MIC Final Report: Diffusion LMS Algorithm Domain

Jiani Li, Manyao Peng, Yongtai Liu,

December 13, 2017

1 Introduction

Malicious attackers always seek to find various systems' vulnerabilities and penetrate into the system to cause severe physical damage by cyber-attacks. Examples include Stuxnet worm attacking nuclear centrifuges in Iran [1], Maroonchy Shire water-services incident [2], and cyber-attack on a sewage treatment facility in Queensland, Australia [3]. Thus, cyber-security is a significant problem to be addressed in CPS.

Diffusion Least-Mean Squares (DLMS) is a powerful algorithm for distributed state estimation [4]. It enables networked agents to interact with neighbors on a local level in response to streaming data and diffuse information across the network to continually solve the estimation tasks. Comparing to the centralized strategy, diffusion strategy has the advantages of robustness to drifts in the statistical properties of the data, scalability, relying solely on local data, responding in real-time, and so forth. Emerging applications include spectrum sensing in cognitive networks [5], target localization [6], distributed clustering [7], biologically inspired designs [8].

DLMS has been deemed robust to node and link failures [9]. However, such robustness is only for high noise level of nodes or noisy links and few literature has discussed the resilience of DLMS from a cyber-security perspective. In fact, in cyber-physical systems (CPS), the penetration of a malicious attacker will destroy the resilience of the system. To enhance the resilience of DLMS, previous works such as [7] and [10] modified it to adaptively change weights assigned to neighbors according to the Euclidean distance with neighbors. Such modification enhances the resilience of DLMS, yet it introduces another vulnerability into the system, which we will discuss in the following sections.

2 Background and description

2.1 Diffusion Least-Mean-Square Algorithm

We use normal and boldface font to denote deterministic and random variables respectively. The superscript $(\cdot)^*$ denotes complex-conjugate transposition for matrices, $\mathbb{E}\{\cdot\}$ denotes expectation, $\|\cdot\|$ denotes the euclidean norm of a vector.

Consider a connected network of N agents. At each iteration i, each agent k has access to a scalar measurement $\boldsymbol{d}_k(i)$ and a regression vector $\boldsymbol{u}_{k,i}$ of size M with zero-mean and uniform covariance matrix $R_{u,k} \triangleq \mathbb{E}\{\boldsymbol{u}_{k,i}^* \boldsymbol{u}_{k,i}\} > 0$, which are related via a linear model of the following form:

$$\boldsymbol{d}_k(i) = \boldsymbol{u}_{k,i} w_k^0 + \boldsymbol{v}_k(i)$$

where $\boldsymbol{v}_k(i)$ represents a zero-mean i.i.d. additive noise with variance $\sigma_{v,k}^2$ and w_k^0 denotes the unknown $M \times 1$ state vector of agent k. The objective of each agent is to estimate w_k^0 from (streaming) data $\{\boldsymbol{d}_k(i), \boldsymbol{u}_{k,i}\}$ $(k = 1, 2, ..., N, i \geq 0)$.

The state w_k^0 can be computed as the the unique minimizer of the following cost function:

$$J_k(w) \triangleq \mathbb{E}\{\|\boldsymbol{d}_k(i) - \boldsymbol{u}_{k,i} w\|^2\}.$$

An elegant adaptive solution for determining w_k^0 is the least-mean-squares (LMS) filter [4], where each agent k computes successive estimators of w_k^0 without cooperation (noncooperative LMS) as follows:

$$\boldsymbol{w}_{k,i} = \boldsymbol{w}_{k,i-1} + \mu_k \boldsymbol{u}_{k,i}^* [\boldsymbol{d}_k(i) - \boldsymbol{u}_{k,i} \boldsymbol{w}_{k,i-1}]$$

