap protocol with these nodes, possibly obtaining new policies that would then allow it run the bootstrap protocol with yet other nodes. The *OkNeighbor* policy defines whether a particular neighbor relationship is allowed. The *OkNeighbor* policy of node *A* with respect to node *B* is TRUE if *B* is allowed to be a neighbor of *A*. As will be illustrated, this policy is not axiomatic; it can be obtained by running the bootstrap protocol.

In our proposed bootstrap protocol, two nodes that encounter each other can exchange *OkNeighbor* policies (if allowed by their axiomatic policies *MayShare*) even without establishing a neighbor relationship; if their *OkNeighbor* policy with respect to each other are both TRUE, then they are able to additionally establish a neighbor link between themselves.

A. Bootstrap Protocol

We now formally describe the bootstrap protocol. Consider two nodes meeting in a MANET, that are situated such that each can hear the other's transmissions. Without loss of generality, let *T* denote the node under study, and *X* denote the other node just encountered. Let us now consider *T's* bootstrap with *X* (symmetrically, *X* goes through the same process).

We use a Finite State Machine (FSM) [5] to describe the internal state transitions of node *T*. As shown in Figure 1, there are four internal node states: *active beacon (A), passive (P), half-open (H), and full-open (F)*. If a node has no *OkNeighbor* policy or an undefined *OkNeighbor* policy with respect to the other node, it can acquire this policy through message exchange with the other node (if carrying the policy), even though these two nodes have yet to establish a neighbor relationship. When a node has a valid and affirmative *OkNeighbor* (evaluates to TRUE) policy with respect to the other node, it starts its neighbor link setup process with the other node. The notations and terms used in the bootstrap protocol and FSM are given in Table I and Figure 2. The actions and transitions in each state of node *T* are described as follows.

TABLE I NOTATIONS FOR BOOTSTRAP PROTOCOL

Notations for states in Finite State Machine	
T	this node
X	the other node
X→T	if in state, means this node knows *X* has this node as neighbor
T→X	if in state, means this node has *X* as its neighbor
A	active beacon state
P	passive listening state
F	full-open connection state
H	half-open connection state
Messages (note, policy can be attached to *Bcn* or *Nbr*, and policy request (*NP*) can be attached to *Bcn*)	
Bcn(X)	existence beacon sent by node *X*
Nbr(T→X)	neighbor setup request message from *T* to *X*
NbrA(T→X)	ACK to *Nbr(X→T)*
NP(T→X)	*T* requests policy from *X*
Policy(T)	policy for node *T*
Other notations used in state diagram for protocol	
NP	need policy flag, either 0 or 1
SP	share policy flag, either 0 or 1
NP=1	if in state, means the next sent message should contain policy request; if in message, means the message asks for policy
SP=1	if in state, means the next sent message should contain policy for the other node; if in message, means the message contains policy
Bcn message	any message has *Bcn*

Active Beacon state (A).

- There are three actions in state A: send a beacon and after each beacon, clear the NP and SP flags. Note that if *NP* is set to *1*, then the beacon sent will contain a policy request; if *SP* is set to *1*, the beacon sent will contain policy for the other node.
- When *T* sends a beacon, it remains in state A if nothing else is changed.
- When *T* receives any message from node *X* and *T* has no or undefined *OkNeighbor* policy with respect to *X*, the following actions are taken: (i) if received message has policy for *T*, verify it and make a decision to accept it or not based on its axiomatic *PolicyAccept* policy; (ii) *T* sets its internal *NP* flag to 1 if its *OkNeighbor* policy remains undefined (not received in the previous step); (iii) if the

