Graph-Based Verification and Misbehavior Detection in Multi-Agent Networks

Phillip Lee[§], Omar Saleh[†], Basel Alomair[†], Linda Bushnell[§], Radha Poovendran[§]
[§]Department of Electrical Engineering, University of Washington, Seattle, WA 98195
[†] King Abdulaziz City for Science and Technology, Riyadh, Saudi Arabia
Email: leep3@uw.edu, {osaleh, alomair}@kacst.edu.sa, {lb2, rp3}@uw.edu *

ABSTRACT

Multi-agent networks consist of autonomous nodes, where each node maintains and updates its state based on exchanged information with its neighboring nodes. Due to the collaborative nature of state updates, if one or more nodes were to misbehave by deviating from the pre-specified update rule, they can bias the states of other nodes and thus drive the network to an undesirable state. In this paper, we present a query-based mechanism for a third-party verifier to detect misbehaving nodes. The proposed mechanism consists of two components. The first component determines whether the state of the queried node is consistent with its ideal value. The second component identifies the set of misbehaving nodes that induced the inconsistency. We prove that our approach detects the set of misbehaving nodes, as well as the times of their misbehaviors, by establishing the equivalence of our approach to a tree-generation algorithm. We evaluate our approach through simulation study which corroborates the theoretical guarantees, and analyzes the performance of our scheme as a function of the number of queried nodes.

Categories and Subject Descriptors

C.2.0 [**General**]: Security and protection

General Terms

Security

Keywords

security; misbehavior; multi-agent network; graph-based verification

1. INTRODUCTION

Formation flight [15], environmental monitoring [1] and wide-area surveillance [18] are applications where networked nodes collaborate in order to perform maneuvering, sensing and computation. This collaboration is typically achieved by having each node maintain an internal state, which is updated according to a pre-specified rule, based on information exchanged with the node's neighbors [14]. The objective of the system can be quantified as a desired set of node states to be achieved. Under this approach, the update and broadcast of individual node state forms part of the network control signals that are required for coordination of network missions.

Since the state update relies on inputs provided by neighboring nodes, a set of misbehaving nodes can bias the state updates of their neighbors by broadcasting incorrect state values. A node could be misbehaving because it is faulty or malicious [17]. These errors in the node states introduced by misbehaving nodes will propagate through the network via the local update. Detecting such misbehaving nodes has received significant attention from the control and fault tolerant computing literatures [9, 10, 17, 20].

Existing methods consider detection of misbehaving nodes at a local level [17, 20] by the nodes themselves. However, in applications such as formation flight of unmanned vehicles [11] and network consensus [7, 19] , there exists an external entity with knowledge of the deployed network (initial states, state dynamics, and network topology) that can communicate with a subset of nodes at any given time and hence can participate in the detection process. An example is leader-follower systems [11] where the set of leader nodes receive control inputs from the external entity, and act as actuators that drive the system to a desired operating point.

In this paper, we study the problem of detecting misbehaving nodes in a multi-agent network in the presence of a third-party verifier. Under our approach, each node cryptographically signs its state update messages, which are stored by the neighboring nodes. We propose a detection mechanism in which a third party verifier queries the subset of network nodes and receives the signed messages stored by the queried nodes. In developing this detection mechanism, we make the following specific contributions:

- We develop a query-based mechanism for a third-party verifier to detect misbehaving nodes. The proposed mechanism consists of two components. In the first component, the verifier determines whether the nodes are updating their states correctly by comparing the queried states to the ideal states. Upon detection of inconsistent states, the second component identifies the set of misbehaving nodes that are inducing the inconsistency.
- We classify three types of disjoint events that could lead to an inconsistent state of a node: the event where the node computes its state according to the correct update rule, but receives inconsistent inputs; the event where the node computes the update rule incorrectly; and the case where both of these events occur. We present an algorithm that identifies which type of event caused an inconsistent state.

*This work was supported by a grant from the King Abdulaziz City for Science and Technology (KACST).

HiCoNS'14, April 15–17, 2014, Berlin, Germany.

http://dx.doi.org/10.1145/2566468.2566477.

- We develop a decision-tree growing algorithm that identifies misbehaving nodes and times of their misbehaviors by systematically searching for sources of inconsistency based on the identified type of event. Under the assumption that misbehaving nodes do not share cryptographic information (non-colluding), we show the proposed tree growing algorithm detects and identifies misbehaving nodes that are inducing inconsistent states of nodes.

The rest of the paper is organized as follows. In Section 2, we review relevant related work on detection of misbehaving nodes in multi-agent networks. In Section 3, we state the definitions and assumptions used in this paper. In Section 4, we formulate the problem of detecting and identifying misbehaving nodes in multi-agent networks and propose a query-based detection mechanism. The effectiveness of the proposed detection mechanism is then proved. In Section 5, we present a simulation to illustrate the effectiveness of our approach. Section 6 presents our conclusions and future work.

2. RELATED WORK

Ensuring reliable performance of a distributed system in the presence of faulty and malicious components has been studied in the context of Byzantine failures in distributed computing [13], fault-tolerant control of dynamic systems [4], and secure routing in networking [2]. The Byzantine failure model in distributed computing assumes a set of nodes (processors) can fail in arbitrary ways while executing an algorithm in a distributed manner. A successful termination of an algorithm requires a bound on the number of failed nodes [13]. Specifically, if f nodes can fail simultaneously, at least $(2f+1)$ nodes must be deployed in the network.

An analogous result was proved in the context of distributed consensus in multi-agent networks [20] where each node updates its state using linear iteration to obtain correct initial states of other nodes. In these works, the state dynamics is represented as a linear dynamical system, and incorrect state values of malicious nodes as unknown inputs. It was proved that if f nodes are malicious in the network, a graph connectivity of at least $(2f+1)$ is required for every node to agree on the initial states of other nodes. This work is based on the theory of unknown input observer (UIO) [5, 6, 8]. Moreover, it was also shown that once a node has obtained correct initial states of other nodes, it can detect and identify malicious nodes in the network given the full network topology and state update dynamics of all other nodes. However, this approach requires that each node has the knowledge of the entire network topology and state dynamics, and is computationally hard for each node due to combinatorial search.

