An Effective Distributed GHSOM Algorithm for Unsupervised Clustering on Big Data

Chui-Hui Chiu  
Dept. Management Information Systems  
National Chengchi University  
Taipei, Taiwan  
104356019@nccu.edu.tw

Jin-Jie Chen  
Dept. Management Information Systems  
National Chengchi University  
Taipei, Taiwan  
103356039@nccu.edu.tw

Fang Yu  
Dept. Management Information Systems  
National Chengchi University  
Taipei, Taiwan  
yuf@nccu.edu.tw

Abstract—Clustering techniques that group samples based on their attribute similarity have been widely used in many fields such as pattern recognition, feature extraction and malicious behavior characterization. Due to its importance, various clustering techniques have been developed with distributed frameworks such as K-means with Hadoop in Apache Mahout for scalable computation. While K-means requires the number of clusters and self organizing maps (SOM) requires the map size to be given, the technique of GHSOM (growing hierarchical self organizing maps) that clusters samples dynamically to satisfy the requirement on tolerance of variation between samples, poses an attractive unsupervised learning solution for data that have limited information to decide the number of clusters in advance. However it is not scalable with sequential computation, which limits its applications on big data. In this paper, we present a novel distributed algorithm on GHSOM. We take advantage of parallel computation with scala actor models for GHSOM construction, distributing vertical and horizontal expansion tasks to actors and showing significant performance improvement. To evaluate the presented approach, we collect and analyze execution behaviors of thousands of malware in real life and derive detection rules with the presented unsupervised clustering on millions samples, showing its performance improvement, rule effectiveness and potential usage in practice.

Keywords—unsupervised clustering; GHSOM; actor model; malware detection; parallel computation;

I. INTRODUCTION

Unsupervised clustering techniques have been widely used in many fields, such as characterizing malicious behaviors. We motivate this work from the need of unsupervised data modeling on logs for security of cloud systems. In recent years, people can easily gain computing resources and access data storage space through cloud service providers, like Amazon Web Service (AWS). While people have benefited from the cloud, the massive concentrations of resources and data set in the cloud poses an attractive target to attackers. Successful cloud attacks may cause tremendous losses of information assets. As a result, malware detection systems were invented and designed to prevent devices from being attacked. A key issue is to characterize malware behavior as specific patterns that can be used to detect ongoing malicious attacks. Previous research proposes clustering techniques to differentiate between benign behaviors, which demonstrate very similar patterns, and malicious behaviors. For instance, Leung et al. [1] have proposed an unsupervised anomaly detection system using clusters, and Burguera et al. [2] have proposed a behavior-based malware detection system for android applications. They collect system calls of smart phones with android operating system and using K-means [3] clustering algorithm to find some behaviors of different mobile applications.

A typical clustering algorithm is the K-means algorithm that iteratively separates samples into k clusters so that within a cluster samples having its sum of square on Euclidean distance minimized. There have been various research on variations for K-means performance enhancement. Esteves et al. [4] proposed distributed computation to enhance the K-means training performance using Apache Mahout [5], an open source cloud computing approach to K-means that runs on the Hadoop system. Kanungo et al. [6] proposed a filtering algorithm using kd-tree for model training. Nevertheless, the main disadvantage of K-means algorithm is that the k value has to be determined before training.

The self organizing map (SOM) [7] [8] is an unsupervised clustering algorithm that creates a topology preserving projection of the prototype vectors (samples). The algorithm represents the dataset from n-dimensional input space onto a low-dimensional (usually 2D) grid, which shows an appropriate visualization surface to understand the clustering result. SOM has been widely used in many fields. Tamayo et al. [9] used SOM to interpret patterns of gene expressions. Alhoniemi et al. [10] applied it to monitor complex industrial processes. Hydrological processes [11] can also be applied with the SOM analysis for estimation and prediction.

However, there are two main disadvantages of SOM. One is the map size has to be defined prior to training. There’s no effective and efficient way to determine the map size for the best performance. In many cases, one may need to train SOM models many times for finding suitable one. The other disadvantage is that we do not have the hierarchical relations among clusters due to the same representation space of the map.

The growing hierarchical self organizing map (GHSOM) is designed for dealing with the limitations of SOM. GHSOM adopts an incrementally growing neural network architecture with multiple layers where each layer consists of independent SOMs instead of a fixed SOM structure. The model starts from
an initial SOM, which consists of many units at the first layer. And then iteratively expand the maps or the unit until the error rates are tolerated. There is then no need to assign the map size in advance, but to define the tolerance of variations. GHSOM has been used for understanding the evidence of attacks or anomalous behavior in network environment [12] [13] and Virtual machine [14]. In this research, we use it to find patterns that are specific to malware for detection.

