A Policy Enforcing Mechanism for Trusted Ad Hoc Networks

Gang Xu, *Member, IEEE,* Cristian Borcea *Member, IEEE,* and Liviu Iftode *Senior Member, IEEE*

Abstract

To ensure fair and secure communication in Mobile Ad hoc Networks (MANETs), the applications running in these networks must be regulated by proper communication policies. However, enforcing policies in MANETs is challenging because they lack the infrastructure and trusted entities encountered in traditional distributed systems. This paper presents the design and implementation of a policy enforcing mechanism based on Satem, a kernel-level trusted execution monitor built on top of the Trusted Platform Module. Under this mechanism, each application or protocol has an associated policy. Two instances of an application running on different nodes may engage in communication only if these nodes enforce the same set of policies for both the application and the underlying protocols used by the application. In this way, nodes can form trusted application-centric networks. Before allowing a node to join such a network, Satem verifies its trustworthiness of enforcing the required set of policies. Furthermore, Satem protects the policies and the software enforcing these policies from being tampered with. If any of them is compromised, Satem disconnects the node from the network. We demonstrate the correctness of our solution through security analysis, and its low overhead through performance evaluation of two MANET applications.

Index Terms: Trusted computing, ad hoc networks, mobile computing

I. INTRODUCTION

With the maturity of short-range wireless technologies and proliferation of mobile computing devices, building real-life applications over mobile ad hoc networks (MANET) becomes feasible. For instance, two potential applications are traffic monitoring in vehicular networks and peer-to-peer file sharing in ad hoc networks of smart phones. A key to the success of such applications is a mechanism assuring secure communication and proper collaboration among all participant entities. To achieve this goal, communication policies

Gang Xu is with the Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019. He is also with AT&T, 200 Laurel Ave, Middletown, NJ 07748. E-mail: gxu@cs.rutgers.edu.

Cristian Borcea is with the Department of Computer Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102. E-mail: borcea@cs.njit.edu.

Liviu Iftode is with the Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019. E-mail: iftode@cs.rutgers.edu.

that govern the interactions between entities must be defined and enforced. For instance, in a traffic monitoring application, the policy can guarantee that a car always forwards accident alerts to cars coming behind it. Similarly, in a peer-to-peer application, the policy can guarantee that a smart phone can post a query only if it has made several contributions such as publishing files or forwarding other queries.

Mechanisms to define and evaluate security policies have been well studied in traditional distributed system [1], [2]. While these methods provide sufficient expressive power to represent policies for MANET applications, the challenge is how to enforce such policies in MANETs. Most of the existing policy enforcement solutions have focused on Internet-based systems [3], [4], [5], [6]. Unfortunately, these solutions are not fit for MANET for two reasons. First, they enforce policies on trusted "choke points" (e.g., firewall or proxy), which do not exist in MANETs due to the lack of infrastructure. Furthermore, determining where to place a choke point in a MANET is almost impossible because the paths between nodes change frequently due to mobility [7]. Second, existing methods aim to protect the servers from unauthorized client accesses. In MANET, this distinction does not exist as every node can be a server and a client at the same time, and no entity can be trusted more than another.

A potential solution for such a peer-to-peer environment is Law-Governed Interaction (LGI) [8], [9]. LGI governs the communication between all nodes in the network by enforcing a unified group policy on a set of middleware controllers. However, LGI requires the controllers to be trusted, but does not provide means of establishing the trust. Consequently, in practice, it can only be applicable in controlled environments where the enforcers can be deployed or elected, such as corporate intranet [10], [6] and Internet P2P [11]. McCune et al [12] advanced another step by developing a shared trusted reference monitor (Shamon) across a coalition of nodes using remote attestation. Shamon enforces communication policies at the virtual machine level and requires that each node runs multiple virtual machines (one for each application), which may not be practical for mobile devices. Additionally, Shamon does not provide enough flexibility to compose applications and policies. If an application depends on others, then all of them together with their policies must be isolated in one virtual machine.

Different than enforcing policies in the network, another approach is to allow only nodes owned by trusted principals to participate in the network [13]. The method does not address the case of anonymous nodes spontaneously establishing MANETs. Furthermore, such methods provide insufficient level of security because a known-to-be-trusted node is more likely to be compromised and taken over by an attacker in MANETs than in infrastructure-based networks, due to the lack of physical protection.

This paper presents the design and implementation of a policy enforcing mechanism based on a kernel-level trusted execution monitor. Under this mechanism, each MANET application or protocol has its own policy [1]. All nodes supporting a certain application and enforcing its policy form a trusted application-centric network. Since an application may depend on other applications, our policy enforcing mechanism creates a trusted multi-tier network. The member nodes in such a network must enforce the policies associated

---

[1]In the rest of the paper, we will use the terms application and protocol interchangeably to denote a piece of software regulated by its own policy.

with these applications as well. For instance, a peer-to-peer file sharing application may depend on an on-demand routing protocol. In this case, the mechanism creates a two-tier trusted file sharing network. It first establishes a trusted routing tier, and hence, a trusted network for routing, comprising of all the nodes that enforce the routing policy. On top of this tier, it then creates a file sharing tier, enforcing the file sharing policy.

In our policy enforcing mechanism, nodes can be members of multiple multi-tier networks simultaneously. For example, let us consider that a vehicular traffic monitoring application uses the same routing algorithm with the file sharing application. Nodes in the aforementioned file sharing network can also establish a traffic monitoring network by creating, on top of the routing tier, a separate trusted tier enforcing the traffic monitoring policy. Two nodes may communicate through an application if and only if they enforce the same application tier policy and all the underlying tier policies.

Our policy enforcing mechanism allows each node to uniformly enforce the policies without assuming any prior trust with other nodes. This is similar to the method of building trusted ad hoc network we developed previously [14]. To ensure trusted policy enforcement, we augment each node with a *trusted agent*, which protects the policy enforcement components from being compromised. When a node joins a trusted tier, its trusted agent helps establish trust by proving the execution of a correct trusted agent, a trustworthy policy enforcing software component (referred to as *policy enforcer* hereafter), and the right policy. Furthermore, it ensures that the integrity of the agent, the enforcer, and the policy will not be compromised. This is possible because the trusted agent is part of the operating system kernel and guarantees the integrity of the kernel and all programs involved in policy enforcement. Therefore, it can foil attacks, including those launched by local users, to tamper with the enforcer or the policy being enforced. If any of these components is compromised, the trusted agent will disconnect the node from the trusted network.

The trusted agent is built on top of Satem [15], our trusted execution monitor based on a low-end trusted hardware, Trusted Platform Module (TPM) specified by the Trusted Computing Group (TCG) [16]). Due to its low cost and broad support by computer makers, the TCG TPM has been already integrated in many laptops. In the near future, it will also be installed on smaller mobile devices such as PDAs and mobile phones [17], which makes our TPM-based approach feasible for MANETs.

This mechanism provides a number of benefits, which make it suitable for MANETs. First, policy enforcement in the multi-tier networks is entirely distributed without relying on any central trusted choke points. Second, the trusted networks are self-organized. They can be established and managed spontaneously without requiring pre-deployed trusted entities or centralized management. Third, the multi-tier trust enables flexible enforcement of complex policies, which can be defined across various interdependent protocols and enforced independently, tier by tier. Furthermore, nodes running multiple applications can join multiple trusted networks, each enforcing policies for different applications without interfering with each other.

We implemented a prototype of the policy enforcing mechanism in Linux and tested it over an IEEE 802.11-based wireless ad hoc network that is composed of TPM-enabled laptops. We also ran NS-2 [18] simulations to evaluate the performance in large scale

MANETs. The experimental results demonstrate low overall costs in application execution and network communication despite high one-time initial cost in network establishment. The simulation results reveal that nodes can join the trusted tiers with high probability even if the underlying MANETs are highly volatile. The overall communication overhead over long network paths increases but still remains at low levels: less than 10% in networks with infrequent connectivity loss and about 20% in high-mobility networks where connectivity among nodes is unstable.

The paper is organized as follows. We motivate the research in Section II through three examples. An overview of the multi-tier network is presented in Section III, followed by the design of the policy enforcing mechanism and the details of the trust establishment protocols in Section IV. Section V describes the prototype implementation. The experimental and simulation results are analyzed in Section VI. The limitations of our method and related work are discussed in Section VII and VIII, respectively. Finally, the paper concludes in Section X.

II. MOTIVATION

In this section, we illustrate the challenge of enforcing even simple policies for three MANET applications. We will show how to solve these problems using our approach in next section.

A. Example 1: Secure Routing

Consider a group of nodes supporting Ad hoc On Demand Distance Vector(AODV) [19] routing protocol. AODV is known to be vulnerable to wormhole attacks [20], in which an attacker exploits a fast tunnel to attract all network traffic through it. One way to defeat this attack is to implement Packet Leashes [20]. For example, a geographical leash can ensure that the destination node is within a certain distance from the source node. It is implemented as follows:

The source node r checks for each AODV reply from the destination node s, $d_{max} > ||p(s) - p(r)|| + 2(t_r - t_s) \times v + e$, where d_{max} is the max distance that the destination node s is allowed from the source node r, $p(s)$ is the position of s at t_s, the time of sending the AODV packet, $p(r)$ is the position of r at t_r, the time of receiving the AODV packet, v is the maximum relative moving speed of the two nodes, and e is the acceptable error. Replies that do not pass the check are deemed as from wormholes and rejected.