To overcome this difficulty, a detection scheme based on decentralized UIO was introduced in [17] where a set of nodes cooperatively detect misbehaving nodes. Similar approach was also taken in [16] where the authors designed a distributed intrusion detection system based on UIO. Malicious nodes in wireless control networks was studied in [21], where the authors proved that if the connectivity of the wireless network is sufficiently high, it suffices to observe only a set of nodes in the network to jointly diagnose faults in the plant and detect malicious nodes.

A related body of work that focuses on providing resilience of multi-agent networks against misbehaving nodes has been explored in [12,22]. These approaches are based on local filtering techniques where a node chooses from the set of received inputs to update its state. In [12], a low complexity algorithm based on local filtering was introduced where a node removes f largest and smallest inputs. A sufficient condition for consensus was given as a function of the in-and out degree distributions. A consensus protocol based on similar local filtering technique was studied in [22]. The authors introduce the notion of r-robustness, and characterize conditions in terms of r-robustness to guarantee consensus when there are at most r malicious nodes in the neighborhood of any node. However, these works do not consider proactive detection and identification of misbehaving nodes.

Existing works utilize mathematical structures of network dynamics and topology of the network to provide detection and mitigation strategies at a local level. These methods, however, do not incorporate the available cryptographic primitives to facilitate detection, and do not leverage the presence of external entities that have knowledge of the network dynamics.

In our view, our work is complementary to the existing methods in that we employ cryptographic mechanisms to detect and identify misbehaving nodes by an external entity. Therefore, our proposed methods may be used in conjunction with the existing methods.

Table 1: A summary of notations used

Symbol	Definition
n_i	Identity of node i
$\mathcal{V}$	Third party verifier
$\mathcal{N}(n_i)$	Set of neighboring nodes of node i
M	Time slot of inspection
Q	Set of queried nodes
$x_{n_i}(m)$	State of node i at time m
$X(m)$	$\{x_{n_i}(m) \forall n_i\}$
$X_{\mathcal{N}(n_i)}(m)$	$\{x_{n_j}(m) n_j \in \mathcal{N}(n_i)\}$
$\hat{x}_{n_i}(m)$	Ideal state of node i at time m
$f_i(\cdot)$	Node n_i's state update function.
$IN_{n_i}(m)$	Set of input values for node n_i's state update at time m
$\widehat{IN}_{n_i}(m)$	Set of ideal input values for node n_i's state update at time m
$\widetilde{IN}_{n_i}(m)$	Elements in $IN_{n_i}(m)$ that are inconsistent
$ID(x)$	ID of the node that transmitted message x
K_{n_i}	Public key assigned to node n_i
$K_{n_i}^{-1}$	Private key assigned to node n_i
$sig_{K_{n_i}^{-1}}(x)$	Message x signed with private key $K_{n_i}^{-1}$

3. PRELIMINARIES

In this section, we present models of the network, state update dynamics, misbehaving nodes and third party verifier. Table 1 lists the notations used in this paper.

3.1 Network Model

We consider a homogeneous network with N nodes where each node communicates with a set of neighboring nodes $\mathcal{N}(n_i)$, which is assumed to be time-invariant for all nodes n_i. We assume a broadcasting communication model where each node transmits its message to all its neighbors simultaneously. A suitable media access control (MAC) protocol [3] is assumed to be implemented so that whenever a node broadcasts its message, it will be received by all its neighboring nodes. We assume the deployed network is connected, i.e., there exists at least one path between any two nodes in the network.

Each node n_i is assigned a unique public and private key pair $(K_{n_i}, K_{n_i}^{-1})$. Every node stores its own private key and public keys of its neighboring nodes. Each node is also required to store all messages received from its neighbors in the previous M time slots.

3.2 State Update Dynamics

We consider a discrete-time multi-agent network where time is slotted into intervals of equal duration. It is assumed that each node is aware of the beginning and end of each time slot.

During mth time slot, each node n_i is responsible for updating its state $x_{n_i}(m)$ to $x_{n_i}(m+1)$ before the beginning of $(m+1)$th time slot in a specified manner. To update its state during mth time slot, each node n_i first constructs a message by concatenating its ID, current time slot, and its current state. Node n_i then digitally signs the constructed message with its private key, attaches the signature to the constructed message, and broadcasts the message to its neighboring nodes. In addition, after receiving signed messages from its neighbors, node n_i verifies validity of received messages by verifying signatures using stored public keys of neighbors.

If the verification test is successful, then node n_i accepts the message as valid. If not, node n_i discards the message. If node n_i does not receive a valid message from node n_j during the mth time slot, node n_i sets $x_{n_j}(m) =$ Null. It is assumed that Null is a valid input parameter for the function $f_i(\cdot)$, but not a valid output. Each node is responsible for broadcasting its state to its neighbors for all time slots.

After verification, node n_i extracts the current states of neighbors, $X_{\mathcal{N}(i)}(m) = \{x_{n_j}(m) | n_j \in \mathcal{N}(n_i)\}$, and updates its state with its current state and current states of neighbors. The update process of node n_i during mth time slot is described as below.

1. n_i constructs the message described as above and broadcasts the message to its neighbors.

$$n_i \rightarrow \mathcal{N}(n_i) : \left(n_i||m||x_{n_i}(m)||sig_{K_{n_i}^{-1}}(n_i||m||x_{n_i}(m))\right)$$

2. n_i verifies validity of the message received from node n_j. If valid, accept the message. If not, discard. If node n_i does not receive a valid message from node n_j, record n_j's state as Null.