When dataset is small, such advanced learning algorithms with high-complexity computation are affordable. However, in most cases a large dataset is collected from cloud system logs, and the learning algorithms that can not be scaled out are not applicable in practice. In this research, we propose distributed GHSOM algorithm to facilitate distributed computation for a KVM-based cloud malware detection system. Applying the proposed unsupervised clustering, we are able to construct the GHSOM models among a large set of VM executions and derive effective detection rules with clusters that characterize malicious behaviors.

Our system consists of two parts. The first part is data collection [15]. In this part, we use open source hypervisor Linux Kernel Virtual Machine (KVM) as our virtualization hypervisor. Every virtual machine running on KVM installed Cuckoo Sandbox [16] which is to simulate an environment for different malwares to execute. When a VM is running within a KVM environment and is simultaneously running as a process of qemu-KVM. The VM runtime status can be monitored by using strace to record the kinds of system calls of VM executions. Then we count each call in a fixed size piece of executions as a sample. Each sample represents a small piece of VM execution, and can be used to find out the specific behavior during execution of a malicious program. This is done by the second part: data analysis with unsupervised clustering. We use these sequential system calls to reanalyze and reclassify by doing a similarity analysis using distributed GHSOM. Distributed GHSOM can group samples that have tolerable differences into the same cluster in a distributed fashion where scale-out actors can be used to take the tasks of SOM expansions, and hence enhance the scalability of the training process. In practice, we are able to apply the presented clustering algorithm to construct GHSOM models on millions samples from VM execution logs and derive malware detection rules to identify the moments when malwares are behaving abnormal behaviors (not observed in executions of benign VMs) in the cloud systems. We also show that compared to detection rules that are derived from K-means clusters, using GHSOM clusters, we can achieve much higher accuracy with lower false positive/negative rates.

We present our actor-based GHSOM algorithm in the next section and evaluate its performance. We then present how to conduct malware detection based on the GHSOM clustering and evaluate its accuracy against K-means clustering.

II. ACTOR-BASED GHSOM ALGORITHMS

GHSOM has already used for understanding the evidence of attacks or anomalous behavior in network environment [12] [13] and Virtual machine [14]. In this research, we use it to find patterns that are specific to malware for detection.

FAHAD et al. [17] have conducted a survey of clustering algorithms for big data. They did some experiments to test the accuracy and runtime of each clustering algorithm. For accuracy, model-based clustering algorithm, like EM algorithm, provides the best clustering performance, while model-based clustering algorithm is less efficient when dealing with big data.

In this paper, we choose GHSOM to be our clustering algorithm, a model-based clustering algorithm that construct dynamically hierarchical structures of SOM. We proposed an architecture to train GHSOM model in a distributed manner and use bloom filter [18] techniques to represent each cluster in the SOM layers and hence reduce the storage cost.

Figure 1 shows the framework of distributed GHSOM. The GHSOM model is a hierarchical architecture with different layers. Layers are composed of independent training of neuron maps. To train the GHSOM model in a distributed manner, our framework consists of a LayerHandler and multiple Expanders (that can be scaled out). Expanders receive a neuron map as the message to conduct the SOM clustering work and output one or many neuron maps. When an expander finishes its work, it returns the GrownResult to the LayerHandler. The LayerHandler then appends the result to the tree. Since the training can be conducted independently in the same level, we apply actor models to scale out expanders.
A. The Actor Model of GHSOM

An actor model [19] is designed for implementation of message-passing concurrent systems. An actor-based system consists of entities called actors. Each actor is responsible for the corresponding task with asynchronous communications. There are two ways to send a message to actors, SendMsgAndForget and SendMsgAndWait. Both algorithms send a message asynchronously to actors and return immediately. The only difference is that SendMsgAndForget does not expect actors to respond but SendMsgAndWait does return a future object representing a possible reply. In other words, both algorithms are non-blocking and asynchronous but one does return something the other does not. The behavior of an actor is defined by how the actor handles messages when they are received. Since the state of an actor is isolated from the rest of the system, an actor has nothing to do with synchronization issues, such as thread locking, when modifying its state.