Compared to noncooperative LMS, diffusion strategies introduce an aggregation step that incorporates into the adaptation mechanism information collected from other agents in the local neighborhood. One powerful diffusion scheme is adapt-then-combine (ATC) [4] which optimizes the solution in a distributed and adaptive way using the following update:

$$\boldsymbol{\psi}_{k,i} = \boldsymbol{w}_{k,i-1} + \mu_k \boldsymbol{u}_{k,i}^* [\boldsymbol{d}_k(i) - \boldsymbol{u}_{k,i} \boldsymbol{w}_{k,i-1}] \qquad \text{(adaptation)}$$

$$\boldsymbol{w}_{k,i} = \sum_{l \in N_k} a_{lk}(i) \boldsymbol{\psi}_{l,i} \qquad \text{(combination)}$$

where N_k denotes the neighborhood set of agent k including k itself, $\mu_k > 0$ is the step size (can be identical or distinct across agents), $a_{lk}(i)$ represents the weight agent k assigns to its neighbor

l that is used to scale the data it receives from l, andt the weights should satisfy the following constraints:

$$a_{lk}(i) \geq 0, \qquad \sum_{l \in N_k} a_{lk}(i) = 1, \qquad a_{lk}(i) = 0 \text{ if } l \notin N_k.$$

In the case when the agents estimate a common state w^0 (i.e., w_k^0 is the same for every k), several combination rules can be adopted such as Laplacian, Metropolis, averaging, and maximum-degree [11]. In the case of multiple tasks, the agents are pursuing distinct but correlated objectives w_k^0. The combination rules mentioned above are not applicable because they simply combine the estimation of all neighbors without distinguishing if the neighbors are pursuing the same objective. An agent estimating a different state will prevent its neighbors from estimating the state of interest.

Diffusion LMS (DLMS) has been extended for multitask networks in [7] using the following adaptive weights:

$$a_{lk}(i) = \begin{cases} \frac{\gamma_{lk}^{-2}(i)}{\sum_{m \in N_k} \gamma_{mk}^{-2}(i)}, & l \in N_k \\ 0, & \text{otherwise} \end{cases}$$

where $\gamma_{lk}^2(i) = (1 - \nu_k)\gamma_{lk}^2(i-1) + \nu_k \|\boldsymbol{\psi}_{l,i} - \boldsymbol{w}_{k,i-1}\|^2$ and ν_k is a positive step size known as the forgetting factor. This update based on adaptive weights enables the agents to continuously learn which neighbors should cooperate with and which should not. During the estimation task, agents pursuing different objectives will assign to each other continuously smaller weights according to (2.1). Once the weights become negligible, the communication link between the agents does not contribute to the estimation task. As a result, only agents estimating the same state will end up being connected.

2.2 Other combination rules

Besides the adaptive combination rule just introduced in the last subsection, there are multiple combination rules. In this project, we consider three other popular combination rules, namely, uniform, maximum degree and metropolis combination rule, summarized in table 1.

2.3 Attacker introduced

As we discussed in section 2.1, adaptive combination rule is resilient to the attacker that objective at a different state than the agent itself. Whereas all the other combination rules are not resilient

Combination Rule	$a_{lk}(i)$
Uniform	$\frac{1}{n_k}$
Maximum Degree	$\frac{1}{N}$
Metropolis	$\frac{1}{\max(n_k,n_l)}$
Adaptive	$\frac{\gamma_{lk}^{-2}(i)}{\sum_{m\in N_k}\gamma_{mk}^{-2}(i)}$

Table 1: Combination rules

to such attacks because they do not change weights to neighbors and will always be affected by a possible attacker in their neighbor.

We thus design a type of attacker specific for network applying adaptive combination rule. At each iteration, the attacker sends the following message to all its normal neighbors:

$$\boldsymbol{\psi}_{a,k,i} = \boldsymbol{w}_{k,i-1} + r_k^a(w_k^a - \boldsymbol{w}_{k,i-1})$$

where w_k^a is the state of attacker's interest that it wants normal agents to converge to, r_k^a is step size.

But it should be noted that even though the designed attack can work on adaptive combination rule. It does not work for other non-adaptive combination rules. Because the requirement for such attack to succeed is for the attacker to be able to get significant weight from normal agents. Yet the non-adaptive combination rules does not satisfy such requirements.