3. n_i extracts $X_{\mathcal{N}(i)}(m) = \{x_{n_j}(m) | n_j \in \mathcal{N}(n_i)\}$, current state of neighbors that are valid from received messages.

4. Update its state using the specified update function $f_i(\cdot)$

$$x_{n_i}(m+1) = f_i(X_{\mathcal{N}(n_i)}(m), x_{n_i}(m), Nulls)$$

The set of input values node n_i uses to update its state during time slot m is denoted as $IN_{n_i}(m) = \{X_{\mathcal{N}(n_i)}(m-1)\}, x_{n_i}(m-1)\}$.

3.3 Adversary Model

We say that a node n_i is misbehaving if it broadcasted its initial state falsely during time slot $m = 0$ or if it computed its updated function $f_i(\cdot)$ incorrectly for at least one time slot $m \geq 0$. Mathematically, node n_i is misbehaving if

$$x_{n_i}(m+1) \neq f_i(X_{\mathcal{N}(n_i)}(m), x_{n_i}(m))$$

for at least one time slot $m \geq 0$. The set of misbehaving nodes is classified into two categories: faulty and malicious. We assume that malicious nodes are capable of following behaviors. First, when asked by a third party verifier to give its current or previous states, a malicious node is capable of returning false states. Second, while a faulty node always broadcasts its current state truthfully to its neighbors, a malicious node is capable of broadcasting its state untruthfully. We assume, however, that malicious nodes are not colluding with each other. Specifically, each malicious node is unaware of the identity or private key of any other malicious node.

We also assume that there exists at least one node that is not malicious in $\mathcal{N}(n_i)$ for every node n_i that is capable of returning correct message to the third party verifier at the time of query.

3.4 Third Party Verification Model

A third party verifier who is aware of the initial states of nodes, and the set of specified update functions can generate ideal states of nodes at any given time slot. The table of ideal states of every node till the current time slot is referred to as the truth table. In what follows, the third party verifier will be referred to as the verifier and is denoted as $\mathcal{V}$.

The verifier can be the network owner if he is within a communication range of at least one node in the network. If not, the network owner relegates the responsibility of the verifier to a trusted third party that is within the communication range of at least one node. In order to relegate, the network owner sends the third party its truth table, set of update functions for every node, and public keys associated with each node.

Given the ideal state of node n_i at time m, $\hat{x}_{n_i}(m)$, the verifier is capable of checking whether the actual state is within an acceptable bound. The state of node n_i at time m is defined to be inconsistent if

$$|\hat{x}_{n_i}(m) - x_{n_i}(m)| \geq \epsilon(n_i, m)$$

where $\epsilon(n_i, m) > 0$ is a predefined threshold. $\epsilon(n_i, m)$ is chosen based on how much deviation the verifier is willing to tolerate for node n_i at time m. In this paper, a Null state is classified as an inconsistent state. For example, consider a distributed averaging update dynamics given as

$$X(m+1) = WX(m).$$

In such scenario, the network may stop further linear iteration when every node reaches a value that is within some ϵ range of the true average to save further communication and computation overhead. Furthermore, it is known [14] that the linear update dynamics for distributed averaging is marginally stable, i.e., matrix W has one eigenvalue equal to 1 and all other eigenvalues have magnitude less than 1. Moreover, the eigenvector that is associated with the eigenvalue 1 has same values in all entries. This implies that, as long as $|\hat{x}_{n_i}(m) - x_{n_i}(m)| < \epsilon$, for all n_i, the deviation on the $m+1$th time slot is bounded as $|\hat{x}_{n_i}(m+1) - x_{n_i}(m+1)| < \epsilon$. Such deviations can be tolerated since the state of every node will converge to a value that lies within ϵ of the average of the initial states.

This notion of tolerable deviation from the ideal behavior can be generalized in the following way. The verifier can choose $\epsilon(n_i, m)$ such that if all inputs were consistent at time m, and $f_i(\cdot)$ was computed correctly, then $x_{n_i}(m+1)$ would also be consistent. A sufficient condition for the existence of such ϵ values is given in the following lemma.

LEMMA 3.1. *If the update function $f_i(\cdot)$ for each node n_i is continuous around the ideal input states for all time slots $m \leq M$, then $\epsilon(n_i, m) > 0$ values can be constructed such that if $|\hat{x}_{n_j}(m-1) - x_{n_j}(m-1)| < \epsilon(n_j, m-1)$ for all $n_j \in \mathcal{N}(n_i)$, and $|\hat{x}_{n_i}(m-1) - x_{n_i}(m-1)| < \epsilon(n_i, m-1)$, then $|\hat{x}_{n_i}(m) - x_{n_i}(m)| < \epsilon(n_i, m)$ for all time $m \leq M$.*

PROOF. Choose $\epsilon(n_i, M) > 0$. From the definition of continuity, there exists $\epsilon(n_j, M-1)$ for $n_j \in \mathcal{N}(n_i)$ and $\epsilon(n_i, M-1)$ such that if $|\hat{x}_{n_j}(M-1) - x_{n_j}(M-1)| < \epsilon(n_j, M-1)$ and $|\hat{x}_{n_j}(M-1) - x_{n_i}(M-1)| < \epsilon(n_i, M-1)$, then $|\hat{x}_{n_i}(M) - x_{n_i}(M)| < \epsilon(n_i, M)$. The same argument can be made for time slot $M-1$. Therefore, $\epsilon(n_i, m)$ values can be constructed recursively starting from any time slot $M > 0$. □

4. QUERY-BASED DETECTION

Under our approach, the external verifier periodically checks whether the network states are consistent. Further inspections are made as needed to determine the source of any inconsistencies.

We propose a detection mechanism where the verifier queries the stored state information in nodes to detect inconsistent states and identify misbehaving nodes. The detection process starts at the beginning of time slot M. The verifier selects a set of nodes, Q to be queried. Each node in Q is queried to return the messages it received from its neighbors during the previous time slot. If no inconsistent states are found, then the detection process ends. For each inconsistent state found during time slot $M-1$, further inspections are required to identify the set of misbehaving nodes that induced the inconsistent state at time $M-1$. The following two subsections describe the query-based algorithms that are building blocks of the detection mechanism.