B. The Distributed GHSOM Algorithm

Algorithm 1 GHSOM()

1: meanVector ← GetMeanVector(dataSet)
2: dataSetMQE ← GetDataSetMQE(dataSet)
3: trainedBloomFilter ← Training(dataSet)
4: initNeuron ← CreateNeuron(dataSetMQE, meanVec, trainedBloomFilter)
5: initNeuronMap ← CreateNeuronMap(initNeuron, dataSetMQE, trainedBloomFilter)
6: initGHSOMLayer ← CreateGHSOMLayer(initNeuronMap)
7: GHSOMTreeBuilder ← createGHSOMTreeBuilder(initGHSOMLayer)
8: expander ← CreateNeuronMapActor(GHSOMTreeBuilder)
9: neuronMap1 ← CreateNeuronMap(initNeuron)
10: sendMsgAndForget(toMsg(Expand, neuronMap1), expander)

The basic structure of GHSOM training is a tree composed of layers, each layer contains many neuron maps, which have neurons in represent of distribution of input patterns. Algorithm 1 states that GHSOM is composed of some parameters and two actors, GHSOMTreeBuilder (Algorithm 2) and expander (Algorithm 4). For GHSOM algorithm (Algorithm 1), firstly we need to calculate the mean vector of input dataset, then we can use it to calculate the mean quantization error (MQE) of dataset. The dataset’s MQE value will be used as an important parameter for judging whether horizontal expansion of a neuron map and vertical expansion of a neuron should keep performing or not. In order to save storage space, we train a bloom filter to reduce the size of input dataset. A bloom filter does not store the input elements themselves, it just record the existence of each input data. So we train a bloom filter in each neuron rather than store input elements in it. However, bloom filter can lead to false positive situation. In other words, it is possible to find an input data that does not belongs to the neuron. Fortunately, the false positive rate can be adjust to a very low level. Because there are large amounts of data in our experiment, low false positive rate is acceptable. After we have all the initial information of input dataset, we can start to create a neuron by using these information. For the GHSOM algorithm, layer0 is composed of one neuron map with only one neuron. Thus we use the neuron we just created to create a neuron map, and then use this neuron map to create a GHSOM layer (layer0). Once we have layer0, we can create GHSOMTreeBuilder which can be used to create expander. Finally, we can start to deliver neuron map as a message to expander for training GHSOM model.

C. The GHSOM Tree Builder

Algorithm 2 GHSOMTreeBuilderReceive

1: if msg = GrowthResult then
2: expandedResult ← GetExpandedResult(msg)
3: AppendResult(growthTree, expandedResult)
4: layer ← LayerAtGHSOMTree(expandedResult)
5: if isLayerFinished(layer) then
6: msg ← toMsg(LayerFinished, layer)
7: SendMsgAndForget(msg, self)
8: end if
9: KillActor(sender)
10: else if msg = SaveGHSOMLayer then
11: layer ← LayerAtGHSOMTree(msg)
12: for each neuronMap ∈ GHSOMMaps(layer) do
13: saveAndReleaseBloomFilter(GetBloomFilter(neuronMap))
14: printNeuronMapInfo(neuronMap)
end for
15: nextLayerSize ← 0
16: for each neuronMap ∈ GHSOMMaps(layer) do
17: for each neuron ∈ GetNeurons(neuronMap) do
18: if ShouldHaveChild(neuron) then
19: nextLayerSize ← nextLayerSize + 1
20: end if
21: end for
22: end for
23: nextGHSOMLayer ← GetNextLayerOrCreate(layer, nextLayerSize)
24: AppendResult(growthTree, nextGHSOMLayer)
25: if isFinished(growthTree) then
26: CloseActorSystem()
27: end if
28: end if
29: end if

The GHSOMTreeBuilder initializes with a layer which takes a neuron map with a trained bloom filter as the structure of acceptable input patterns for corresponding layer. Since the size of input patterns may exceed the memory size that caused out of memory exception when trying to load whole input patterns into memory, bloom filter could be the alternative data structure to represent the input pattern already present in it or not. The main duty of GHSOMTreeBuilder is to record training result, preserve the result in disk, and terminate the whole actor system if GHSOM tree matches some specific pattern. Expanders would receive a neuron map as a message to perform expansion algorithm of SOM. Once finished, expanders will send the neuron map as a message after expansion to GHSOMTreeBuilder, and wait for next mission.

As the description above, GHSOMTreeBuilder (Algorithm 2) is an actor that maintains a tree-like data structure in respect of current training progress and waits for some instructions from other actors. The builder waits for two types of messages, GrowthResult (line 1) and SaveGHSOMLayer (line 10). Once builder receives GrowthResult as a message, it would extract a trained neuron map inside the message and append it to the tree (line 3) to reflect the current progress. After appending message to the tree, we get two value: the size and expected size of current layer to be the judge criterion for current layer training progress. Furthermore, builder would check some condition (line 5) and send a message to notify itself if
the GHSOM layer of the received neuron map is fulfilled. For resource management, builder would kill the sender who sends the trained neuron map to free some memory. On the other hand, once the builder gets an order to save an GHSOM layer, it would serialize each neuron map in the layer including the subordinate bloom filter to a file(line 12). Again, for the memory concern, builder would release each bloom filter after serialization. In this stage, builder would initiate the metadata of the next layer (line 20) with the information like how many neuron maps should exist in the next layer as the condition for builder to check when receiving a GrownResult message (line 5). Finally, builder would check whole tree structure to see if it is time to terminate the actor system when the whole tree is fully expanded (line 27).