2.4 Evaluation

The estimation performance can be measured by the steady-state mean-square-deviation of the network:

$$\text{MSD} \triangleq \lim_{i\to\infty}\frac{1}{N}\sum_{k=1}^{N}\mathbb{E}\|\tilde{\boldsymbol{w}}_{k,i}\|^2 = \lim_{i\to\infty}\frac{1}{N}\sum_{k=1}^{N}\mathbb{E}\|w_k^0 - \boldsymbol{w}_{k,i}\|^2$$

The better the estimation performance, the smaller MSD level is. Therefore, the task for the normal agents is to minimize the MSD level of the network. Whereas the attacker objectives at maximizing the MSD level.

3 Problem

In this project, we want to take use of the great visualized property of WebGME to build network. And the problem we try to solve is to design a plugin that given the network built in WebGME, the plugin generates the adjacency matrix based on the network topology. The plugin should also generates the agent information and attacker information (if any). We write diffusion LMS algorithm code in Matlab. The Matlab files need the input of adjacency matrix and attacker info from WebGME. And Given those info, Matlab runs the algorithm and generates the evaluation plot – MSD plot. Therefore, we also needs a plugin to trigger Matlab in local server and after Matlab running the algorithm and generating plots, the plugin should get the generated plots and show these results in WebGME.