4.1 Determining Cause of Inconsistency

In Section 3.4, we defined the notion of consistent state such that if all input states were consistent, and update function was computed correctly, then the resulting updated state is also consistent. Hence, if $x_{n_i}(m)$ was inconsistent for some time slot $m \geq 1$, it would be due to the occurrence of one of the following events. We label these different types of events as E_1, E_2, E_3.

- E_1: Node n_i computed $f_i(\cdot)$ correctly but one or more elements of the set $IN_{n_i}(m-1)$ were inconsistent.
- E_2: Node n_i computed $f_i(\cdot)$ incorrectly during the $(m-1)$th time slot but every element of $IN_i(m-1)$ was consistent.
- E_3: Node n_i computed $f_i(\cdot)$ incorrectly during the $(m-1)$th time slot and one or more elements of $IN_{n_i}(m-1)$ were inconsistent.

During the initial time slot $m=0$, if the broadcasted state $x_{n_i}(0)$ was inconsistent, we label this event as type E_2. A illustration of the three types of events is shown in Figure 1.

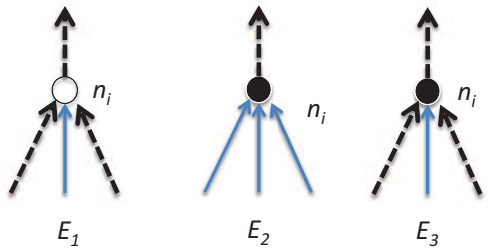

Figure 1: Illustration of the three types of events that could result in inconsistent state of node n_i. Dashed lines indicate inconsistent states. Filled circles indicate incorrect computation of the update function. (a) E_1: Node n_i computes the function $f_i(\cdot)$ correctly but resulting state is inconsistent due to inconsistent inputs. (b) E_2: All inputs are consistent but node n_i computes $f_i(\cdot)$ incorrectly. (c) E_3: Node n_i computes $f_i(\cdot)$ incorrectly and one or more inputs are inconsistent.

Determining which type of event induced inconsistency of a given state is the first step of the detection process. We propose an algorithm that determines which type of event led to a given inconsistent state.

Given an inconsistent state $x_{n_i}(m)$, the verifier must first reconstruct the input values node n_i has used during time slot $m-1$ to update its state. To do so, The verifier sends a request to node n_j for the messages received from node n_i during time slot $(m-1)$. The verifier also requests to node n_i for the messages n_i received from its neighbors during time slot $(m-1)$. Since we assume that malicious nodes are capable of responding to queries with incorrect state values, no node is queried to return its own state.

Both n_i and n_j sign the requested messages with their respective private keys and send the requested messages back to the verifier. After having received requested messages, the verifier authenticates the received messages via the attached digital signatures and extracts $X_{n_i}(m-1)$ and $X_{\mathcal{N}(n_i)}(m-1)$ from the messages. A pseudocode description of the algorithm to construct $IN_{n_i}(m-1)$ is given as algorithm **ConstructIN**.

Algorithm ConstructIN	
Input: $x_{n_i}(m)$	
Output: $IN_{n_i}(m-1)$	
Query $n_j \in \mathcal{N}(n_i)$	1
$n_j \rightarrow \mathcal{V}$: signed message n_i sent to n_j or Null	2
Query n_i	3
$n_i \rightarrow \mathcal{V}$: signed message n_i received from $\mathcal{N}(n_i)$ or Nulls	4
Return $IN_{n_i}(m-1) = \{X_{\mathcal{N}(n_i)}(m-1), x_{n_i}(m-1)\}$	5

Figure 2: Algorithm for reconstructing input values node n_i has used to update its state during time slot $(m-1)$

After constructing the input values $IN_{n_i}(m-1)$ using the algorithm ConstructIN, the next step for the verifier is to determine which type of event led to the inconsistent state $x_{n_i}(m)$. There are two separate cases to be considered.

Case 1. $IN_{n_i}(m-1)$ did not contain a Null message: In the case when $IN_{n_i}(m-1)$ did not contain a Null message, the verifier computes $\bar{x}_{n_i}(m) = f_i(IN_{n_i}(m-1))$ using the returned output values from ConstructIN, $IN_{n_i}(m-1)$. If $\bar{x}_{n_i}(m) = x_{n_i}(m)$, then the verifier concludes that event type E_1 has occurred since the function was computed correctly at time slot $(m-1)$. If $\bar{x}_{n_i}(m) \neq x_{n_i}(m)$, then the verifier needs to distinguish between E_2 and E_3.

The verifier collects inconsistent elements from the set $IN_{n_i}(m-1)$ by comparing the values with the truth table and constructs a subset $\widetilde{IN}_{n_i}(m-1)$. If $\widetilde{IN}_{n_i}(m-1) = \emptyset$, (there was no inconsistent element in $IN_{n_i}(m-1)$) then E_2 must have occurred. If there is an inconsistent element $\widetilde{IN}_{n_i}(m-1) \neq \emptyset$, then E_3 has occurred.

Case 2. $IN_{n_i}(m-1)$ contained at least one Null message: Unlike a valid signed state which cannot be forged, any node can falsely claim that it did not receive any message from its neighbors at a given time slot, since Null message does not have any signature attached to it by nature. If n_i claims that it did not receive any message from its neighbor n_j at time slot $(m-1)$, the verifier takes the following steps to determine whether node n_i is falsely reporting or not. First, the verifier requests to every node in $\mathcal{N}(n_j)$ for the message it received from node n_j at time slot $(m-1)$.