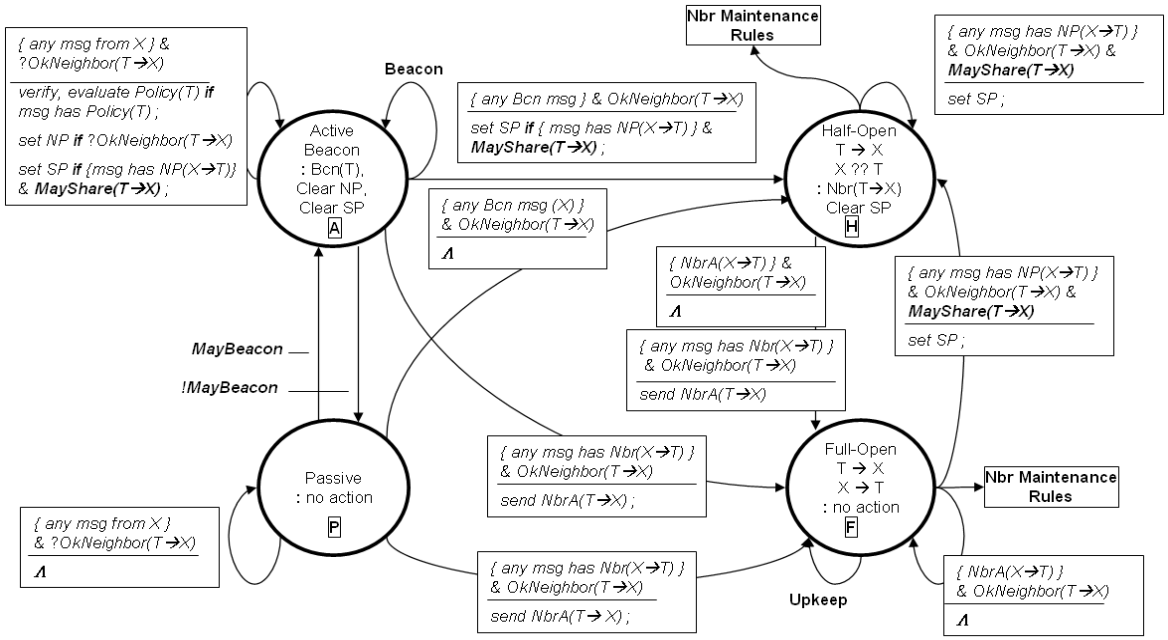

Figure 1. Finite State Machine for Bootstrap Protocol

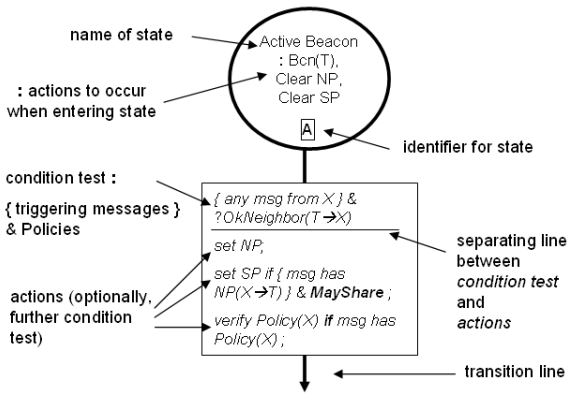

Figure 2. FSM Notation

TABLE II

NEIGHBOR MAINTENANCE RULES FOR A NODE

Notations for states in state diagram	
Nbr timeout & MayBeacon is TRUE	enter state A
Nbr timeout & MayBeacon is FALSE	enter state P
OkNeighbor is FALSE & MayBeacon is TRUE	enter state A
OkNeighbor is FALSE & MayBeacon is FALSE	enter state P
OkNeighbor undefined & MayBeacon is TRUE	set NP flag, enter state A
OkNeighbor undefined & MayBeacon is FALSE	enter state P

Comments: *Nbr timeout* means if this node does not hear from the other node in a pre-specified time interval, then this node thinks that the link with the other node is broken. *OKNeighbor* becomes undefined or invalid once it is expired.

received message also contains a policy request and *T*'s axiomatic policy *MayShare* with respect to *X* evaluates to TRUE, then set the *SP* flag, which will cause the next sent beacon message to contain policy for *X*; (iv) return back to state A.

- Three transitions leave state A. If *T* receives any beacon messages from *X* and *T*'s *OkNeighbor* policy evaluates to TRUE, it enters half-open state H and sets the *SP* flag if the received message contains a policy request and if its *MayShare* evaluates to TRUE. If *T* receives a message containing a neighbor setup request and its *OkNeighbor* policy evaluates to TRUE, then it sends an ACK and enters the full-open state F. If *T*'s axiomatic policy *MayBeacon* becomes FALSE, then it enters passive state P.

Passive state (P)

- There are three transitions out of this state. If *T*'s *MayBeacon* policy becomes TRUE, it enters state A. If *T* receives any beacon from *X* and its *OkNeighbor* policy evaluates to TRUE, it enters the half-open state H. If *T* receives any message containing a neighbor setup request and its *OkNeighbor* policy evaluates to TRUE, then it sends out an ACK message *NbrA* and enters the full-open state F.
- If *T* does not have valid *OkNeighbor* policy with respect to node *X*, *T* ignores all messages from *X*.