We can directly translate the above leash into a routing policy $\mathcal{P}_R$. However, the implementation of the above leash or enforcement of the policy $\mathcal{P}_R$ requires that node r and s be loosely synchronized and r can authenticate s. In general, this is non-trivial in MANET due to the lack of a central time server. In case of anonymous environment, this becomes more difficult since the two nodes can not trust each other. The node has to rely on round-trip delay to estimate the time needed for an AODV message to reach the other. However, this method will accumulate large errors with number of hops and distance between the two nodes increasing. Therefore, the best place to detect the wormhole is on the node that is close to either end of the tunnel. The further away the node is, the less

precise the estimate becomes, and the higher false positives and negatives the method incurs. However, this is infeasible since we do not know
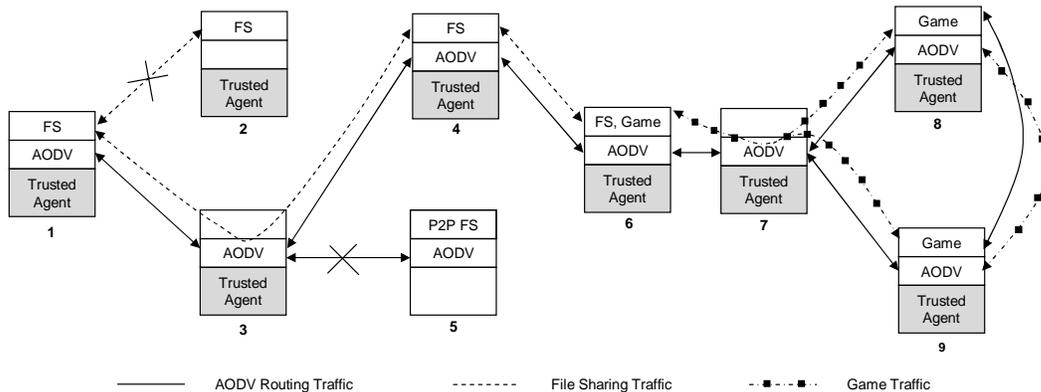

Fig. 1. Policy Enforcement in Multi-tier Networks. Nodes 1, 3, 4, 6, 7, 8 and 9 establish an AODV routing tier. On top of it, nodes 1, 4 and 6 establish a file sharing tier and nodes 6, 8 and 9 establish a game tier. Hence, nodes 1, 4 and 6 form a trusted two-tier file sharing network enforcing both the file sharing and routing policies. Nodes 6, 8 and 9 form a trusted two-tier game network enforcing both the game and routing policies.

B. Example 2: Unselfish Sharing

Consider cars on a highway forming a vehicular network to obtain traffic information ahead of them [21]. Each node simultaneously posts queries, answer queries, receives responses, and forwards queries for others. To benefit all cars in the network, it is vital to ensure that enough cars respond to and relay the queries posted by others. Similar concerns exist in other applications such as a P2P file sharing network, where sufficient file providers are desired. To achieve these goals, each node must abide by a policy $\mathcal{P}_F$, like the following, before joining the network:

Every mobile node has to serve or relay at least 1 request from others after posting 3 queries to the network.

Clearly, the only way to enforce the policy is to do it on every node in the network. Due to the anonymous nature, any identity based policy enforcement method, such as Peace [13], does not apply.

C. Example 3: Fair Game

Consider a group of smart phones using a MANET to play an online game. They are separated into n teams and each of them chooses to join one of the teams at the beginning of the game. To ensure each node can only take one role in the game, the following game policy $\mathcal{P}_G$ is defined:

Each gaming node is free to join any of the n teams. But once it joins one, it can not join another team without first withdrawing from the current team.

The above policy is similar to Chinese Wall Policy [22]. Enforcing such a policy for Internet based applications has been addressed in literatures such as [9]. The existing methods rely on the capability of differentiating one node from another. However, due to Sybil attack [23], this is difficult in MANET.

It is difficult for existing methods to enforce any individual policy aforementioned. To make it more challenging, these policies can be related. For example, secure routing may be prerequisite to secure the file sharing and gaming applications. Therefore, enforcing the file sharing or gaming policy requires that the underlying routing policy have already been enforced. On the other hand, a node may run the file sharing application side by side with the gaming application. Enforcing the policy for one should not interfere with the other. Since nodes can run these applications in any combinations, it is critical to enforce their associated policies flexibly and organically.

III. TRUSTED MULTI-TIER NETWORKS

In this section, we first formally define the trusted multi-tier networks. Then, we illustrate how to create the network through an example.

A. Definition and Policy Enforcement

For some application S, we define the trusted policy enforcing tier $\mathcal{T}_0$, as follows:

$$\mathcal{T}_0 =< N, S, P >$$

where N are the set of nodes communicating through S, and P is the policy defined for S. To facilitate description, we use "." to represent "member of" relation, i.e., $\mathcal{T}_0.N$ means the set of nodes in the tier $\mathcal{T}_0$.

Assume S calls a set of h independent protocols and each protocol is associated with a policy, the nodes running these applications and enforcing their associated policies form $\mathcal{T}_0$'s underlying tiers $\mathcal{T}_1$, $\mathcal{T}_2$, ... $\mathcal{T}_h$. Similarly, each of these protocols may also depend on other protocols and therefore has its own underlying tiers. Assume that there are totally m direct and indirect underlying tiers of $\mathcal{T}_0$, these $m + 1$ tiers form the trusted multi-tier network of S defined as follows:

$$\mathcal{N} =< \bigcap_{i=0}^{m} \mathcal{T}_i.N, \quad \bigcup_{i=0}^{m} \mathcal{T}_i.S, \quad \bigcup_{i=0}^{m} \mathcal{T}_i.P >$$

Each policy is enforced at its associated trusted tier independently. Each trusted tier $\mathcal{T}_i$ ensures both compliance and authenticity of the messages in $\mathcal{T}_i.S$ as follows:

1) **Compliance.** For each member node of $\mathcal{T}_i$ to send a message in $\mathcal{T}_i.S$, it must be permitted by $\mathcal{T}_i.P$.

2) **Authenticity.** For a message of $\mathcal{T}_i.S$ to be accepted by a member node of $\mathcal{T}_i$, it must be sent by another member node.

Compliance ensures that all member nodes abide by $\mathcal{T}_i.P$ in communicating with each other through $\mathcal{T}_i.S$. This is accomplished because only nodes that are trusted to enforce P can join the trusted tier. Once the trust is established, the node's underlying trusted computing system ensures that it will not be compromised. Otherwise, the node will lose its membership of the trusted tier. We will discuss how these are accomplished in the next section. Authenticity prevents a non-member node from creating and injecting messages to the trusted tier. To achieve this, the enforcer on the node attaches a Message Authentication Code (MAC) to each message
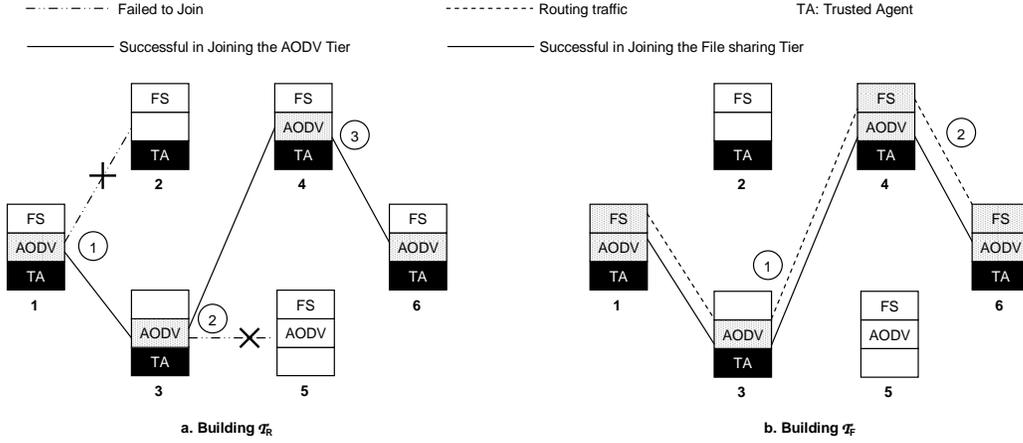
Fig. 2. **Multi-Tier Creation. Building the trusted 2-tier file sharing network demands creating the AODV tier $\mathcal{T}_R$ followed by the file sharing tier $\mathcal{T}_F$. To create the AODV routing tier $\mathcal{T}_R$, node 1 initiates $\mathcal{T}_R$ and invites its neighbor nodes (2 and 3) to join the tier. At step 2, node 2 joins $\mathcal{T}_R$ and further invites its neighbors (node 4 and 5). Finally, node 6 joins the tier at step 3. The file sharing $\mathcal{T}_F$ is then built similarly on top of the AODV tier.**

X of S it sends out. The trusted tier key k_T is used to compute the MAC code e.g., $M_T(X) = HMAC(k_T, X)$. k_T is created when $\mathcal{T}$ is established and shared by all member nodes in $\mathcal{T}$. We will discuss more on the trusted tier key in the following sections.

B. Example: Two trusted two-tier networks

In order to understand the main idea of our solution, let us first consider the example presented in Fig. 1. This example shows a group of nodes using a MANET to run a file sharing and a game application, denoted by F and G. Both applications rely on AODV routing denoted by R. The nodes can build two trusted two-tier networks: (1) a file sharing network $\mathcal{N}_F$ consisting of a file sharing tier $\mathcal{T}_F$ and a routing tier $\mathcal{T}_R$; and (2) a game network $\mathcal{N}_G$ consisting of a game tier $\mathcal{T}_G$ and the same routing tier $\mathcal{T}_R$. A node can join more than one multi-tier networks at the same time (e.g., node 6 in this example). Nodes in each tier must enforce the tier policy, P_R for $\mathcal{T}_R$, P_F for $\mathcal{T}_F$, and P_G for $\mathcal{T}_G$. Formally, $\mathcal{N}_F$ and $\mathcal{N}_G$ are defined as follows:

$$\mathcal{T}_R = < \{1, 3, 4, 6, 7, 8, 9\}, R, P_R >$$

$$\mathcal{T}_F = < \{1, 4, 6\}, F, P_F >$$

$$\mathcal{T}_G = < \{6, 8, 9\}, F, P_G >$$

$$\mathcal{N}_F = < \{1, 4, 6\}, \{R, F\}, \{P_R, P_F\} >$$

$$\mathcal{N}_G = < \{6, 8, 9\}, \{R, G\}, \{P_R, P_G\} >$$

Enforcing P_R, P_F, and P_G is no longer a problem in $\mathcal{N}_F$ and $\mathcal{N}_G$ because they are enforced on every member node of them. For P_R, if there is a wormhole in the networks, the node closest to the wormhole will check the leash and detect the existence of the wormhole. Enforcing P_F and P_G is also straightforward since the history of the node posting queries, serving requests and registering its identity is available on this node.

For two nodes to communicate, they have to be in the same multi-tier network. For example, in Fig. 1, node 1 cannot share files with 3 because node 3 does not enforce P_F and is not a member of the trusted two-tier file sharing network. Neither can files be shared between node 1 and 2 as node 2 does not join the underlying trusted routing tier. On the other hand, the two nodes do not have to be neighbors, as the higher tier application traffic can be routed by the trusted lower tier in a multi-hop fashion. For example, node 1 and 4 can share files securely by routing through node 3. Node 3 is trusted to enforce P_R even though it is not trusted to enforce P_F. Nodes in any trusted multi-tier network must have the trusted agent. Otherwise, they cannot join any trusted tier, such as node 5.

C. Creating a Trusted Multi-tier Network

Building a trusted multi-tier network involves establishing all the trusted tiers it is composed of in a bottom-up fashion. For example, to build the file sharing multi-tier network $\mathcal{N}_F$ in Fig. 1, the trusted AODV tier $\mathcal{T}_R$ is first established followed by the trusted file sharing tier $\mathcal{T}_F$. Fig. 2 illustrates this procedure.