If at least one node in $\mathcal{N}(n_j)$ returns a valid signed message from node n_j during time slot $(m-1)$, then the verifier concludes that node n_i is falsely claiming that it did not receive any input state from n_j. Consequently, n_i is identified as a misbehaving node for ignoring a received input. The verifier now needs to distinguish between event types E_2 and E_3. To do so, the verifier reconstructs the message $IN_{n_i}(m-1)$ by replacing a falsely claimed Null with the signed message of n_j and check whether newly constructed $IN_{n_i}(m-1)$ has contained any inconsistent element. If $\widetilde{IN}_{n_i}(m-1) = \emptyset$, then E_2 has occurred, if not E_3 has occurred.

On the other hand, if all nodes in $\mathcal{N}(n_j)$ claim that no message was transmitted from n_j at time slot $(m-1)$, or if n_i is the only neighboring node of n_j, then it is concluded that n_j did not transmit any message during time slot $m-1$. Since Null state is classified as an inconsistent state by convention, the verifier now needs to distinguish between event types E_1 and E_3. The verifier computes $\bar{x}_{n_i}(m) = f_i(IN_{n_i}(m-1))$. If $\bar{x}_{n_i}(m) = x_{n_i}(m)$, then verifier concludes E_1 has occurred. If $\bar{x}_{n_i}(m) \neq x_{n_i}(m)$, then E_3 has occurred. A pseudocode description of the algorithm is given as algorithm **DecideType**.

Algorithm DecideType

Input: Inconsistent state $x_{n_i}(m)$
Output: Type of Event
Initialization:
$IN_{n_i}(m-1) \leftarrow$ ConstructIN($x_{n_i}(m)$) 1
if Null $\notin IN_{n_i}(m-1)$ 2
$\bar{x}_{n_i}(m) \leftarrow f_i(IN_{n_i}(m-1))$ 3
  if $\bar{x}_{n_i}(m) = x_{n_i}(m)$ 4
    return E_1; **exit** 5
  else 6
    Construct $\widetilde{IN}_{n_i}(m-1)$
    if $\widetilde{IN}_{n_i}(m-1) = \emptyset$ 7
      return E_2; **exit** 8
    else 9
      return E_3; **exit** 10
  end 11
else Null $\in IN_{n_i}(m-1)$ 12
For all n_j such that $x_{n_j}(m-1) =$ Null 13
Request to all $n_k \in \mathcal{N}(n_j)$ for $x_{n_j}(m-1)$ 14
 if exists at least one n_k such that 15
 $n_k \rightarrow \mathcal{V}$: signed message n_j sent to n_k during time $(m-1)$ 16
 Replace Null with $x_{n_j}(m-1)$ in $IN_{n_i}(m-1)$ 17
 Construct $\widetilde{IN}_{n_i}(m-1)$ 18
   if $\widetilde{IN}_{n_i}(m-1) = \emptyset$ 19
     return E_2; **exit** 20
   else 21
     return E_3; **exit** 22
 else 23
  $\bar{x}_{n_i}(m) \leftarrow f_i(IN_{n_i}(m-1))$ 24
   if $\bar{x}_{n_i}(m) = x_{n_i}(m)$ 25
     return E_1; **exit** 26
   else 27
     return E_3; **exit** 28

Figure 3: Algorithm for determining which event led to inconsistent state $x_{n_i}(m)$

The algorithm DecideType cannot be executed for an inconsistent state $x_{n_i}(m)$ at time slot m if the necessary data required to construct $IN_{n_i}(m-1)$ is not stored in any nodes. Since each node is required to store all messages it received from its neighbors in the previous M time slots, DecideType can be executed for any inconsistent state during time slots $m \in \{1...M-1\}$. Using the algorithm DecideType as the building block, we now proceed to describe the main algorithm which detects all misbehaving nodes that induced an inconsistent state at time slot $M-1$.

4.2 Identifying Misbehaving Nodes

The intuition behind the proposed algorithm is the following: Given an inconsistent node state, the cause for the inconsistency is either E_2 or other types of events (E_1 or E_3). If the event type is E_2, then the node's misbehavior during the previous time slot was the sole cause of inconsistency. If the type is not E_2, then there existed at least one inconsistent state used by the node to update its state during the previous time slot, and the same argument can be made for each inconsistent state during the previous time slot.

This detection process is represented as a tree-generation algorithm. The output of the algorithm is a tree $\mathcal{T}$ with root vertex $x_{n_i}(M-1)$ and maximum depth of M. Vertices at depth d (denoted as V_d) represent the set of inconsistent states at time slot $(M-d)$ that contributed to the inconsistency of $x_{n_i}(M-1)$. Every vertex is labeled with the event type ($E_1, E_2, or E_3$) of the inconsistency of the vertex.

The algorithm starts by setting an inconsistent state $x_{n_i}(M-1)$ as a root vertex. The root vertex is at depth $d=1$. Root vertex is labeled with the output of DecideType($x_{n_i}(M-1)$). If the output is E_2, then the algorithm terminates. If not, then $\widetilde{IN}_{n_i}(M-2)$ are added as children vertices to the root vertex. The same process is repeated for every vertex v at depth d. Vertex v is labeled with DecideType(v), and $\widetilde{IN}_{ID(v)}(M-(d+1))$ are added as children vertices to v. The algorithm is terminated if all vertices in V_d are labeled with E_2 or the depth of the tree is equal to M. A pseudocode description of the algorithm is given as algorithm **GrowTree**.