Half-open state (H)

- When entering this state, *T* sends out a neighbor setup request to node *X* and clears the *SP* flag.
- If *T* receives any message containing policy request from *X* and *T* is allowed to be a neighbor of *X* and *T*'s *MayShare* policy with respect to *X* evaluates to TRUE, then *T* sets its *SP* flag and returns back to the half-open state H. Note that since *SP* is set, its next neighbor setup request message should be attached with policy for *X*.
- If *T* receives any message containing a neighbor setup request and its *OkNeighbor* policy evaluates to TRUE, it sends out an ACK message, *NbrA,* and enters the full-open state, F. If *T* receives an ACK to its previously sent neighbor setup request and its *OkNeighbor* evaluates to TRUE, then it enters full-open state. There are a number of neighbor maintenance rules that can lead node *T* to other states. They are given in Table II. They deal with

various cases where either the neighbor link setup request message timeouts, or *OkNeighbor* policy expires, or its *MayBeacon* policy with respect to the other node becomes FALSE. In all these cases, *T* enters either passive or active beacon states to restart the neighbor link establishment process.

Full-open state (F)

- In this state, node *T* periodically exchanges messages with node *X* for upkeep purposes. *T* stays in this state if it receives an ACK to its neighbor link setup request and its *OkNeighbor* policy with respect to the other node is still TRUE.
- Similar to the half-open state, this node also follows the neighbor maintenance rules shown in Table II that will lead this node out of state F.

Example. We now illustrate the operation of our bootstrap protocol. Consider two nodes *A* and *B* that encounter each other and are part of a larger MANET. Both *A* and *B* are preconfigured with *MayBeacon* equal to TRUE. Node *A* carries *B*'s valid *OkNeighbor* with respect to *A*, and vice versa. But neither one has its own valid *OkNeighbor* with respect to the other node. Thus, in order to set up a neighbor relationship, they need to first receive their *OkNeighbor* policies from each other. Figure 3 shows the neighbor relationship establishment process, which executes as follows.

1. Node *A* actively beacons. Its beacon is received by *B*. Since *B* has no or invalid *OkNeighbor* policy with respect to *A* and *B*'s *MayBeacon* is TRUE, *B* sends a beacon with policy request (the NP flag set to 1) to *A*.
2. When *A* receives *B*'*s* beacon, it finds that it can share policy with *B* but does not have an *OkNeighbor* policy with respect to *B*. *A* thus sends a beacon to *B,* with its own policy request and a policy for *B*.
3. When *B* receives its policy from *A*, it verifies, accepts, and evaluates the policy. The evaluation of the received policy sets *B*'s *OkNeighbor* policy (with respect to *A*) to TRUE. Since *B*'s *MayShare* with respect to *A* is TRUE, it responds to *A*'s policy request by attaching a policy for *A* to its next beacon sent to *A*.
4. When *A* receives its policy from *B*, it verifies, accepts, and evaluates the policy. The evaluation of the received policy sets *A*'s *OkNeighbor* policy (with respect to *B*) to TRUE. *A* remains in active beacon state, and sends a beacon to *B*.
5. Since *B*'s *OkNeighbor* with respect to *A* is TRUE now, *B* sends a neighbor link setup request to *A* when it receives a beacon from *A*, and then *B* enters the half-open state.
6. Since *A*'s *OkNeighbor* with respect to *B* is TRUE now, *A* accepts *B*'s neighbor link setup request, sends a neighbor ACK message (*NbrA*) to *B*, and enters full-open state.
7. *B* enters the full-open state when it receives *NbrA* from *A*. The neighbor relationship of *A* and *B* is now established.