A tier is created step-by-step. First, a node begins to enforce the tier policy. It creates the tier key, which is used to authenticate in-tier communications as discussed earlier. By doing so, it becomes the first member of the tier, called *originator* of the tier, e.g. node 1 in Fig. 2. The originator then broadcasts an invitation to its neighbors, e.g. node 2 and 3, to join the newly created tier. Assume node 2 and 3 choose to join this tier. Since node 3 enforces P_R, it succeeds in joining the tier and receives the tier key from node 1, but node 2 fails because it does not enforce P_R. Next, node 3 extends the tier one step further by inviting nodes 4 and 5. Similarly, node 4 joins and continues the process to include node 6 in the tier. The tier originator controls the size of the tier by setting a TTL parameter in the invitation message. Each node decrements the TTL after joining the tier and stops forwarding the invitation message once the TTL becomes 0. The joining procedure is defined in JOIN protocol, which will be discussed in details in next Section.

Once the routing tier is built, the upper-layer file sharing tier can be built in a similar way. The difference is that the broadcast is in a multi-hop manner. That said, a member node of the file sharing tier broadcasts the invitation to its neighbors. If a neighbor node decides to join the tier, it re-broadcasts the invitation in the same way as in the routing tier creation. Even if the neighbor node does not join the tier, it forwards the invitation message to its neighbors and acts as a router for further communication between the sender and other potential members of the new tier. In an ad hoc network, the neighbor node may also choose not to forward the invitation. If all intermediary nodes are not cooperating, the tier expansion stops.

Each tier application comes with a signed policy. When a node installs an application, it installs its associated policy as well. Tier policies are defined and enforced independently. The policy for a multi-tier network is composed automatically from the individual tier policies based on the application dependencies. For example, Node 1 in Fig. 1 is a member of the routing tier, which ensures the packets sent and received at its file sharing tier are routed through the trusted routing tier. On the other hand, the applications

may come with contradictory policies or may use different label spaces. We do not address these issues in this paper.

The policies are enforced by each node in the multi-tier trusted network rather than by a trusted central authority. Therefore, it is critical to verify a node's trustworthiness of enforcing every tier policy. This is accomplished when the node joins the network through two protocols: JOIN or MERGE. We will discuss them in the next section.

IV. Node Architecture and Protocols

In this section, we introduce the node architecture of our method. As shown in Fig. 3, it consists of a trusted agent (Satem), a tier manager and a number of enforcers, each of which enforces a tier policy. We then discuss in details the two protocols: JOIN and MERGE, followed by the analysis of their correctness.

A. Satem: The Foundation of Trust

We leveraged Satem [15] to build the trusted policy enforcing mechanism. Originally, we designed and implemented Satem to ensure requesters of a remote network service that the service executes only trusted code. Satem is composed of a *trusted agent* in the OS kernel of the service platform and a *trust evaluator* on the user platform. The service provider performs the attestation of the OS kernel (including the trusted agent) through a trusted boot process using the TPM specified by the Trusted Computing Group (TCG). Subsequently, the trusted agent takes advantage of the service execution context to only verify the integrity of the code loaded dynamically by the service. More importantly, it ensures that the service executes only trusted code by protecting the service execution in the OS kernel.

Central to Satem is the commitment protocol. Before starting a transaction with a service, requesters ask the trusted agent to provide the integrity measurements of the OS kernel, a *system commitment*, and a *service commitment*. The commitments are certificates describing all the code files the kernel and the service may execute in all circumstances (e.g., executables, libraries). The system commitment includes the kernel binary and all the modules it may load. The service commitment includes the entire code stack of the service including the service application binary, shared libraries, other applications it calls during execution and their shared libraries.

Each piece of software described in the commitments is defined by a combination of its identifier (e.g., name and version) and the `SHA1` hashes of all its code files. It is the service provider's responsibility to create appropriate commitments. The service provider uses static or runtime analysis to determine the code base. Commitments for an application typically includes several dozens of code hashes. For example, the system and service commitments altogether for the P2P file sharing application running Mute [24] include 47 code file hashes. The service provider collects the code hashes and generates the commitment certificate as follows:

1) *Request code certificates.* The service provider requests each vendor to generate a vendor-signed code certificate in the same format as the commitment for its code.
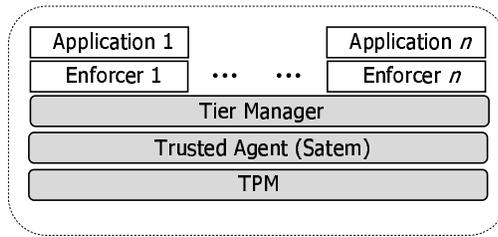
Fig. 3. Node Architecture of the Trusted Multi-tier Network

2) *Sign the commitment.* The requester forwards all the code certificates and the commitment to a third-party trusted Certificate Authority (CA). The CA needs to verify the signatures of all code certificates and compare the code hashes in the commitment against the certificates. The CA signs the commitment if and only if it verifies all code certificates and code hashes in the commitment.

Satem only guarantees the integrity and the authenticity of the code, but not its correctness. The requester must have a local trust policy that governs which kernel and services are trusted. It takes two steps to verify whether a service is trusted. First, it authenticates the kernel and service commitment certificates and learns the identities of the kernel, its modules, and the service [2]. Second, it verifies the kernel and the service against the trust policy.
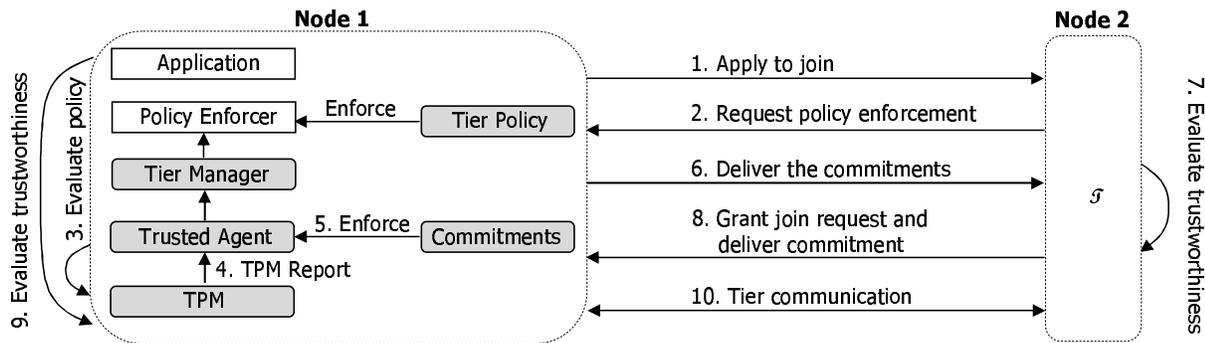

Fig. 4. JOIN Protocol. Node 1 joins the tier by conducting the JOIN protocol with node 2 that is already in the tier.

In Satem, evaluating a commitment is reduced to authenticating the certificate. In general, this is non-trivial in ad hoc networks due to the lack of constant connection to the Internet and the Public Key Infrastructure (PKI) [25]. A node may still be able to authenticate a certificate if it locally holds the public key of the signing certificate authority or a valid certificate chain to it. However, it is unable to validate in real-time the certificate since it has no access to the CA's certificate revocation list. This issue can be significantly alleviated given the special nature of the problem we aim to solve. As discussed in [26], although nodes do not have persistent Internet connectivity, they can still get on-line from time to time. For example, a user may be off-line on an inter-city train most of the time, but get online when the train enters a train station. Furthermore, a Satem commitment only states that a code file has a certain corresponding `SHA1` digest. This fact is invariant under any circumstance. Lastly, since the ad hoc network is formed for a specific task and only lasts for a relatively short period of time, the likelihood of revoking a certificate is negligible.

[2]The commitment authentication is not trivial in ad hoc networks. We will presents our solution for such networks later in this section.

Based on the above observations, we introduce a short-life certificate to authenticate the commitment. Each node obtains a regular long-life commitment certificate C_L, a short-life commitment certificate C_S, and the authority's certificate C_A when being connected to the Internet. When losing Internet connectivity, it can still use the C_A to authenticate C_S of other nodes. Since this certificate is only good for a short period of time, there is no need to be concerned about revocation. After C_S expires, the node needs to regain Internet access to renew it using its C_L. The CA verifies the C_L using PKI and grants the renewal request without re-authenticating it from scratch.

The trusted agent enforces the system commitment at boot time and the service commitment upon being started such that the kernel and the service are forbidden to load any code files that are either undefined in the commitment or tampered with. Therefore, if the requester verifies that the kernel, the agent, and the commitments are trusted, it is convinced that (1) the service has executed only trusted code up to the time of integrity measurement; and (2) the service will continue to do so in the following phases due to the protection provided by the trusted agent. We will discuss more implementation details on commitment enforcement in Section V.

B. Tier Manager and Enforcer

The tier manager is an application that allows the node to create, join and merge into a tier. When the user decides to create a new tier, she calls the tier manager to create the tier key and start the tier creation procedure. Then, the tier manager communicates with the tier managers on other nodes through the JOIN or MERGE protocol. The node may join multiple tiers and thereby, run multiple enforcers. An enforcer is any software that can enforce the tier policy. In the simplest form, the tier application itself has built-in capabilities of enforcing certain policies and can be the enforcer. Both the tier manager and the enforcer must be trusted. This is achieved by defining the code base of the tier manager in the system commitment and the code base of each enforcer in a service commitment (called enforcer commitment in this paper). Consequently, Satem enforces these commitments to prevent the tier manager and the tier enforcers from being tampered with.

Before creating or joining a tier, the user first registers the tier enforcer with the tier manager. As explained later in JOIN protocol, this enables the tier manage to deliver the correct enforcer commitment. Moreover, at the end of the JOIN and MERGE protocols, the tier manager receives the tier key. Then, the tier manager can deliver the key to the right enforcer that has been protected by the trusted agent.

C. Joining a Tier

The JOIN protocol is used when a node wants to join a trusted tier for the first time. The new node communicates with a member node of the trusted tier. The member node has to verify that the new node is trustworthy to enforce the tier policy. At the same time, the new node must also verify the trustworthiness of the member node. Fig. 4 illustrates the JOIN protocol. In the figure, we assume

that Node 2 is already a member of a trusted tier and Node 1 wants to join this trusted tier. In this example, we also assume that each application comes with an associated policy, which is stored on each node together with the application [3].