4 Model

4.1 Meta model

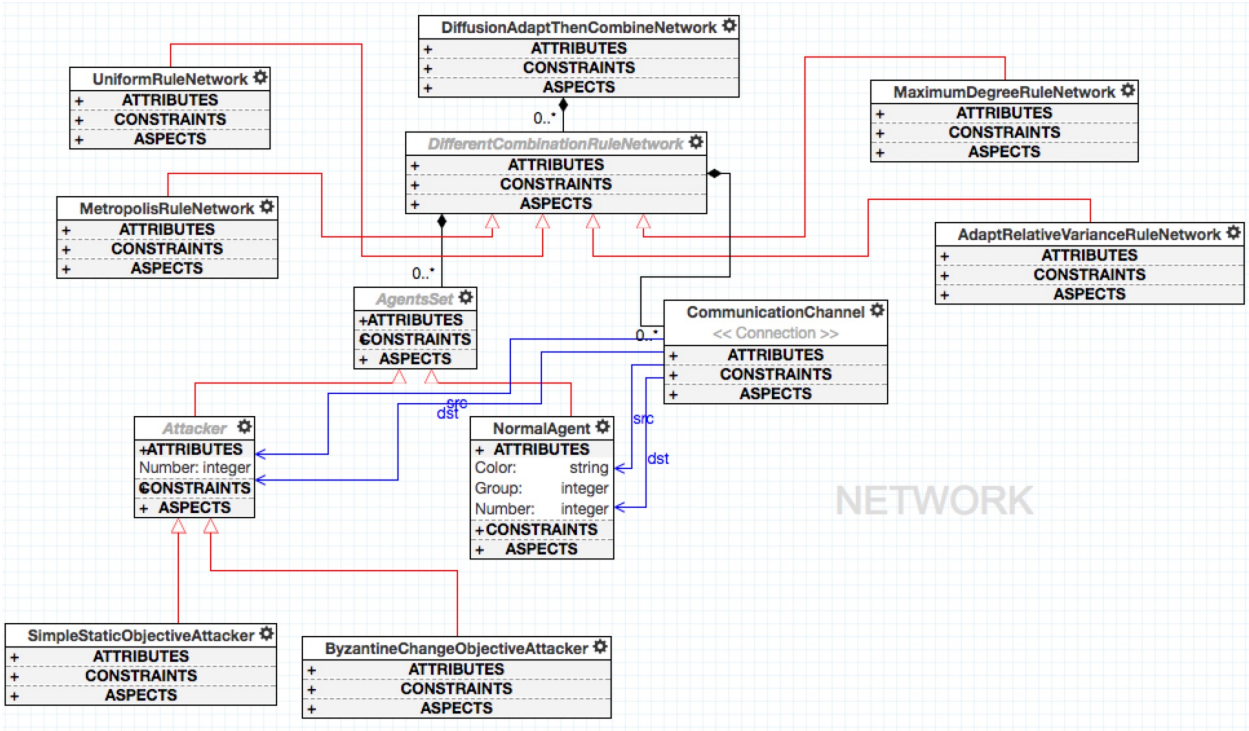

Figure 1: Meta model for diffusion LMS network

Figure 1 shows the meta model for our diffusion LMS network. Here, the diffusion LMS network under consideration implements the above mentioned four combination rules – uniform, maximum degree, metropolis, adaptive combination rules. In each network, we have agents that form AgentSet. Agents can be normal agents and attacker. And there are two types of attackers: simple static objective attacker, which behaves nothing different than a normal agent, except for that it objectives at a different state than normal agents; byzantine change objective attacker is designed for adaptive combination rules, which always sends the communication message as section 2.3 indicates. Normal agents and attackers are connected by the communication channel.

4.2 Composition

Figure 2 shows the composition our diffusion LMS network.

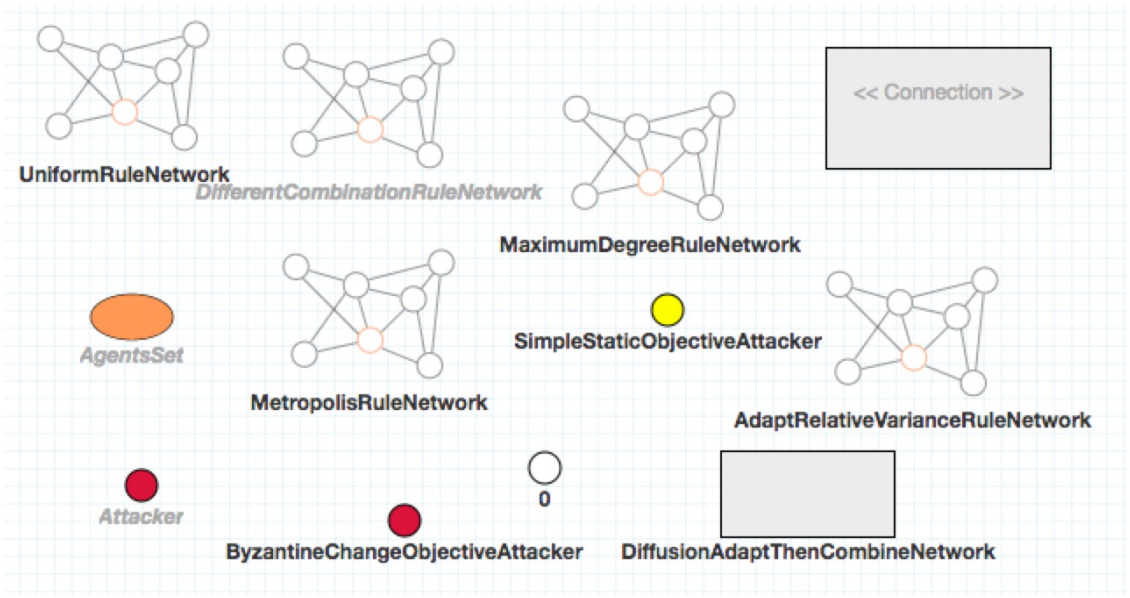

Figure 2: Composition for diffusion LMS network

4.3 Instances

Instances include four network applying the four above mentioned combination rules. We implement four of those because each will call the plugin to run the corresponding Matlab codes. Figure 3 shows an instance of adaptive combination rule network. In this instance, we build a network of 12 normal agents and a byzantine attacker. There are four instances in the project. When you

build new networks, every time when you add a node into the network, remember to set the No. of the node in its attributes. The No. of the node is the key how the plugin generates the adjacency matrix.