Algorithm GrowTree

Input: Inconsistent state $x_{n_i}(M-1)$
Output: Tree $\mathcal{T}$ with root vertex $x_{n_i}(M-1)$, depth $d \leq M$
Initialization:
Set $x_{n_i}(M-1)$ as root vertex of tree $\mathcal{T}$,
$d \leftarrow 1$;
 While $d < M$ 1
  For $v \in V_d$ 2
   Label v with DecideType(v); 3
   if DecideType($x_{n_i}(m)$) $\neq E_2$ 4
     Add children vertices $\widetilde{IN}_{ID(v)}(M-(d+1))$ to v; 5
  end 6
  If $|V_{d+1}| == 0$ 7
   Return $\mathcal{T}$ **exit;** 8
  else 9
   $d \leftarrow d+1$ 10
 end 11
Return $\mathcal{T}$

Figure 4: Algorithm of tree generation to detect misbehaving nodes that contributed to the inconsistent state of node n_i at time $M-1$

After having obtained the output tree $\mathcal{T}$, the verifier can identify misbehaving nodes in the following way. If a vertex that resides in depth d, v_d is labeled with E_2 or E_3, this implies the node that broadcasted v_d misbehaved during $M-(d+1)$.

In order for the proposed algorithm to be executed successfully at time slot M, each node needs to store all signed states from its neighbors in the previous M slots. Assuming each node is equipped with the same amount of memory, the least amount of memory required is $M \cdot \max_{n_i} |\mathcal{N}(n_i)|$ messages. We now proceed to prove properties of the proposed algorithm.

DEFINITION 4.1. *We define the set of misbehaving nodes $B(x_{n_i}(M-1))$ as* inconsistency-inducing nodes *of an inconsistent state $x_{n_i}(M-1)$ if the following two conditions are true:*

- *Node $n_j \in B(x_{n_i}(M-1))$ misbehaved at least once during time slots $\{0...M-2\}$.*
- *If $x_{n_j}(m+1) = f_j(IN_{n_j}(m))$ for all time slots $m \in \{0...M-2\}$, then $x_{n_i}(M-1)$ would be a consistent state.*

Definition 4.1 states that if every node in $B(x_{n_i}(M-1))$ behaved correctly from time 0 to time $M-2$, then $x_{n_i}(M-1)$ would be a consistent state.

DEFINITION 4.2. *A full information tree $\mathcal{T}_F$ with root vertex $x_{n_i}(M-1)$ is a tree with depth M where each vertex represents a state of a node and children vertices are input states used to compute the parent vertex.*

An example of a full information tree with $M = 4$ is shown in Figure 5.

THEOREM 4.3. ***GrowTree($x_{n_i}(M-1)$)** detects all inconsistency-inducing nodes of $x_{n_i}(M-1)$, $B(x_{n_i}(M-1))$.*

PROOF. Consider a full information tree $\mathcal{T}_F$ with root vertex $x_{n_i}(M-1)$. The tree $\mathcal{T}$ generated by the algorithm **GrowTree** is a subtree of $\mathcal{T}_F$ with the same root vertex. Note that every leaf vertex v_{leaf} of $\mathcal{T}$ is labeled with E_2, and all vertices on the path from a leaf vertex to the root vertex is labeled with either E_1 or E_3. If all misbehaving nodes that induced events E_2, E_3 behaved correctly, then the tree will only be left with vertices with label E_1, but all information flow through vertices labeled E_1 will be consistent since childeren vertices of v_{leaf} in $\mathcal{T}_F$ are all consistent states. Since **GrowTree** can detect all nodes that are responsible for vertices labeled E_2, E_3 in the tree, **GrowTree** detects all nodes in $B(x_{n_i}(M-1))$. □

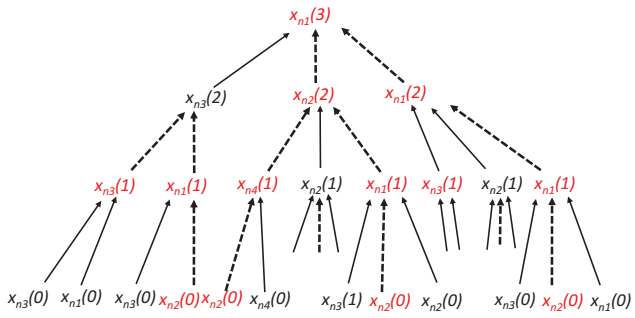

Figure 5: Example of inconsistent states from previous time slots affecting the state of node n_i at time slot 3. Dashed lines indicate inconsistent states transmitted to neighboring nodes.

The verifier runs the algorithm **GrowTree**($x_{n_i}(M-1)$) for each inconsistent state $x_{n_i}(M-1)$ to identify the set of misbehaving nodes that induced the inconsistency of $x_{n_i}(M-1)$.

5. SIMULATION STUDY

In this section, we conduct a numerical study using MATLAB. The goal of this study is to evaluate relationships between the inspection time slot M, average number of nodes being queried $|Q|$, and the probability of detecting misbehaving nodes in the network.

A wireless network consisting of 300 nodes is simulated. Nodes are distributed uniformly on a square with side length of 2000 units. Transmission range of each node is 200 units.

Each node's state dynamics is given as

$$X(m+1) = (I - \alpha \cdot L)X(m) \qquad (1)$$

where L is the Laplacian matrix [14] of the deployed network, and $\alpha = 0.01$ to guarantee convergence. Under the dynamics 1, each

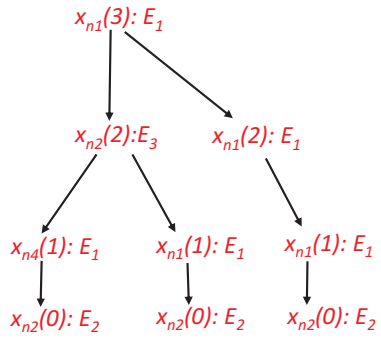

Figure 6: Illustration of the tree-generation algorithm executed on the example 5 (Figure 5). DecideType($x_{n_1}(3)$) returns event 1, inconsistent inputs used by node n_1 at time slot 2 is added as children vertices to the root vertex. Each vertex at depth 2 are labeled with outputs of DecideType($x_{n_2}(2)$), DecideType($x_{n_1}(2)$) and children vertices are added accordingly. Same process is run at depth 3. The verifier concludes that node n_2 broadcasted inconsistent initial state during time slot 0 and updated its state incorrectly during time slot 1.

node's state converges to the average value of the initial states of the nodes, provided the network is connected. Each node is given an initial state drawn uniformly from the interval $(0, 100)$. Five percent of nodes are chosen at random as misbehaving nodes. At each time slot, a misbehaving node adds a Gaussian random variable of zero mean and variance 4 to its updated state. Each node is independently chosen as a queried node with probability q. An example of the simulated network topology is shown in Figure 7.