1) **Request to join.** As illustrated in steps 1-2 of Fig. 4. Node 1 sends a join request to Node 2 by specifying the application identity (e.g., the IP address and port number) and receives a request for a guarantee of trusted enforcement of the tier policy.

2) **Deliver the commitment.** This is done by steps 3-6 of Fig. 4. Node 1 first evaluates whether the policy can be enforced. Then, it calls the trusted agent to generate a Satem report including (a) its system commitment, (b) the enforcer commitment (i.e. the service commitment defined for the enforcer), and (c) the integrity measurement of booting. Finally, Node 1 sends the Satem report to Node 2 for evaluation.

3) **Evaluate the commitment.** This is step 7 of Fig. 4. Node 2 first authenticates and verifies the integrity of the commitments and attestation. Then, it verifies the system commitment, the enforcer commitment, and the boot attestation in the Satem report against the local trust policy before accepting Node 1 to the tier. From the boot attestation, the member node learns that the requesting node has been booted into a trusted Satem kernel. Knowing the system commitment convinces the member node that the kernel of the requesting node will not load untrusted modules, which protects the trusted agent from being tampered with. Knowing the enforcer commitment convinces it that the enforcer software execution stack on the requesting node is trusted because the trusted agent will enforce the commitment to prevent untrusted code from being loaded by the enforcer.

4) **Grant the join permission.** As shown in step 8-10 of Fig. 4, Node 2 accepts the join request and sends the trusted tier key along with its own Satem report to Node 1. Node 1 then verifies the report the same as Node 2 did at step 7. After it sets the tier key and enforces the tier policy, Node 1 becomes a member of the tier and can not exchanged with other nodes in the same tier using the tier key.

D. Security Analysis of JOIN Protocol.

Attacker model. Let us consider a local attacker on Node 1 (the analysis holds if the attacker is on Node 2). We assume that the attacker cannot break the TPM or launch hardware based attacks, and in particular, cannot use direct memory access (DMA). We further assume that the attacker is unable to bypass the node operating system to gain access to system resources, such as memory, CPU, network card, and disk. To achieve this, direct memory access from user space via /dev/mem and /dev/kmem is disabled in Satem implementation. Other than these restrictions, the attacker can have full control of the software system, including root privilege.

- **Disable enforcement of the tier policy.** The most direct attack is to disable the enforcement of the tier policy after obtaining the connectivity. The attacker can do so by disabling the policy enforcer. This requires removing the policy enforcer's kernel module. The trusted agent intercepts the removal request and clears the tier key before removing the module. Thus, the attacker has to first disable the trusted agent (Satem). The attacker may choose not to enable the trusted agent. However, this will make

---

[3]Policy distribution is discussed in more details later in this section

the node fail to join the tier at step 7. Alternatively, the attacker may want to disable the trusted agent after being accepted to the trusted tier. However, the only way to turn off the trusted agent is to reboot the system, which will wipe out all tier keys and force the node to rejoin the tier.

- **Modify the policy.** The attacker may attempt to modify the current policy at runtime. The trusted agent secures the memory space holding the policy, such that only the tier manager has write permission to it. Additionally, the tier manager is protected by the trusted agent. The attacker may try to run a malicious tier manager. But this will make the node fail to pass evaluation at step 7.

- **Steal the tier key.** The attacker may attempt to steal the session key on the node. One of the creators was selected to generate the key, which became the first node in the tier. Which node was selected is irrelevant since the key is secured by the trusted agent in memory and accessible only to the tier manager and the tier policy enforcer. The tier manager and the enforcer are not allowed to disclose it to any other program or save it to disk. Violating enforcers will either be stopped due to the enforcer commitment, or revealed to the member node, thus causing the join request to fail. The protocol ensures secure distribution of the key by establishing mutual trust between the joining node (Node 1) and the member node (Node 2). On one hand, the member node (key owner) will not distribute the key to any untrusted node (step 8). On the other hand, the joining node will not accept the key from a member node unless the member node has been verified to be trusted (step 9). Consequently, an untrusted node cannot create a key and fool others to accept it.

- **Hijack the key.** The attacker may try to steal the tier key in distribution by intercepting it at step 8. To ensure secure distribution of the key, the tier manager on Node 1 dynamically creates a public-private key pair (PK_1, SK_1) when it is loaded to run. The public key PK_1 is passed to Node 2 along with the TPM report at step 4 which is used by Node 2 to encrypt the tier key at step 8. Node 1 then decrypts the encrypted tier key using the correpsonding private key SK_1. The private key is cached in the tier manager's memory and never disclosed to any other processes or saved to disk. The trusted agent protects the key from being disclosed to any process other than the tier manager. In case of system suspension, all tier keys are wiped out. When the system resumes operation, it has to re-join the previous tiers. Hence, the attacker is unable to acquire SK_1 to decrypt the tier key intercepted at step 8.

- **Play man-in-the-middle.** The attacker may want to replay a valid TPM report and exploit it to gain trust from Node 2. The protocol foils the attacker by including a nonce in the attestation report. The attacker may also attempt to play a man-in-the-middle attack by creating her own public-private key pair ($\widetilde{PK_1}$, $\widetilde{SK_1}$) and replacing the aforementioned (PK_1, SK_1) with them in order to decrypt tier key with $\widetilde{SK_1}$.

Although PK_1 is not authenticated in the protocol, it is attested in the TPM report. When the TPM report is generated, the tier manager computes the hash value of PK_1 and passes it as a parameter to the TPM. By evaluating the TPM report, Node 2

knows that the public key belongs to the node that generates the report. Since the joining node is trusted because of the report and is the only one that has the corresponding private key to decrypt the tier key, the member node is assured that tier key is distributed to a node that is trusted to enforce the policy.

Our method was originally developed in Satem [15] and is similar to [27]. It is orthogonal to identity based authentication and can be used in combination with any existing MANET key authentication methods such as [28], [29], [30], [31], [32].
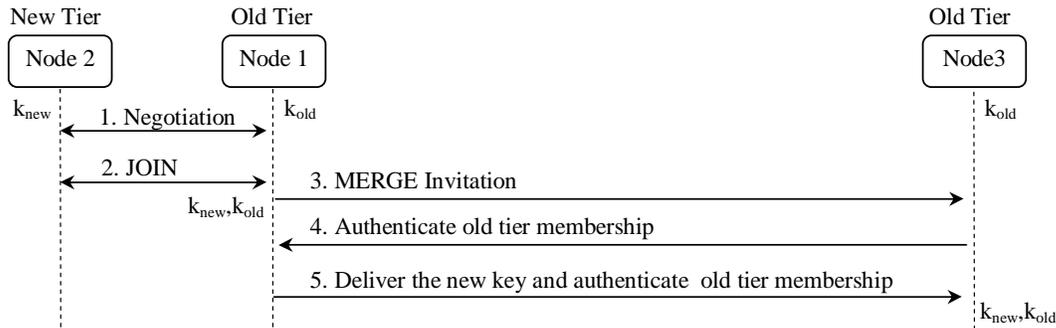

Fig. 5. Merge Protocol. The old tier and the new tier enforce the same policy but were created separately and have different tier keys. Node 1 in the old tier joins the new tier by conducting the JOIN protocol with Node 2. Then, Node 1 calls for other nodes (e.g. Node 3) in the old tier to merge into the new tier through the MERGE protocol.

E. Merging Tiers

Two tiers enforcing the same policy but using different keys can be united by the MERGE protocol. Every node has an equal opportunity to establish a trusted tier. To prevent multiple nodes from establishing the same trusted tier at the same time, the originator may first query its neighborhood for the existing tier. However, this method does not work if two nodes are not reachable from each other. As a result, they will create two trusted tiers running the same application and enforcing the same policy, but holding different trusted tier keys. When later connectivity becomes available between them, they cannot communicate with each other. In this case, the two trusted tiers can be merged by unifying the two tier keys into one common key. This procedure starts when a node (Node 1) in a tier (tier A) learns the existence of another tier (tier B) nearby. For instance, it may receive a message from Node 2 that it cannot authenticate. This may indicate that Node 2 is running the same application but having a different key. To verify whether tier B is enforcing the same policy, Node 1 exchanges its policy with Node 2. If the policies are the same, Node 1 starts the MERGE protocol to unify the key. Fig.5 shows the steps of the MERGE protocol, which can be categorized in two phases.

1) **Change membership.**

This is the steps 1 and 2 in Fig.5. Nodes 1 and 2 first negotiate the new key to be used by the merged trusted tier. They compute a hash of their own keys and select the key with greater hash value as the new trusted tier key. Assume the key of Node 2, k_{new} is chosen, then Node 1 joins Node 2's tier through the JOIN protocol mentioned above. After receiving the key,

Node 1 verifies it against the hash received in the previous step. It still keeps the old key, k_{old}, for a certain period of time. The old key will be used to authenticate other member nodes in the old tier.

2) **Convert membership.**

This is the steps 3-5 in Fig.5. Node 1 broadcasts a MERGE message to its neighbor nodes in its old tier with a nonce. A neighbor node, Node 3, and Node 1 then mutually authenticate to each other by exchanging a nonce and a message authentication code (MAC) computed using their old tier key, k_{old}. Successful authentication of the MACs verify both nodes' membership of the old tier. Finally, the new tier key is encrypted with k_{old} and delivered to Node 3.

Similar to the tier creation procedure, the MERGE is broadcast in a multi-hop manner. For instance, the MERGE message of the file sharing tier can be delivered to other nodes in the old tier through a series of AODV router nodes that are not in either of the new and old file sharing tier. The MERGE protocol does not guarantee two trusted tiers to be merged completely in one run. In fact, merging is driven by the need of facilitating communication. As a result, it only aims to merge the nodes which interact with each other. In practice, a TTL can be used in MERGE messages to limit the scope of merging in the same way as trusted tier establishment. Nodes beyond the coverage of the TTL may later merge into the trusted tier when they need to interact with nodes in the trusted tier. In this way, merging is carried out on-demand step by step.

F. Security Analysis of MERGE Protocol

As discussed in JOIN, the attacker is unable to break the tier manager, the policy enforcer, or the trusted agent. As a result, she cannot obtain the tier key without enforcing the policy. Additionally, she cannot modify the policy or steal tier keys on the machine either. Therefore, the new key distributed at step 5 is safe.

Except for initial membership change (e.g. merging Node 1 into the new tier), verifying the trustworthiness of the nodes in the old tier is reduced to simply tier key authentication. This is because these nodes must have been verified before they were allowed to join the old tier for the first time. Owning an old key implies it has been protected by the trusted agent and is still protected by Satem. Otherwise, if any program defined in the commitments was compromised since last JOIN, the trusted agent would have wiped out the key. This property helps simplify the trust verification process in the MERGE protocol. Since computing the MAC code is much cheaper than the Satem report, the performance is greatly improved.