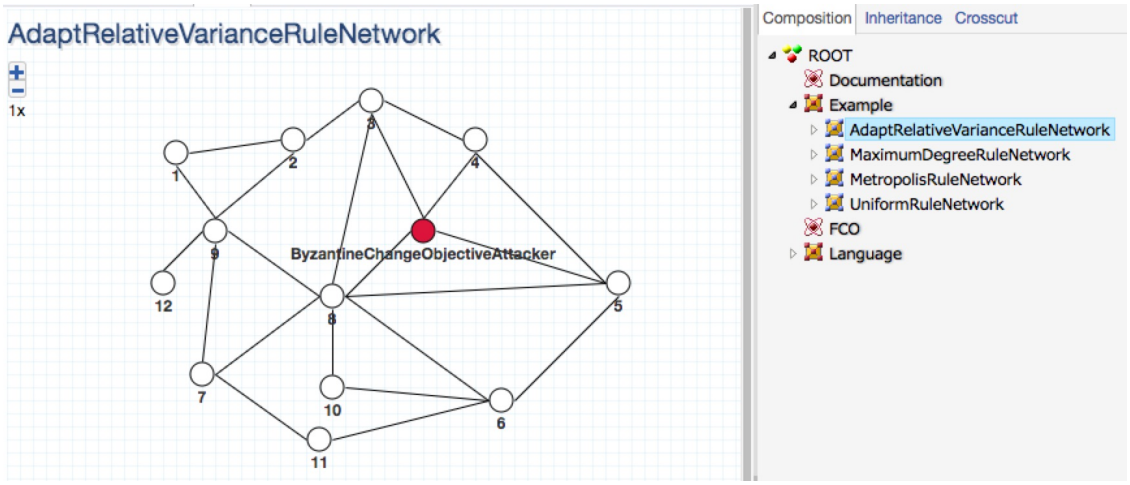

Figure 3: An instance of an adaptive combination rule network

5 Implementation

We have implemented two plugins. One of them is to generate the network information (adjacency matrix, attacker info) in Matlab format. The other is to interact with Matlab. Specifically, it calls Matlab in local server and sends the generated file to Matlab. After Matlab runs the algorithm, it feeds back the MSD plots generated by Matlab to WebGME.

5.1 Plugin A: DACodeGenerator

DACodeGenerator is the plugin we used to generate network information. This plugin use a depth first search to traverse all the members in the model, get their information, and then use javascript string concatenation to generate Matlab file, the generated file will be save to local directory. This plugin has two parts: first part is used to generate the adjacency matrix, which records the connection between agents(nodes). The second part is used to get attackers' information, which consists of a attacker list and a attacker-type list. The attacker list contains the index of attackers, and the attacker-type list, which contains the type of each attacker (Byzantine or Simple). Figure

4 and 5 are the corresponding javascript codes.

```
function buildMatlabScript(members){
  var mFileString = '';

  mFileString += 'function Adjacency = getAdjacency()' + '\n';
  mFileString += '  size = '+ members.nbrOfAgents +'\n';
  mFileString += '  Adjacency = zeros[size]' + '\n';

  for (var i in members.children){
    if (members.children[i].metaType == 'Link'){
      let index1 = members.children[i].src;
      let index2 = members.children[i].dst;

      mFileString += '  Adjacency('+index1+','+index2+ ') = 1' +'\n';
      mFileString += '  Adjacency('+index2+','+index1+ ') = 1' +'\n';
    }
  }
  return mFileString;
}
```

Figure 4: Generate adjacency matrix

```
else if (chMetaType == 'ByzantineChangeObjectiveAttacker' || chMetaType == 'SimpleStaticObjectiveAttacker'){
          let index = members.children[i].Number;
          mAttackerString1 += index +' ';
          mAttackerString2 += "'" + chMetaType+"'" + ' ';

  }
```

Figure 5: Generate Attacker

5.2 Plugin B: SimulateMatlab

SimulateMatlab is the plugin we used to interact with Matlab. This plugin first calls Matlab from command line, run the generated script. The Matlab program will generate a MSD plot, SimulateMatlab plugin will pass the result back to WebGME, save the figure as an attribute of the network, and display it in the plugin results frame. Figure 6 and 7 shows codes for calling Matlab and display result parts respectively.