The detection probability is estimated via Monte-Carlo methods. We performed 30 trials for each $M = 3, 5, 7$. Probability of detection is given as number of detected nodes divided by the number of misbehaving nodes in the network. We simulated two scenarios. In the first scenario, a state of node n_i is considered to be inconsistent at time slot m if $|\hat{x}_{n_i}(m) - x_{n_i}(m)| > 0$ for all m. In the second scenario, the verifier is willing to tolerate a small deviation from the ideal state, and a node's state is considered to be inconsistent if $|\hat{x}_{n_i}(m) - x_{n_i}(m)| > 0.3$. The acceptable range of 0.3 is chosen to be small compared to the average of the initial states of nodes. The simulation results of the first and second scenarios are shown in Figures 8 and 9, respectively.

For the networks simulated in our study, misbehaving nodes were detected with high probability (Figure 8). The probability of detection increases monotonically in q and M. As q increases from 0.01 to 1, the probability of detection increases in a logarithmic rate for all simulated values of M. This implies that there exists a threshold value of q such that once q exceeds the threshold value, detection probability does not increase as q increases. The fraction of nodes that needs to be queried to detect all misbehaving nodes is approximately 0.4 for $M = 3$, 0.2 for $M = 5$, and 0.1 for $M = 7$. In this case, any misbehaving node always induces inconsistency and any misbehaving node that is within M hops away from the queried node will be detected with probability one.

Figure 9 illustrates the detection probability of misbehaving nodes when small deviations from the ideal state are accepted. In this case, only the misbehaving nodes that induced deviations greater than the acceptable range are detected. It is observed that probability of detection increases in a similar logarithmic rate in q as in

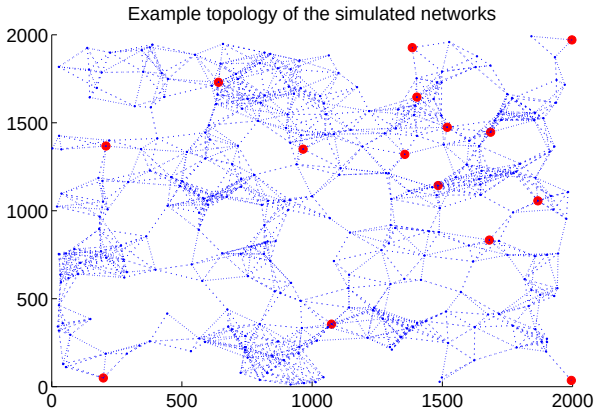

Figure 7: Example topology of the simulated network. 300 Nodes were uniformly deployed in a squre with length 2000 units. Each node has a communication range 200 units. 5 % of the nodes were chosen as misbehaving nodes. Filled red circles indicate locations of misbehaving nodes.

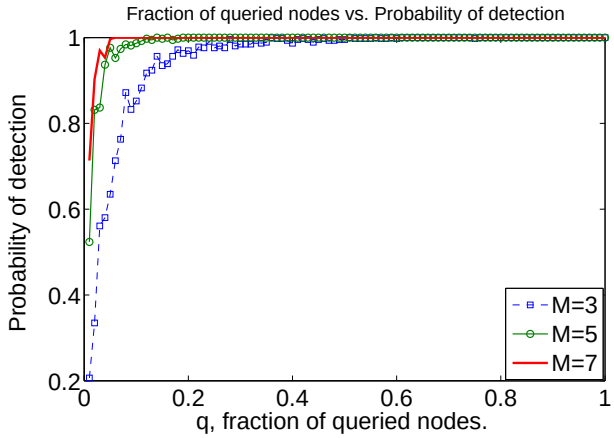

Figure 8: Simulation results comparing probability of detection with inspection period M. The fraction of nodes being queried ranges from $q = 0.01$ to $q = 1$. Three different values of $M = 3, 5, 7$ were simulated via Monte-Carlo methods with 30 trials each. Misbehaving nodes update its state incorrectly at each time slot by adding an zero mean Gaussian noise with variance of 4, causing each state to deviate from its ideal state with probability one. The fraction of nodes that needs to be queried to detect all misbehaving nodes is approximately 0.4 for $M = 3$, 0.2 for $M = 5$, and 0.1 for $M = 7$.

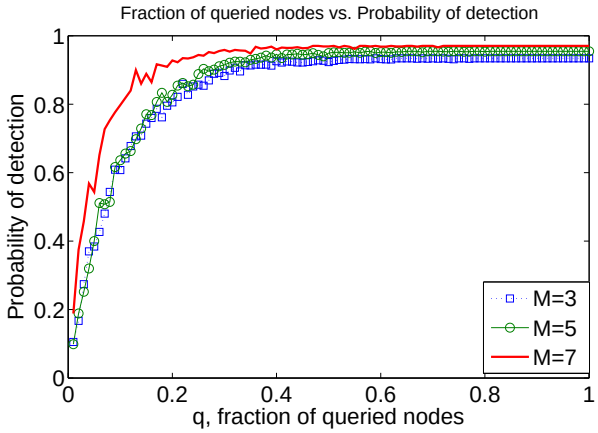

Figure 9: Simulation results comparing probability of detection with inspection period M. The fraction of nodes being queried ranges from $q = 0.01$ to $q = 1$. Three different values of $M = 3, 5, 7$ were simulated via Monte-Carlo methods with 30 trials each. Misbehaving nodes update its state incorrectly at each time slot by adding an zero mean Gaussian noise with variance of 4. A state of node n_i was considered inconsistent at any given time slot m if $|\hat{x}_{n_i}(m) - x_{n_i}(m)| > 0.3$. In this case, only the misbehaving nodes that induced deviations greater than the acceptable range are detected.