V. IMPLEMENTATION

We implemented the policy enforcing mechanism prototype under the Linux 2.6.12 kernel. It consists of the Satem based trusted agent and the tier manager. To evaluate the performance, we also implemented enforcers for two MANET applications: AODV (user-level daemon) [33] for ad hoc routing and Mute [24] for P2P file sharing.

A. Satem Trusted Agent

The focus of the trusted agent is to provide a fail-stop protection mechanisms to enforce commitments. Our implementation is integrated into the OS kernel in many places by inserting checkpoints to kernel calls such as do_execve, sys_init_module, and sys_open to intercept new code execution invoked by protected processes. We add these modifications by patching the original Linux kernel.

Trusted System Initialization. In Satem, the first step is to establish the trusted computing base that includes the trusted agent and the entire OS kernel. This process involves a trusted boot, in which each component in the boot sequence, starting from the TPM, measures the integrity of the next one before handing over the control. In our case, the TPM measures the integrity of the BIOS image by computing the SHA1 hash over it and then transfers control to the BIOS. Next, the BIOS calls TPM to compute SHA1 hash over the OS loader (e.g., LILO), and the latter does the same over the OS kernel image, denoted as OSK. The measurement is saved in a PCR register (PCR_0), which is an internal configuration register of TPM. As a result, after the OS kernel is loaded,

$$PCR_0 = SHA1(SHA1(SHA1(0|BIOS)|$$

$$LILO)|$$

$$OSK)$$

assuming $PCR_0 = 0$ initially.

The SHA1 hash is computed in a chained fashion using the TPM API TPM_Extend, which is the only way to change the content of a PCR. As a result, all the components can be measured using a single PCR. More important, this prevents the PCR content from being reset (i.e., the attacker can change the measurement value, but it cannot set it to an arbitrary predetermined one). The content of selected PCRs is reported via the TPM_Quote API, which signs the content with a TPM internal key. We assume that the TPM is unbreakable and, therefore, we consider that the integrity measurement it makes and the report it produces cannot be tampered with [4]. As a result, the TPM report containing the measurement result saved in the TPM is sufficient to prove a genuine kernel and trusted agent.

Commitment Enforcement. Before a policy enforcer is started, the user needs to associate the commitments with the system and enforcer executables such that the trusted agent can load the right commitment when executing the enforcer. To do so, the user defines a list of enforcers in a configuration file, which maps the enforcer binary to its commitment. Once the enforcer starts, the trusted agent locates the commitment from the configuration file. The trusted agent then associates a protection flag with all processes executed by the enforcer, memory regions mapped by these processes, and code files opened by them. The trusted agent enforces the commitment in each checkpoint only if the flag is present. The only exception is modifications to the kernel. Satem always enforces system commitments regardless which process attempts to load the kernel modules. The checkpoints used by our method to enforce

[4]The current TPM specifications use SHA1, which has been found breakable [34]. We expect future releases of TPM to be upgraded with a stronger one-way hash function such as SHA256.

system and enforcer commitments are the following:

1) *Loading.* When the enforcer e is executed, the trusted agent intercepts system calls such as `do_execve` and loads the pre-configured enforcer commitment. Then, it computes `SHA1(e)` and verifies its integrity against the enforcement commitment. Besides, it marks the current process executing e as protected. Whenever a protected process loads a code file c, the trusted agent will verify c's integrity against the enforcer commitment.

2) *Binding.* If the program is a service, it needs to bind a network socket upon being loaded. The `sys_bind` system call traps into the kernel. The trusted agent links the socket to the commitment associated with the process;

3) *Module Inserting.* When a kernel module is loaded, the trusted agent intercepts `sys_init_module` and verifies its integrity against the system commitment without checking the calling process's protection flag.

4) *Mapping.* Modern OSes map rather than load code files, including shared libraries, into memory. When a protected process maps a segment of a code file c into its memory region r, the trusted agent marks r as protected, computes `SHA1(c)` and verifies its integrity against the commitment. Additionally, it partitions r into a series of page-sized chunks $r_1, r_2, ..., r_k$, computes every `SHA1(`r_i`)` ($i \in [1..k]$), and saves them in kernel memory.

5) *Paging.* When a mapped code chunk r_i of a protected memory region is loaded into memory via kernel calls like `file_nopage`, the trusted agent computes `SHA1(`r_i`)` and verifies its integrity with the values previously computed in the mapping.

6) *Forking.* When a protected process p forks a child process p', the trusted agent marks p' as protected.

Satem exploits a lazy integrity measurement to minimize overhead. It measures the code file page by page and caches the results. When the code file is executed subsequently, Satem does not measure it again if the associated commitment has not been changed. Instead, it verifies each of the loaded chunks against the cached measurement results. Satem will re-measure the entire code file only if the execution loads a chunk that has not been loaded before.

The commitments are maintained in the kernel memory as a table. The trusted agent uses the file name as the key to look up the corresponding hash value. The trusted trusted agent maintains the list of `SHA1` hashes for the mapped binaries of each protected program at page level. We use the Linux kernel crypto API to implement the `SHA1` functions.

B. Policies and Enforcers

We define the policy for Mute, P_F, and the policy for AODV, P_R, as illustrated in Fig. 6. These policies address the security issues for the routing and file sharing described in Section II. In P_R, when node i receives an AODV reply message `RREP(o,d)` to a previous query originating at `o` for destination `d`, it must compute and check the geographical leash. The route is accepted if and only if the leash is valid. In the evaluation, the max distance $dmax$ and runtime distance between the two nodes $d(i,d)$ are constant and pre-defined. td and ti are the local time on node d and i respectively. In P_F, each node is given 3 credits in the beginning. The

credits are deducted by 1 every time the node rejects a request for a file it owns or requests a file from other nodes, and added by 3 every times the node serves a request. The node must maintain positive credits to be able to request new files from others.

The enforcers can be any software that understands and enforces the policies. In general, users need to provide the right enforcers for the applications as external components. For evaluation purpose, we implemented simplified enforcers for the above Mute and AODV policies by modifying the applications' source code and hard coding enforcement of the policies. Hence, the applications are also the enforcers of themselves and must be trusted. This is done by having them included in the enforcer commitments and protected by Satem.

C. Tier Manager

The tier manager is a service application. It implements the tier creation, JOIN and MERGE protocols. In addition, it provides the registration service for the user to register tier enforcers by providing the server port that the enforcer listens on. Since Satem maintains the mapping between the application port and its commitment, the tier manager can use the port number to retrieve the commitment of the enforcer during JOIN protocol. Therefore, when the tier manager receives the tier key at the end of JOIN or MERGE protocol, it delivers the key to the registered enforcer that has been verified.
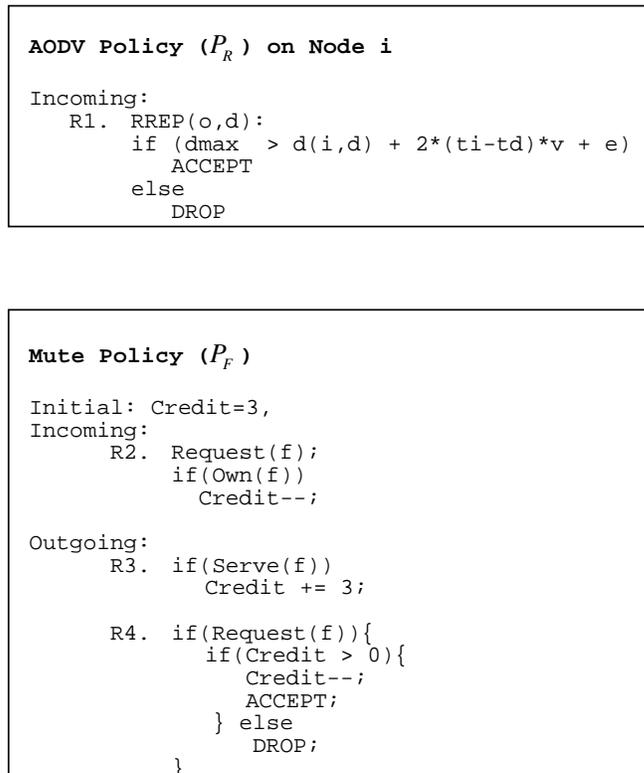
```
AODV Policy (P_R) on Node i

Incoming:
    R1.  RREP(o,d):
            if (dmax  > d(i,d) + 2*(ti-td)*v + e)
               ACCEPT
            else
               DROP
```

```
Mute Policy (P_F)

Initial: Credit=3,
Incoming:
      R2.  Request(f);
             if(Own(f))
                Credit--;

Outgoing:
      R3.  if(Serve(f))
              Credit += 3;

      R4.  if(Request(f)){
              if(Credit > 0){
                 Credit--;
                 ACCEPT;
              } else
                 DROP;
           }
```

Fig. 6. **The Example Policies.** P_R **implements the secure routing policy** P_R **for AODV tier, which requires that each node check the packet leash.** P_F **is the unselfish sharing policy for Mute tier, which ensures that nodes in the file sharing network serve at least one file download request from other nodes after posting 3 requests.**
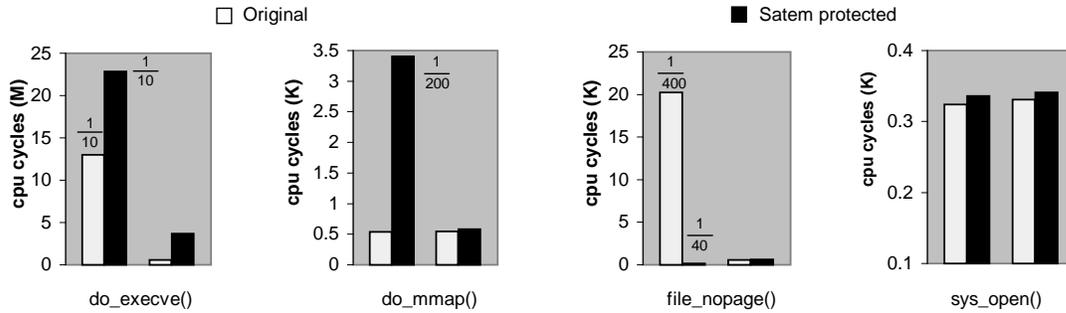
VI. EVALUATION

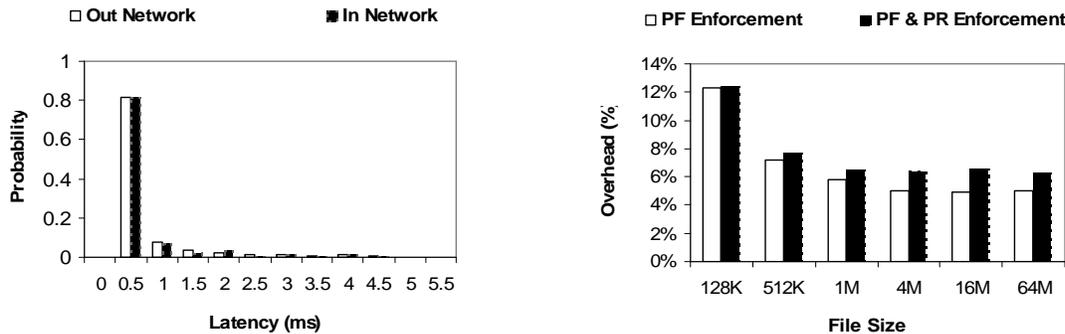**Fig. 7. Overhead of Commitments Enforcement in Kernel Calls**

Fig. 8. Probability Distribution of Ping Latency with and without Policy Enforcement for Routing (AODV)

Fig. 9. Policy Enforcement Overhead in Mute and Mute+AODV

To evaluate the performance of our mechanism, we ran prototype experiments and simulations. In the experimental evaluation, we created a 2-tier trusted ad hoc network over 3 laptops and measured the overhead incurred by our policy enforcing mechanism in application execution and communication. To understand the performance overhead of network creation and policy enforcement in large MANETs, we also ran simulations in various mobility scenarios using NS-2 [18].