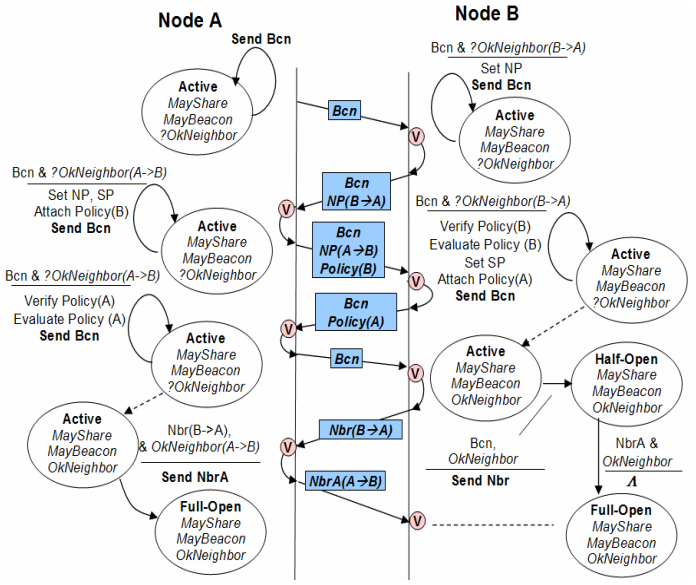

Figure 3. Sample path of bootstrap protocol execution. V stands for the verification of received message.

B. Formal verification of bootstrap protocol

In order to verify the correctness of our bootstrap protocol, we implemented it in SPIN [9], a powerful model checker for formal verification of distributed software systems. We built a formal model of our protocol using the PROMELA (PROcess MEta LAnguage) language [9]. In order to verify the correctness of a protocol, SPIN exhaustively searches the entire state space of communicating nodes and message channels. We allow message errors or loss in transfer in a deterministic manner. In Figure 4, we show representative SPIN code[1] for a loop-back transition (the top left-most transition shown in Figure 1). This piece of code shows that once a node (with missing or invalid *OkNeighbor* policy) receives a message, it performs a sequence of actions (with conditional tests) and returns to its initial state. Depending on the values of the *NP* and *SP* flags, a node may attach a policy request or policy to a message, indicated by message variable name containing *PolicyReq* or *Policy* in Figure 4.

We have formally verified the correctness (safety and liveness) of our bootstrap protocol. Specifically, for safety, we have shown that the protocol is deadlock free, and that if the *OkNeighbor* policy of either one of the two nodes evaluates to FALSE, then the nodes will never be able to set up a neighbor relationship. Regarding liveness, we have proved the following claims. First, if at least one of two nodes do not have a valid *OkNeighbor* policy with respect to the other node, and they are not connected via neighbor link, and both nodes can actively beacon, and they have affirmative

[1] Complete SPIN code is given in technical report [17].

MayShare policy with respect to each other, and they are allowed to accept policy from each other, then eventually they both can receive their *OkNeighbor* policies even without a neighbor relationship. Second, if the *OkNeighbor* policies of both nodes evaluate to TRUE with respect to each other, and at least one of them can actively beacon, then eventually they will establish a neighbor relationship with each other.

```
131 active_bcn_state:
132   msg_rcv( ); /* always listening */
133
134   if
135   :: PolicyOkNbr==0 && PolicyMayBcn==2 && anymsg==1 ->
136     if
137     ::polmsg==1 ->
138       atomic {
139         if
140         :: policy_ok_nbr_flag==1 -> PolicyOkNbr=2
141         :: policy_ok_nbr_flag==0 -> PolicyOkNbr=1
142         fi;
143         polmsg=0;
144       }
145     :: else
146     fi;
147     if
148     :: PolicyOkNbr==0 -> NP=1
149     :: else
150     fi;
151     if
152     :: reqmsg==1 && PolicyMayShare==2 -> SP=1
153     :: else
154     fi;
155     if
156     :: NP==1 && SP==1 -> mysend(out, bcn_PolicyReq_Policy)
157     :: NP==1 && SP==0 -> mysend(out, bcn_PolicyReq)
158     :: NP==0 && SP==1 -> mysend(out, bcn_Policy)
159     :: else -> mysend(out, bcn);
160     fi;
161     SP=0; NP=0;
162     atomic {
163       anymsg=0;
164       reqmsg=0;
165     }
166     goto active_bcn_state; /* stay in active_bcn_state */
```

Figure 4. Sample SPIN code written in PROMELA for modeling the protocol. This piece of code describes a loop-back transition of active beacon state of a node. Variables prefixed with *bcn* represent beacon messages. Each policy variable (prefixed with *Policy*) can take three values: *0* means no such policy; *1* means the policy evaluates to FALSE; *2* means the policy evaluates to TRUE. For instance, *PolicyOkNbr=0* means that this node does not have valid *OkNeighbor* with respect to the other node. *PolicyMayBcn=2* means that this node can actively beacon. "*mysend*" is a procedure for sending messages in wireless channel, and "*msg_rcv*" is a procedure for receiving messages from wireless channel. We use bit variables *anymsg, reqmsg,* and *polmsg* to indicate whether any message has been received, the received message contains policy request, and the received message contains policy, respectively. We use bit variable *policy_ok_nbr_flag* to indicate whether the received and accepted *OkNeighbor* policy evaluates to TRUE or not. Note that this variable represents a decision made by some other system-wide component, not by the bootstrap protocol.