5.3 Matlab implementation

We implemented four networks with four different combination rules for diffusion LMS algorithm, each is done in a separate Matlab file. In the main function of the four network, function "getData()"

```
function simulateModel(dir,modelName){
        var command;
        command = 'matlab -nodesktop -nosplash -r '+modelName+',quit';

        return Q.ninvoke(cp, 'exec', command, {cwd:dir})
                .then(function (res) {
                        logger.info(res);

                        return{
                                dir: dir,
                                resultFilename: 'Network.jpg'

                        };
                });
}
```

Figure 6: Calling Matlab

```
.then(function (csvFileHash) {
      self.result.addArtifact(csvFileHash);
       self.core.setAttribute(activeNode, 'simResults', csvFileHash);

       //show image in plugin results.
       var imgURL = self.blobClient.getRelativeViewURL(csvFileHash);
       self.createMessage(activeNode, '<img src="'+imgURL+'"'+'width="50%" height="50%"/>' );

       return self.save('Attached simulation results at '+ self.core.getPath(activeNode));
})
```

Figure 7: Display result

is called to obtain the network information from WebGME. This "getData()" function is generated by WebGME and should look like what is shown in Figure 8.

6 Example

We can design a lot of diffusion strategies, and our application can be used to check whether the defined strategies can defend the attack from different types of attackers or not. We use WebGME to build network with the diffusion strategies we want to test, and then add attackers to the network, then we simulate the attack and get the MSD (Mean Standard Deviation) plot as result. MSD is a measurement of the network's robustness, lower MSD means the network is more robust. Here, I will use two different strategies as examples to illustrate our applications.

```
function [Adjacency, attacker, attackType]  = getData()
  size = 10;
  Adjacency = eye(size);
  Adjacency(9,8) = 1;
  Adjacency(8,9) = 1;
  Adjacency(7,9) = 1;
  Adjacency(9,7) = 1;
  Adjacency(10,9) = 1;
  Adjacency(9,10) = 1;
  Adjacency(8,7) = 1;
  Adjacency(7,8) = 1;
  Adjacency(9,1) = 1;
  Adjacency(1,9) = 1;
  Adjacency(2,9) = 1;
  Adjacency(9,2) = 1;
  Adjacency(1,2) = 1;
  Adjacency(2,1) = 1;
  Adjacency(10,1) = 1;
  Adjacency(1,10) = 1;
  Adjacency(2,3) = 1;
  Adjacency(3,2) = 1;
  Adjacency(3,4) = 1;
  Adjacency(4,3) = 1;
  Adjacency(3,8) = 1;
  Adjacency(8,3) = 1;
  Adjacency(4,5) = 1;
  Adjacency(5,4) = 1;
  Adjacency(5,8) = 1;
  Adjacency(8,5) = 1;
  Adjacency(5,6) = 1;
  Adjacency(6,5) = 1;
  Adjacency(6,8) = 1;
  Adjacency(8,6) = 1;
  attacker = [10];
  attackType = ['ByzantineChangeObjectiveAttacker'];
  %attackType = ['SimpleStaticObjectiveAttacker'];
```

Figure 8: getData() function generated by WebGME

6.1 Example 1: Adaptive Relative Variance Rule Network

To test the of Adaptive Network, we first build the network with only normal agents, run program to test its MSD level. Then we add some new attackers in different type, and test the MSD level again. Compare the result and draw conclusion. Figure 9 are networks with only normal agents, simple attacker and byzantine attacker and figure 10 shows their corresponding MSD level. We can see that the normal adaptive network and the network with simple attackers have similar MSD value (around -120), which means that simple attacker has no influence on Adaptive network. On the other hand, the network with Byzantine attacker has a higher MSD (-6) than previous two, which means that the network build with Adaptive Relative Variance strategy can not defend

'Byzantine Change Objective' attack.