A. Experimental Evaluation

Our method impacts system performance by adding latency to (1) the kernel call execution due to enforcing the Satem commitment, (2) joining the trusted network due to verifying trustworthiness during JOIN and MERGE protocols, and (3) the network communication due to enforcing the policy and computing and verifying the MAC for application messages.

1) Methodology:

We created an 802.11g ad hoc network consisting of three laptops (IBM T43 with a 1.7Ghz Pentium M CPU, 512M RAM, and Atheros wireless card). In order to test multi-hop communication, the network was configured with a line-like logic topology (i.e., the direct link between laptop 1 and 3 was disabled, but they could still communicate with each other through laptop 2 as a router). This was achieved by enabling MAC filtering using `iptables` [35] on each laptop.

Each laptop ran two applications: AODV (user-level daemon) [33] for ad hoc routing and Mute [24] for P2P file sharing. We used P_R and P_F defined in Fig. 6 and created a two-tier trusted network consisting of a file sharing tier enforcing P_F and an underlying routing tier enforcing P_R. We used a simplified method to implement the Mute and AODV enforcers by directly modifying the applications' source code and adding the policy enforcement functionality. As a result, the two applications themselves are trusted and attested by the enforcer commitments.

2) Results:

Kernel call cost. The overhead in kernel calls was incurred by the Satem trusted agent, which enforces the system and enforcer commitments. We measured it in terms of extra CPU cycles needed to complete these calls. Fig. 7 shows the overhead in four kernel calls: do_execve, do_mmap, sys_open, and filemap_nopage. We measured the cost of these functions in enforcement of P_F in Mute enforcer. All functions are measured in two cases: in the original Linux 2.6.12 kernel and in the Satem kernel. For each function, we measured its overhead for the first call (the left two columns in the figures) and the overhead for subsequent calls (the right two columns in the figures).

The graphs show that the impact of Satem on the first time function call is significant [5] . For do_execve and do_mmap, the cost is dramatically increased because of the by-page integrity measurement for mapped code files. For filemap_nopage, by contrast, the cost is significantly reduced. This is because Satem needs to load every mapped page into the page cache when it measures a protected memory region. As a result, when a page fault occurs, it is very likely that the page is still in the cache. Furthermore, the costs of the subsequent calls in the Satem kernel are significantly reduced. This is because of the lazy integrity measurement mechanism in Satem. Cost of the subsequent do_execve calls is still large compared with the original kernel, but the impact on the system is limited since it is only one-time cost.

Joining latency. We measured the joining latency as the delay between the time a node starts joining or merging into a tier and the time it is accepted as shown in Table I. Both protocols incur significant latency. The high overhead is mitigated from two perspectives. First, JOIN is only used once for the initial connection. Cost of reconnection is greatly reduced by MERGE protocol. Second, if a node loses wireless connectivity to others just for a short period of time, it is unlikely the trust tier the node belongs to will update its policy or merge with other tiers. Therefore, once the node regains the wireless connectivity, it can simply reconnect to the tier without running either JOIN or MERGE protocols. With that said, even MERGE protocol is rarely needed.

The latency of JOIN protocol is largely due to the time the TPM takes to generate signatures, which varies dramatically by the TPM models and vendors. The TPM we used in the test were manufactured by National Semiconductors. TPMs with higher performance are already available [36]. JOIN latency between machines equipped with these TPMs is expected to be much lower.

[5]The cost of the functions of the first and subsequent invocations may be dramatically different. In order to compare them in one figure, we use a reduced scale to plot the following figures: first call to do_execve at 1/10, first call to do_mmap in the Satem kernel at 1/200, first call to filemap_nopage in original kernel at 1/400, and first call to filemap_nopage in the Satem kernel at 1/40.

Scenario	Latency (in seconds)
Join	2.54
Merge	0.38

<div align="center">TABLE I</div>

<div align="center">**Tier Joining and Merging Delay**</div>

Network communication delay. We measured the routing latency indirectly. We ran the AODV daemons on all three laptops and pinged laptop 3 from laptop 1. Then, we measured the round trip time (RTT), which includes both packet transmission latency and the routing cost. Since all packets are at the same size and enforcement of P_R has no impact on the ICMP protocol, the cost of transmitting each packet is the same. The routing cost varies with the change of the network state. In one extreme, the routing cost is high when laptop 1 has to build a full route from scratch since all 3 laptops have no routes to the others. The overhead of enforcing P_R is also high in this case because enforcement is performed on all nodes on the route. In the other extreme, the routing cost is 0 once the route is established and remains valid; laptop 1 can keep sending packets via the route. Therefore, the enforcer is not invoked and the enforcement overhead also becomes 0. To test the different cases, we randomly invalidated the existing routes on each laptop by blocking and unblocking the wireless network interfaces. The enforcement overhead is determined by the probability distribution of these scenarios.

Fig. 8 compares the probability distributions of ping latency in the trusted network with P_R being enforced (called in-network) and in the original network with P_R not being enforced. Clearly, the latency in the two cases are nearly identical, indicating little impact of enforcing P_R on the routing latency. In both scenarios, the majority (over 85%) of routing efforts incurred low delay (less than 1ms). This was because our route invalidation is infrequent and most of the time, laptop 1 had a valid route to laptop 3 and the enforcer was not invoked. Hence, the mean overhead is low (less than 5%).

To measure the overhead of enforcing P_F, we compared the downloading speed of Mute application in three cases (1) run Mute in the original network with no policy enforcement, (2) run Mute in the 1-tier network with only $\mathcal{T}_F$ tier enforcing P_F, (3) run Mute in the 2-tier network with $\mathcal{T}_F$ and $\mathcal{T}_R$ tiers enforcing P_F and P_R respectively. We measured the download latency in case (1) as the baseline and measure the percentage of increased delay in case (2) and (3). The results are summarized in Fig. 9.

Compared with no enforcement, the enforcement overhead in both case decreases with the size of the file being transferred increasing. This is because the overall costs consist of the initial cost of enforcing the policy in handling file download requests plus the ongoing cost of computing or verifying the MAC code for each Mute message being sent out or received. With the file size increasing, network transmission and routing cost becomes more significant, making the policy enforcement cost relatively small. However, the policy enforcement overhead does not vanish. Instead, it levels off at around 6%. The overhead is mainly due to the commitment enforcement by the trusted agent and the MAC generation and verification by the tier enforcers. The cost of enforcing

the policy set $\{P_R, P_F\}$ demonstrates similar pattern with high level of overhead due to the extra cost of enforcing P_R. One important difference is that the cost of enforcing P_R increases with the length of the route between the two nodes, while the cost of enforcing P_F does not. This is due to the fact that P_R is enforced by all nodes on the route, while P_F is only enforced by the two endpoints. We will show in the simulation that this difference causes the overhead of routing to increase dramatically in complex dynamic networks.

B. Evaluation through simulations

In the simulation, we used the NS-2 simulator to evaluate how the overhead in creating the trusted multi-tier network and enforcing the policies varies in complex MANETs with different mobility scenarios.

1) Methodology:

The simulation includes three types of mobility models: highway vehicular network, city vehicular network and a network with nodes moving randomly at walking speeds. We leveraged our vehicular simulation tool [37] to generate the highway and city networks. The highway scenarios simulated is a 10 mile segment of New Jersey Turnpike with 200 nodes (cars) moving at a speed from 45 miles per hour (20 m/s) to 72 miles per hour (32 m/s). The city scenarios is a 1.2x1 miles region of Los Angeles with 100 nodes moving up to 40 miles per hour (18 m/s). We also modified Carnegie Mellon University *setdest* [38] utility to generate random waypoint mobility models with 100 nodes in a 1x1 miles region, moving at walking speeds. In our simulations, we set the node density to be around 6-8 neighbors per node to avoid connectivity failure due to sparse networks or too much contention in over-crowded networks. In all scenarios, we ran the simulations for 300 seconds.