Figure 8. On the other hand, it is observed that increase in M does not increase the probability of detection as much as it did in the first scenario. Since each misbehaving node injects a zero mean Gaussian noise independently from each other at each state update iteration, positive and negative errors from different nodes may cancel out, resulting in a consistent state update within the acceptable range by the non-misbehaving nodes. Therefore, the overall induced inconsistency does not propagate throughout the network as fast as in the first scenario.

6. CONCLUSION AND FUTURE WORK

In this paper, the problem of detecting and identifying misbehaving nodes in multi-agent networks was studied. We considered a class of multi-agent networks where a third-party verifier has a knowledge of the deployed network topology and node dynamics. We proposed a detection mechanism where the verifier queries a partial set of nodes periodically, obtains the node state values from the query responses, and detects inconsistent states by comparing the state values from the queries with the ideal state values. We proposed an algorithm that systematically searches for misbehaving nodes that could have caused the detected inconsistent state. We proved that the proposed approach is equivalent to as a rooted tree-generation algorithm where the verifier can identify detected misbehaving nodes and their types of misbehaviors by examining the output tree. The effectiveness of the proposed detection mechanism was analyzed. Specifically, it was proved that starting from a node with an inconsistent state, the tree-generation algorithm detects all misbehaving nodes such that if these nodes have not misbehaved, the identified inconsistent state would instead be a consistent state.

In future work, we will investigate detection mechanisms against colluding misbehaving nodes where the misbehaving nodes are operated by one adversarial entity in a coordinated manner. We will also study how to deterministically choose the set of queried nodes Q given the network topology. Also, in this paper, we considered a broadcast communication model where if a node transmits a message, it will be received by all its neighbors. In future work, we will consider a more general case where communication links fail probabilistically, and the quality of the communication channel differs for each sender/receiver pair.

7. REFERENCES

[1] I. N. Athanasiadis and P. A. Mitkas, "An agent-based intelligent environmental monitoring system," *Management of Environmental Quality: An International Journal*, vol. 15, no. 3, pp. 238–249, 2004.

[2] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens, "An on-demand secure routing protocol resilient to Byzantine failures," *Proceedings of the 1st ACM Workshop on Wireless Security*, pp. 21–30, 2002.

[3] D. Bertsekas and R. Gallager, *Data Networks*. Prentice-Hall International, 1992, vol. 2.

[4] M. Blanke, *Diagnosis and Fault-Tolerant Control*. Springer, 2003.

[5] J. Chen, R. J. Patton, and H.-Y. Zhang, "Design of unknown input observers and robust fault detection filters," *International Journal of Control*, vol. 63, no. 1, pp. 85–105, 1996.

[6] W. Chen and M. Saif, "Fault detection and isolation based on novel unknown input observer design," *American Control Conference*, pp. 5129–5134, 2006.

[7] A. Clark and R. Poovendran, "A submodular optimization framework for leader selection in linear multi-agent systems," *IEEE Conference on Decision and Control and European Control Conference*, pp. 3614–3621, 2011.

[8] Y. Guan and M. Saif, "A novel approach to the design of unknown input observers," *IEEE Transactions on Automatic Control*, vol. 36, no. 5, pp. 632–635, 1991.

[9] A. Haeberlen, P. Kouznetsov, and P. Druschel, "The case for Byzantine fault detection," in *Proc. of the 2nd Workshop on Hot Topics in System Dependability*, 2006.

[10] ——, "Peerreview: Practical accountability for distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 175–188, 2007.

[11] M. Ji, A. Muhammad, and M. Egerstedt, "Leader-based multi-agent coordination: Controllability and optimal control," *American Control Conference*, pp. 1358–1363, 2006.

[12] H. J. LeBlanc and X. D. Koutsoukos, "Low complexity resilient consensus in networked multi-agent systems with adversaries," *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control*, pp. 5–14, 2012.

[13] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[14] M. Mesbahi and M. Egerstedt, *Graph Theoretic Methods in Multiagent Networks*. Princeton University Press, 2010.

[15] R. Olfati-Saber, "Flocking for multi-agent dynamic systems: Algorithms and theory," *IEEE Transactions on Automatic Control*, vol. 51, no. 3, pp. 401–420, 2006.

[16] F. Pasqualetti, A. Bicchi, and F. Bullo, "Distributed intrusion detection for secure consensus computations," *IEEE Conference on Decision and Control*, pp. 5594–5599, 2007.

[17] F. Pasqualetti, R. Carli, A. Bicchi, and F. Bullo, "Identifying cyber attacks via local model information," *IEEE Conference on Decision and Control*, pp. 5961–5966, 2010.

[18] J. Pavón, J. Gómez-Sanz, A. Fernández-Caballero, and J. J. Valencia-Jiménez, "Development of intelligent multisensor surveillance systems with agents," *Robotics and Autonomous Systems*, vol. 55, no. 12, pp. 892–903, 2007.

[19] W. Ren, R. W. Beard, and T. W. McLain, "Coordination variables and consensus building in multiple vehicle systems," in *Cooperative Control*. Springer, 2005, pp. 171–188.

[20] S. Sundaram and C. N. Hadjicostis, "Distributed function calculation via linear iterative strategies in the presence of malicious agents," *IEEE Transactions on Automatic Control*, vol. 56, no. 7, pp. 1495–1508, 2011.

[21] S. Sundaram, M. Pajic, C. N. Hadjicostis, R. Mangharam, and G. J. Pappas, "The wireless control network: monitoring for malicious behavior," *IEEE Conference on Decision and Control*, pp. 5979–5984, 2010.

[22] H. Zhang and S. Sundaram, "Robustness of information diffusion algorithms to locally bounded adversaries," *American Control Conference*, pp. 5855–5861, 2012.