V. POLICY PROPAGATION

Once nodes have established a neighbor relationship via our bootstrap protocol, they may receive updated policies that have been propagated through the network. A deny-by-default network is completely bootstrapped once this propagation process is complete. Here we assume that routing protocols are in place and focus on the manner in which policy updates are propagated. Before a node receives the updated policy, it is considered vulnerable. We define *vulnerability* as the expected fraction of time that a node is vulnerable. This value is used as a security metric for evaluating policy propagation. We define the *signaling overhead* as the fraction of time that the network is used for policy propagation – during which the channel cannot be used to transfer other data. It is used as a performance metric. A longer policy propagation time means a longer vulnerable period. We assume that the policy propagation time is given (a function of underlying routing protocol, mobility, etc), and focus on evaluating three basic policy dissemination architectures. Our analysis can help designers of access control to set the policy refresh interval t_r and helps a node to determine how frequently it should pull policies from servers in order to meet certain levels of security and performance.

The three basic policy dissemination architectures are *periodic refresh only, pull,* and *push*. In *periodic refresh only*, a policy server periodically broadcasts policies at interval t_r, regardless whether policies are updated or not. In *pull*, nodes actively and periodically request policies from policy servers. In *push*, a policy server pushes policy to other nodes whenever an update (or change) is available. Intuitively, nodes are less vulnerable if policies are more frequently refreshed through the network, but more policy beacons in the network leads to higher signaling overhead and less available network capacity for other data transfer. A security-concerned user might be inclined to pull actively while sacrificing some performance, whereas a performance-concerned user might simply rely on the default periodic refresh to receive policies. Our analysis below provides insights into this tradeoff and presents design guidelines.

We have conducted analysis via simple queuing models to quantify the tradeoff between network vulnerability and the overhead incurred by the three dynamic policy-dissemination approaches. We solve for vulnerability and signaling overhead, under the assumption that all time intervals (such as policy refresh from server, policy update, and policy pull from nodes) follow some given probability distributions or empirical distributions, and allow policy messages to be randomly lost in the network. For details, see [17].

We now present some numerical evaluation results based on the analysis given in [17]. First, we fix the average policy change (or update) interval, and fix the policy fetching time (or propagation time) K_f at one thousandth of policy change interval. We vary the policy

refresh interval in the *periodic refresh only* architecture. In the *pull* architecture, we set the number of pulls per policy refresh interval to be 10. Results are shown in Figure 5. We can make the following observations: 1) vulnerability decreases with increased signaling overhead, so clearly there is a security vs. signaling tradeoff; 2) the *pull* architecture provides significantly decreased vulnerability over *periodic refresh only* with a ten-fold signaling overhead increase; 3) the *push* architecture's performance depends only on the policy change rate, and provides a vulnerability lower bound.

Next, we let the policy fetching time, K_f, be exponentially distributed and let nodes actively pull policies in each refresh interval. Our results in [17] show that a longer average fetching time leads to a higher vulnerability and larger overhead, but in general, *pull* achieves lower vulnerability and overhead. In addition, we consider that policy message can be lost in the network, and that the policy server re-sends policy message after a certain timeout when loss occurs. Our results in [17] show that a larger loss probability leads to a higher vulnerability and larger overhead, and in general *pull* achieves lower vulnerability and overhead. We also observe that low loss probabilities (less than *0.2* in our case) do not significantly affect vulnerability and signaling overhead.

In sum, our vulnerability analysis of policy propagation in the bootstrapping process quantifies the tradeoff between security and performance, and shows that a good choice of pull frequency can effectively achieve low vulnerability and high performance. More generally, our analysis provides a technique for policy servers and users to achieve their desired levels security and performance.

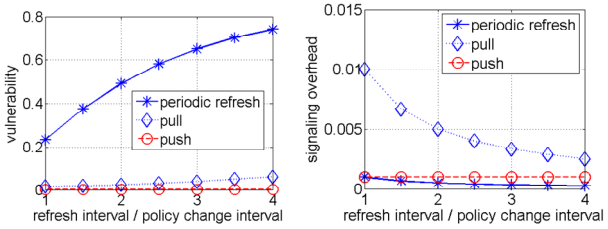

Figure 5. (a) expected vulnerability when K_f is fixed. (b) expected signaling overhead when K_f is fixed.