Since the cost of cryptographic operations associated with JOIN, MERGE, and enforcement cannot be ignored but NS-2 does not account for execution time, we add a certain latency for these operations. Specifically, these additional latencies are modeled as normal random variables with standard deviation equal to 10% of its mean. The mean latencies are: 1150ms for JOIN, 180ms for MERGE, and 0.15ms for enforcement. These numbers were obtained from the previous experimental results.
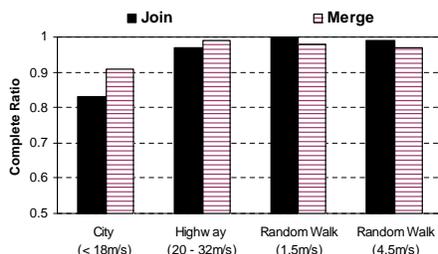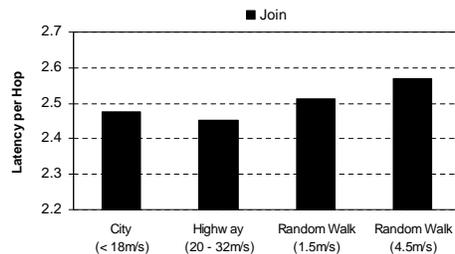

Fig. 10.   JOIN and MERGE completion ratio

Fig. 11.   JOIN latency per hop

2) Results:

Cost of network creation. We measured the cost of network creation in terms of both successful ratio of JOIN and MERGE
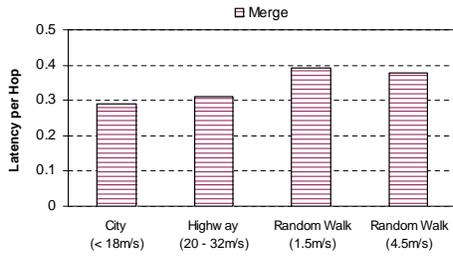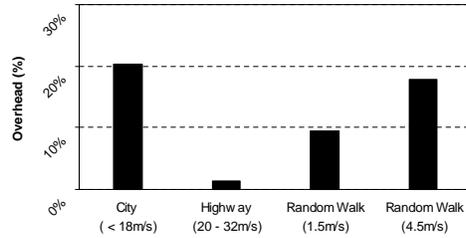
Fig. 12. MERGE latency per hop

Fig. 13. AODV Policy Enforcement Overhead in Ping RTT

operations and the latency it takes for a node to join the network. The completion ratio of JOIN (or MERGE) is defined as the total number of nodes that successfully join (or merge into) the network against the number of nodes that apply to join (or merge into) the network. To test MERGE, we first set all nodes in the network to be the members of the old tier. We randomly selected one node and updated its membership to become the first node of the new tier. Then, this node automatically started the MERGE process with other nodes.

As illustrated in Fig. 10, in most cases we achieved a completion of over 80% for both JOIN and MERGE. The ratio is lowest in the city scenario. This is mainly because nodes exit the region when they reach the boundary of the map. This problem does not exist in the random walking scenarios and is less of a problem in the highway scenario since most cars stay in lanes without exiting the highway.

The latency for a node to join the network is measured as follows:

$$joining\ latency = \frac{t_i - t_0}{dist_i} \tag{1}$$

where we denote t_i as the time node i joins the network, t_0 as the time the originator initiates the network, and $dist_i$ as the number of steps the join invitation message has traversed before reaching node i. In another words, we measured the latency per hop. The reason to do so is because obviously the more number of hops a node is away from the network originator, the longer it takes for it to join the tier. We do not count nodes that fail to join the network since the latency for these nodes is infinite.

Fig. 11 and 12 show that both vehicular networks incurred less latency in joining and merging into the trusted network. This result can be explained by the broadcast storm problem [39] in tier creation. In the random waypoint scenarios, the number of broadcast messages increases exponentially with the network expanding neighborhood by neighborhood, which leads to many packet losses due to contention. By contrast, in the vehicular network scenarios, messages have to propagate along the road unless the sending cars are at the road intersections. This means that most of the time the number of messages that are being broadcast in the network does not grow. Therefore, latency is low in this case.

Enforcement overhead in AODV. We measured the enforcement overhead in AODV in the same way as we did in the previous subsection. We randomly selected the source and destination nodes and let the sources repetitively ping the destinations. We measured the per-hop RTT by dividing the round-trip time by the number of hops traversed. We denote BRTT as the basic per-hop RTT measured

when P_R was not enforced and ERTT when P_R was enforced. We computed the overhead as

$$overhead = \frac{ERTT - BRTT}{BRTT} \times 100\% \tag{2}$$

As Fig. 13 reveals, the overhead is higher than in the simple prototype experiments (Fig. 8), but they still remain under 20% in all cases. The main reason for the overall increase is that the network is highly dynamic and the established routes do not last long. Frequent broken routes trigger route repairs, which involve policy enforcement. In the prototype experiments, the route re-establishment was far less frequent.

The worst case is the city scenario because it is the most dynamic network. The overhead is small in the highway scenario. This is because the relative positions between most nodes do not change compared to other networks though each node itself moves at highest speed.

VII. LIMITATIONS AND FUTURE WORK

We leveraged static root of trust to establish trust on the trusted agent. In practice, this approach is known to be susceptible to a number of attacks due to bugs in implementations of boot loader, BIOS and TPM [40]. These vulnerabilities may be mitigated by the dynamic root of trust feature of new processors [41]. Another limitation is that Satem only measures and protects the code that the application depends on. As pointed out in [42], [43], the trustworthiness of the application also depends on the dynamic data it uses. Roti [43] provides a solution to this problem.

Satem only ensures that a protected service can not load untrusted code from the disk. It is unable to tackle attacks, like buffer overflow, that can cause the protected service to run arbitrary code without changing its disk image. Satem only mitigates the problem in two aspects. First, Satem may reveal the code that has known buffer overflow vulnerabilities by attesting it to the user. Hence, the user can avoid trusting the vulnerable code. Second, in the case of a successful buffer overflow attack, the attacker runs her own code on the service stack without being caught by Satem. But due to the limited size of the stack, the attacker's code typically has to call other local programs on the service provider to make the attack meaningful. Satem restricts the attacker's capability of launching arbitrary local code (i.e., any code launched by the protected service must be defined in the commitment).

The tier keys are protected in memory. However, a recent study [44], [45] demonstrates the possibility of retrieving the keys directly from DRAM, since DRAM still retains the content even after being pulled out from the motherboard. Fully addressing this vulnerability may require architectural changes to DRAM to make it lose memory faster.

Satem kernel code is not modularized due to the need of inserting integrity check points at various places in the kernel. This makes the code difficult to port and modify. We are exploring other methods such as Linux Security Module [46] for improvement.

In the current prototype, we implemented the enforcer by hard coding the policy enforcing function in the application source code. This is inflexible since changing the policy may require modifying the application. In the future, we plan to implement a

standalone enforcer as the transparent application proxy. In this way, the application request is redirected to its local enforcer, which communicates with the application on the remote node. One way to achieve this is to establish the mapping between the application and its enforcer when the enforcer registers with the tier manager. To do so, the user provides the tier manager with the TCP or UDP port number on which the application S listens, p_S and the port number on which the enforcer listens, p_E. The manager then maintains the mapping between the application port p_S and the enforcer port p_E in the kernel by using Linux built-in kernel hooks `NF_IP_LOCAL_OUT` and `NF_IP_PRE_ROUTING` [35] as follows:

1) `NF_IP_LOCAL_OUT`

   When the local node n_l sends a message to S on a remote node n_r, the kernel maps destination port $n_r : p_S$ to $n_l : p_E$. This causes the message to be redirected to the local enforcer. E computes and attaches the MAC code for the application message.

2) `NF_IP_PRE_ROUTING`

   When the local node n_l receives a message for S from a remote node n_r, the kernel maps destination port p_S to p_E, which causes the message to be redirected to the local enforcer. The local enforcer first verify that the attached MAC code is correct. Otherwise, it drops the message. Next, it strips off the MAC code and forwards the message to the application.

VIII. RELATED WORK

Our work leverages previous research on trusted computing and distributed policy enforcement.

Distributed Policy Enforcement. The idea of trusted policy enforcement on each network node can retrospect to our earlier work in [14]. In that paper, we developed a Satem-based method to implement network access control in ad hoc networks. This paper further extends the idea in two fronts. First, the nodes can now verify the trustworthiness of each other at any layer and for any application rather than just at link layer as in [14]. This enhancement enables finer grained control of network establishment. Second, the policies can be associated with any applications rather than just with the network layer. This makes it possible to regulate the communication in any application protocol.

Much of research on security policies focuses on policy representation and evaluation [1], [2] or building security mechanisms based on specific policies [47] without addressing policy enforcement. McDaniel et al. implemented Antigone [48], a general-purpose policy enforcement mechanism. However, it is only concerned with providing API's to integrate various policy enforcing software components.

Enforcement of access control policies can be implemented by use of reference monitors. Conventionally, the reference monitors are managed by a trusted entity in a centralized way, such as in [3], [4], which is suitable for enterprise computing rather than ad hoc environments. Recent research efforts have been seen to distribute the monitors [5], [6]. However, the methods are in essence still server-centric and rely on trusted servers to host the monitors, while our method is application-centric and does not assume any pre-existing trusted nodes.

Minsky et al. developed Law-Governed Interaction [9], [8] to govern the communication between a group of nodes by a unified group policy. By implementing policy enforcement through middleware controllers, LGI can support more flexible and complicated policies, which can not be done in previous enforcement mechanisms. Our method is similar to LGI in that the policy enforcement is performed in the entire network. Moreover, our method supports any policy that can be expressed in the law of LGI.

The major difference between our method and LGI is the lack of the dependence on pre-existing trusted nodes. LGI requires that controllers be trusted but does not provide means of establishing the trust. Consequently, in practice, it can only be applicable in controlled environments where the enforcers can be deployed or elected, such as corporate intranet [10], [6] and Internet P2P [11]. Our method enables trust to be built dynamically based on node's trustworthiness of enforcing the policy without assuming pre-existing trust relationships between nodes. As a result, it is suitable for uncontrolled environment, such as spontaneous ad hoc networks. In addition, instead of just supporting policies defined for a specific protocol, our multi-tier network can enforce more complex policies defined across various interdependent protocols.

Sailer et al. [49] developed a TPM based fine grained policy enforcement mechanism for corporations to control their VPN clients. Compared with our work, the method relies on the fixed infrastructure and a trusted policy owner and distributor, which is the corporate VPN server. Neither exists in ad hoc computing.

Trusted Computing. Both hardware and software based methods have been proposed to ensure trusted software execution. In the hardware approach, the trusted software is executed on a high-end trusted processor or co-processor such as IBM 4758 [50], Citadel [51], Dyad [52], Cerium [53], and XOM [54]. Due to the high cost, the hardware is unlikely to be ubiquitously deployed, which makes these solutions unsuitable for ordinary nodes in ad hoc networks.

The pure software methods, such as [55], SWATT [56], and Pioneer [57], challenge the target system with the requirement of computing a checksum within a certain time limit. These methods assume sufficient knowledge about target system's hardware and a noticeable delay of forging the checksum. Neither of them holds in ad hoc networks computing.

Terra [58], Microsoft NGSCB [59], IBM TCGLinux [60], and Bind [61] leverage a low-end trusted hardware like TCG TPM [16] to boost trust on a set of software components, which further ensures trustworthiness of the execution of target programs. Our method differs from them in the scope and persistence of protection. First, it ensures that all the code the target programs load and depend on is trusted. This property enables our method to catch every attempt to compromise the execution of the protected programs and be immune to false positives, since irrelevant changes in the system will not be monitored. Second, it guarantees the trustworthiness of the target programs not only at the time of integrity measurement, but also for future executions. This is done through the commitment protocol.