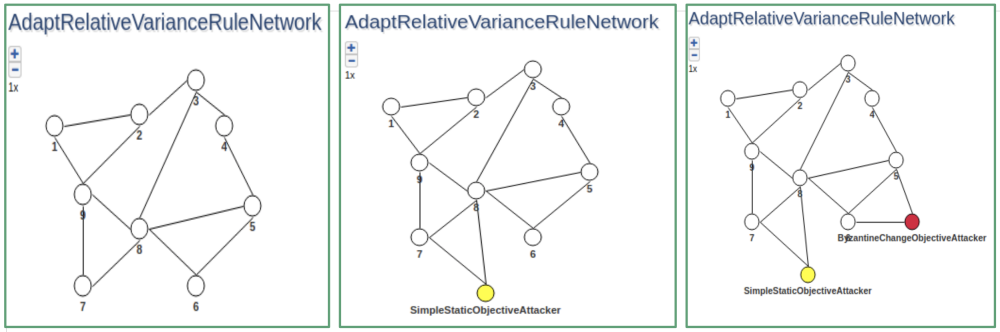

Figure 9: Adaptive Networks with: only normal agents, agents + Simple attacker, agents + Simple attacker + Byzantine attacker

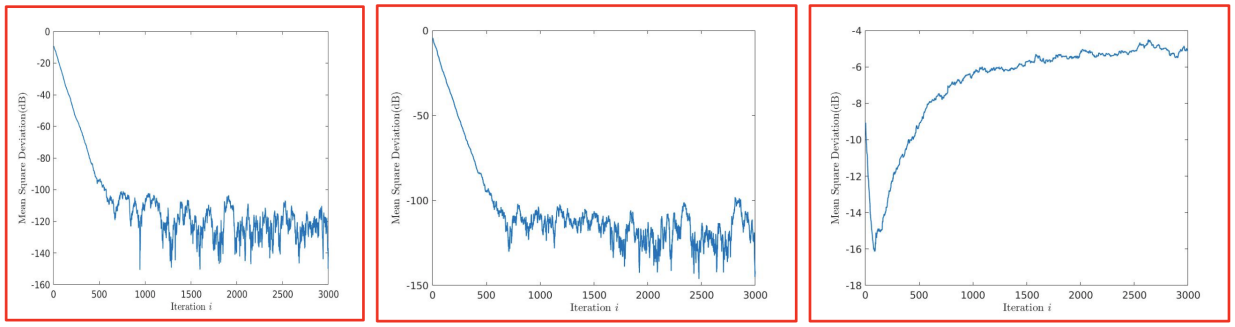

Figure 10: Corresponding result

6.2 Example 2 Maximum Degree Rule Network

Now we test the network with a different strategy: Maximum Degree Rule. Same as previous example, we build the network, add attackers to it, test models and compare their result. From figure 12 , we can see that the the maximum network with byzantine attackers have similar MSD value (around -100) to maximum network with only normal agents, which means that Maximum Degree is robust to Byzantine attack. On the other hand, the network with Simple attacker has a higher MSD (-20) than previous two, which means that the Maximum Degree strategy can not defend 'Simple Static Objective' attack.

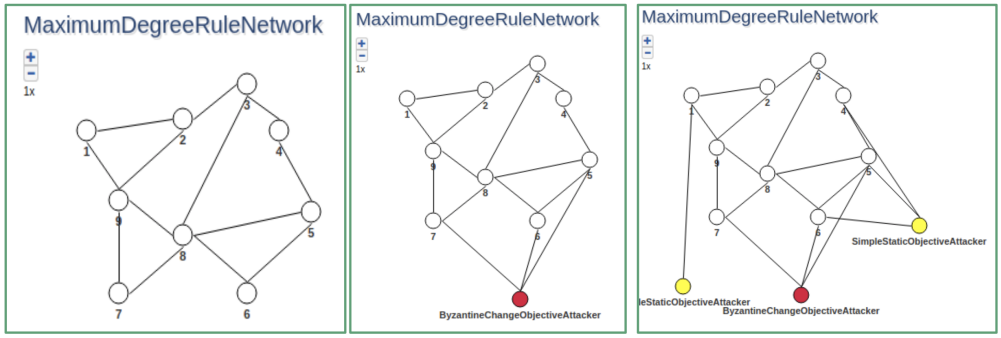

Figure 11: Maximum Networks with: only normal agents, agents + Byzantine attacker, agents + Simple attacker + Byzantine attacker