VI. CONCLUSION AND FUTURE WORK

To address the challenges of bootstrapping deny-by-default access control in MANETs, we have identified a set of axiomatic policies and proposed a bootstrap protocol for establishing secure neighbor relationships among nodes. The safety and liveness of the protocol were formally verified via model checking. We have also analyzed three basic policy dissemination architectures in the bootstrapping process and quantified the tradeoff between security and performance. Our current bootstrap protocol is designed for establishing a neighbor link by using axiomatic policies. As a future work, we will expand this protocol to safely and efficiently bootstrap the whole network, and to control policy propagation. We will implement and experiment with our bootstrap protocol in real deny-by-default MANETs. We will apply and verify our vulnerability analysis in practice and with empirical data.

REFERENCES

[1] K. Argyraki and D. Cheriton, "Network capabilities: The good, the bad and the ugly," in *Proc. of Fourth Workshop on Hot Topics in Networks (HotNets-IV)*, November 2005.

[2] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker. "Off by default!" in *Proc. of Fourth Workshop on Hot Topics in Networks (HotNets-IV)*, November 2005.

[3] M. Bauer, "Paranoid penguin: Introduction to selinux, part ii", *Linux Journal*, vol. 155, 2007.

[4] S. Bhatt, S. R. Rajagopalan, and P. Rao, "Federated security management for dynamic coalitions," in *DARPA Information Survivability Conference and Exposition*, 2003.

[5] G. Bochmann and C. Sunshine, "Formal methods in communication protocol design," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 624– 631, 1980.

[6] S. Bratus, A. Ferguson, D. McIlroy, and S. Smith, "Pastures: Towards usable security policy engineering," in *2nd International Conference on Availability, Reliability and Security*, 2007.

[7] C.-Y. J. Chiang, S. Demers, P. Gopalakrishnan, L. Kant, A. Poylisher, Y.-H. Cheng, R. Chadha, G. Levin, S. Li, Y. Ling, S. Newman, L. LaVergne, and R. Lo, "Performance analysis of drama: a distributed policy-based system for manet management," in *IEEE MILCOM*, 2006.

[8] J. Clark, J. Murdoch, J. McDermid, S. Sen, H. Chivers, O. Worthington, and P. Rohatgi, "Threat modelling for mobile ad hoc and sensor networks," in *Annual Conference of ITA*, 2007.

[9] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003.

[10] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. "Implementing a distributed firewall," in *ACM Conference on Computer and Communications Security*, 2000.

[11] S. Kent and K. Seo, *IETF RFC 4301: Security architecture for the internet protocol*. 2005.

[12] A. Keromytis, S. Ioannidis, M. Greenwald, and J. Smith, "The strongman architecture," in *3rd DARPA Information Survivability Conference and Exposition*, 2003.

[13] H. Peine, "Rules of thumb for developing secure software: Analyzing and consolidating two proposed sets of rules," in *3rd Int. Conf. on Availability, Reliability and Security*, 2008.

[14] T. Wolf, "Design of a network architecture with inherent data path security," in *3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, 2007.

[15] H. Wong, C.-K. Chau, J. Crowcroft, and K.-W. Lee, "How to enable policy-based interactions in dynamic wieless networks?" in *IEEE Workshop on Policies for Distributed Systems and Networks*, 2008.

[16] X. Yang, D. Wetherall, and T. Anderson, "A DOS-limiting network architecture," in *ACM SIGCOMM*, 2005.

[17] H. Zhang, B. DeCleene, J. Kurose, and D. Towsely, "Vulnerability analysis of policy bootstrapping," in *Tec. Report,* UMass. Amherst, 2008, ftp://gaia.cs.umass.edu/pub/Zhang08_vulnerability_analysis.pdf.

[18] R.B. Bobba, L. Eschenauer, V. Gligor, and W. Arbaugh, "Bootstrapping security associations for routing in mobile ad-hoc networks", in *IEEE Globecom*, 2003.

[19] M. Srivatsa, D. Agrawal and S. Balfe, "Bootstrapping Coalition MANETs" in *ITA Technical Report, February 2008.*