IX. Acknowledgment

X. Conclusion

This paper presented a mechanism for MANETs to enforce application communication policies. Under this mechanism, nodes supporting the same set of applications and enforcing the same policies construct a trusted multi-tier application-centric network. Each tier of the network runs one application and enforces its associated policy. The application of the upper tier depends on the applications of the lower tiers to communicate. Only trusted nodes are allowed to join the network. Moreover, communication between them is regulated by the policies at every tier. To ensure trusted policy enforcement, we augment each node with a trusted kernel agent based on the TCG TPM. We evaluated the method through a prototype based on an IEEE 802.11 ad hoc network and through network simulations. The results demonstrate the feasibility of the proposed method as well as its low overhead.

References

[1] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management," in *the Proceedings of IEEE Conference on Privacy and Security*, 1996, pp. 164–173.

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis, "The keynote trust-management system, version 2," in *RFC 2704*, September 1999.

[3] G. Karjoth, "The authorization service of tivoli policy director," in *the Proceedings of the 17th Computer Security Applications Conference (ACSAC)*, December 2001, p. 319.

[4] T. Woo and S. Lam, "A framework for distributed authorization," in *the Proceedings of the 1st ACM Conference on Computer and Communications Security*, November 1993, pp. 112–118.

[5] S. Ioannidis, A. Keromytis, S. Bellovin, and J. Smith, "Implementing a distributed firewall," in *the Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2000, pp. 190–199.

[6] T. Phan, Z. He, and T. D. Nguyen, "Using firewalls to enforce enterprise-wide policies over standard client-server interactions," in *Journal of Computers (JCP)*, April 2006, vol. 1, no. 1, pp. 1–12.

[7] S. Capkun, J. Hubaux, and L. uttyán, "Mobility helps security in ad hoc networks," in *the Proceedings of the ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC)*, June 2003, pp. 46–56.

[8] N. Minsky and V. Ungureanu, "Unified support for heterogeneous security policies in distributed systems," in *the Proceedings of 7th USENIX Security Symposium*, January 1998, p. 10.

[9] N. Minsky and V. Ungureanu, "Law-governed interaction: A coordination & control mechanism for heterogeneous distributed systems," in *ACM Transactions on Software Engineering and Methodology (TOSEM)*, July 2000, vol. 9, no. 3, pp. 273–305.

[10] T. Murata and N. Minsky, "Regulating work in digital enterprises: A flexible managerial framework," in *the Proceedings of the Cooperative Information Systems Conference (CoopIS)*, October 2002, pp. 356–372.

[11] N. M. Mihail Ionescu and T. Nguyen, "Enforcement of communal policies for peer-to-peer systems," in *the Proceedings of 6th International Conference on Coordination Models and Languages*, February 2004.

[12] J. M. McCune, T. Jaeger, S. Berger, R. Caceres, and R. Sailer, "Shamon: A system for distributed mandatory access control," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006.

[13] S. L. Keoh, E. Lupu, and M. Sloman, "Peace: A policy-based establishment of ad-hoc communities," in *the Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, September 2004, pp. 386–395.

[14] G. Xu, C. Borcea, and L. Iftode, "Trusted application-centric ad-hoc networks," in *the Proceedings of the 4th IEEE International Conference on Mobile Ad-hoc Networks and Sensor Systems (MASS 2007)*, 2007.

[15] G. Xu, C. Borcea, and L. Iftode, "Satem: A Service-aware Attestation Method Toward Trusted Service Transaction," in *the Proceedings of IEEE Symposium on Reliable Distributed Systems (SRDS)*, October 2006, pp. 321–336.

[16] Trusted Computing Group, "TCG 1.1b Specifications. https://www.trustedcomputinggroup.org/home."

[17] Trusted Computing Group - Mobile Phone Working Group, "Use Case Scenarios - v 2.7."

[18] "The Network Simulator - NS2," http://www.isi.edu/nsnam/ns.

[19] C. E. Perkins and E. M. Royer, "Ad hoc On-Demand Distance Vector Routing," in *the Proceedings of 2nd IEEE Workshop on Mobile Computing Systems and Applications*, 1999.

[20] Y.-C. Hu, A. Perrig, and D. B. Johnson, "Packet leashes: A defense against wormhole attacks in wireless ad hoc networks," in *the Proceedings of Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies*, April 2003, pp. 1976– 1986.

[21] S. Dashtinezhad, T. Nadeem, B. Dorohonceanu, C. Borcea, P. Kang, and L. Iftode, "Trafficview: A driver assistant device for traffic monitoring based on car-tocar communication," in *the Proceedings of the 59th IEEE Semiannual Vehicular Technology Conference*, May 2004, pp. 2946– 2950.

[22] D. F. Brewer and M. J. Nash, "The chinese wall security policy," in *the Proceedings of IEEE Conference on Privacy and Security*, 1989, pp. 206 – 214.

[23] J. R. Douceur, "The sybil attack," in *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*. London, UK: Springer-Verlag, 2002, pp. 251–260.

[24] Http://mute-net.sourceforge.net/.

[25] S. Berkovits, S. Chokhani, J. Furlong, J. Geiter, and J. Guild, "Public key infrastructure study final report," *MITRE report*, 1994.

[26] W. Bagga, S. Crosta, P. Michiardi, and R. Molva, "Establishment of ad-hoc communities through policy-based cryptography," in *Electronic Notes in Theoretical Computer Science*, 2007, vol. 171, no. 1, pp. 107–120.

[27] K. Goldman, R. Perez, and R. Sailer, "Linking remote attestation to secure tunnel endpoints," in *Proceedings of the first ACM workshop on Scalable trusted computing*, 2006.

[28] L. Zhou and Z. J. Haas, "Securing ad hoc networks," in *IEEE Networks*, Nov-Dec 1999, vol. 13, no. 6, pp. 24–30.

[29] N. Asokan and P. Ginzboorg, "Key agreement in ad-hoc networks," in *Computer Communications*, 2000, vol. 23, no. 17, pp. 1627–1637.

[30] J. Kong, P. Zerfos, H. Luo, S. Lu, and L. Zhang, "Providing robust and ubiquitous security support for mobile ad-hoc networks," in *the Proceedings of the 9th IEEE International Conference on Network Protocols (ICNP'01)*, 2001, p. 251.

[31] S. Capkun, L. Buttyan, and J. P. Hubaux, "Self-organized public-key management for mobile ad hoc networks," Jan-Mar 2003, vol. 2, no. 1, pp. 52–64.

[32] G. Xu and L. Iftode, "Locality driven key management for mobile ad-hoc networks," in *the Proceedings of the 1th IEEE International Conference on Mobile Ad-hoc Networks and Sensor Systems (MASS 2004)*, 2004, pp. 436– 446.

[33] Http://core.it.uu.se/core/index.php/AODV-UU.

[34] X. Wang, Y. Yin, and H. Yu, "Finding collisions in the full SHA1," in *the Proceedings of Crypto*, 2005.

[35] "The netfilter/iptables Project," http://www.netfilter.org.

[36] "Atmel TPM," http://www.atmel.com/.

[37] J. Nzouonta, N. Rajgure, G. Wang, and C. Borcea, "Vanet routing on city roads using real-time vehicular traffic information," in *IEEE Transactions on Vehicular Technology*, 2009, vol. 58, no. 7, pp. 3609–3626.

[38] J. Broch, D. A. Maltz, D. Johnson, Y. Hu, and J. Jetcheva, "A performance comparison of multi-hop wireless ad hoc network routing protocols," in *the Proceedings of the 4th annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 1998)*, 1998, pp. 85–97.

[39] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu, "The broadcast storm problem in a mobile ad hoc network," in *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, 1999.

[40] B. Kauer, "Oslo: improving the security of trusted computing," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[41] AMD, "Secure virtual machine architecture reference manual," 2005.

[42] A. Baliga, P. Kamat, and L. Iftode, "Lurking in the shadows: Identifying systemic threats to kernel data," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2007.

[43] L. S. Clair, J. Schiffman, T. Jaeger, and P. McDaniel, "Establishing and sustaining system integrity via root of trust installation," in *Proceedings of the 23nd Annual Computer Security Applications Conference*, 2007.

[44] R. Anderson, "Technical perspective a chilly aense of security," *Communication of ACM*, vol. 52, no. 5, pp. 90–90, 2009.

[45] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *the Proceedings of the 17th USENIX Security Syposium*, 2008.

[46] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *Proceedings of the 11th USENIX Security Symposium*, 2002.

[47] P. Dinsmore, D. Balenson, M. Heyman, P. Kruus, C. Scace, and A. Sherman, "Policy-based security management for large dynamic groups: An overview of the dccm project," in *the Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX́00)*, January 2000, pp. 64 – 73.

[48] P. McDaniel and A. Prakash, "Enforcing provisioning and authorization policy in the antigone system," in *Journal of Computers (JCP)*, November 2006, vol. 14, no. 6, pp. 483–511.

[49] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn, "Attestation-based policy enforcement for remote access," in *the Proceedings of 11th ACM Conference on Computer and Communications Security*, 2004, pp. 308–317.

[50] S. W. Smith and S. H. Weingart, "Building a high performance, programmable secure co-processor," in *Computer Networks (Special Issue on Computer Network Security)*, April 1999, vol. 31, no. 9, pp. 831–860.

[51] S. White, S. Weingart, W. Arnold, and E. Palmer, "Introduction to the Citadel architecture: Security in physically exposed environments," IBM Thomas J. Watson Research Center, Tech. Rep. TR RC16672, 1991.

[52] B. Yee, "Using secure coprocessors," Ph.D. dissertation, May 1994.

[53] B. Chen and R. Morris, "Certifying program execution with secure processors," in *the Proceedings of 9th Workshop on Hot Topics in Operating Systems*, 2003, pp. 23–23.

[54] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *the Proceedings of Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 168–177.

[55] R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *the Proceedings of 12th USENIX Security Symposium*, 2003, pp. 21–21.

[56] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT:Software-based attestation for embedded devices," in *the Proceedings of 2004 IEEE Symposium on Security and Privacy*, 2004.

[57] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. V. Doorn, and P. Khosla, "Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms," in *the Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[58] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *the Proceedings of the 9th ACM Symposium on Operating Systems Principles*, 2003, pp. 193–206.

[59] Microsoft Corp., "Next generation secure computing base," http://www.microsoft.com/resources/ngscb.

[60] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *the Proceedings of 13th USENIX Security Symposium*, 2004, p. 16.

[61] E. Shi, A. Perrig, and L. van Doorn, "Bind: A time-of-use attestation service for secure distributed system," in *the Proceedings of IEEE Symposium on Security and Privacy*, 2005, pp. 154–168.