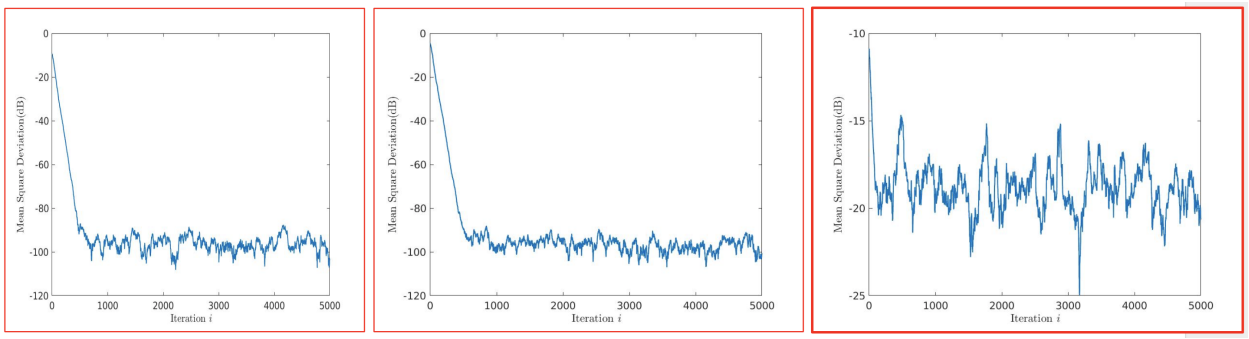

Figure 12: Corresponding result

References

[1] D. Kushner. The real story of stuxnet. *IEEE Spectrum*, 50(3):48–53, March 2013.

[2] M. Abrams and J. Weiss. Malicious. Malicious control system cyber security attack case study - maroochy water services, australia. *Technical Report Mitre.org.*, pages 1–16, 2008.

[3] Jill Slay and Michael Miller. Lessons learned from the maroochy water breach. In *Critical Infrastructure Protection, Post-Proceedings of the First Annual IFIP Working Group 11.10 International Conference on Critical Infrastructure Protection, Dartmouth College, Hanover, New Hampshire, USA, March 19-21, 2007*, pages 73–82, 2007.

[4] Ali H. Sayed, Sheng-Yuan Tu, Jianshu Chen, Xiaochuan Zhao, and Zaid J. Towfic. Diffusion strategies for adaptation and learning over networks: An examination of distributed strategies and network behavior. *IEEE Signal Process. Mag.*, 30(3):155–171, 2013.

[5] J. Plata-Chaves, N. Bogdanovi, and K. Berberidis. Distributed diffusion-based lms for node-specific adaptive parameter estimation. *IEEE Transactions on Signal Processing*, 63(13):3448–3460, July 2015.

[6] Amin Lotfzad Pak, Azam Khalili, Md. Kafiul Islam, and Amir Rastegarnia. A distributed target localization algorithm for mobile adaptive networks. *ECTI Transactions on Electrical Engineering, Electronics, and Communications*, 14:47–56, 08 2016.

[7] X. Zhao and A. H. Sayed. Clustering via diffusion adaptation over networks. In *2012 3rd International Workshop on Cognitive Information Processing (CIP)*, pages 1–6, May 2012.

[8] Sheng-Yuan Tu and Ali H.Sayed. Mobile adaptive networks. *IEEE J. Sel. Topics Signal Process*, 5(4):649–664, 2011.

[9] S. Y. Tu and A. H. Sayed. Diffusion strategies outperform consensus strategies for distributed estimation over adaptive networks. *IEEE Transactions on Signal Processing*, 60(12):6217–6234, Dec 2012.

[10] S. Y. Tu and A. H. Sayed. Optimal combination rules for adaptation and learning over networks. In *2011 4th IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)*, pages 317–320, Dec 2011.

[11] Ali H. Sayed. *Diffusion Adaptation over Networks*, volume 3. Academic Press, Elsevier, 2014.