Algorithm 3 NeuronActorReceive

1. \texttt{weight} ← GetWeight(neuronMeta);
2. \texttt{point} ← GetPoint(neuronMeta);
3. \texttt{mqe} ← 0;
4. \texttt{acceptedDataSize} ← 0;
5. \texttt{bloomFilter} ← BuildBloomFilter(neuronMeta);
6. if msg = CalculateQE then
7. \texttt{msgWeight} ← GetDataWeight(msg);
8. \texttt{distance} ← EuclideanDistance(\texttt{weight}, \texttt{msgWeight});
9. \texttt{SendMsgAndForget}(\texttt{toMsg} \texttt{(point, distance)}, \texttt{sender});
10. \texttt{else if} \texttt{msg = AdaptWeight} then
11. \texttt{winnerPoint} ← \texttt{GetPoint}(msg);
12. \texttt{winnerWeight} ← \texttt{GetDataWeight}(msg);
13. \texttt{adaptedWeight} ← \texttt{UpdateWeight}(\texttt{winnerPoint}, \texttt{point}, \texttt{winnerWeight}, \texttt{weight})
14. \texttt{else if} \texttt{msg = UpdateBloomFilter} then
15. \texttt{msgWeight} ← \texttt{GetDataWeight}(msg);
16. \texttt{BloomFilterUpdate}(\texttt{bloomFilter}, \texttt{msgWeight});
17. \texttt{msg} ← \texttt{msg} × \texttt{EuclideanDistance}(\texttt{weight}, \texttt{msgWeight});
18. \texttt{acceptedDataSize} ← \texttt{acceptedDataSize} + 1;
19. \texttt{else if} \texttt{msg = ShowNeuronInfo} then
20. \texttt{SendMsgAndForget}((\texttt{toMsg} \texttt{(point, weight, mqe, acceptedDataSize, bloomFilter)}, \texttt{sender});
21. \texttt{end if}

D. Expander

Neuron map expander seals the most of original GHSOM algorithm inside (Algorithm 4). As an actor, the only mission of an expander is to apply SOM training algorithm with neurons, which are actors too. For keeping which input patterns were accept by each neuron and the perspective from neurons, the neuron actors instantiate with an empty bloom filter and a weight vector (Algorithm 3). By receiving Expand message as an igniter, expander creates a group of actors as neurons to keep the state of weight (line 4). Besides, a neuron actor takes responsibilities for computing Euclidean distance between its own weight and the selected input pattern (line 10) which was enveloped in a message(line 6) (Algorithm 3) and adapts weight frequently with some parameters on the way of SOM training process (line 10)(Algorithm 3). For each input pattern, expander would ask the winner of neuron, which is the closest one in Euclidean distance, to update the bloom filter upon arrival in a checkpoint during training process (line 21). When it comes to check the MQE of this neuron map, expander sends messages to neuron actors and asks them to report the state of each actors including mqe of each actor (line 28).

Algorithm 4 NeuronMapReceive

1. if \texttt{msg} = \texttt{Expand} then
2. \texttt{superMQE} ← \texttt{GetMQE(neuronMapMeta)};
3. \texttt{filter} ← \texttt{GetBloomFilter(neuronMapMeta)}
4. \texttt{neuronActorList} ← \texttt{BuildActors(neuronMapMeta)}
5. \texttt{learnRate} ← \texttt{InitLearnRate()}
6. \texttt{neighborhoodRate} ← \texttt{InitNeighborRate()}
7. \texttt{shouldRun} ← \texttt{True}
8. \texttt{cycle} ← 0
9. while \texttt{shouldRun} do
10. \texttt{randomWeightFromData} ← \texttt{NextWeight(filter)}
11. \texttt{msg} ← \texttt{toMsg(CalculateQE, randomWeightFromData)}
12. \texttt{distances} ← \texttt{SendMsgAndWait(msg, neuronActorList)}
13. \texttt{winner} ← \texttt{FindWinner(distributions)}
14. \texttt{msg} ← \texttt{toMsg(AdaptWeight, winner, winnerPoint, learnRate)}
15. for each \texttt{neuronActor ∈ neuronActorList} do
16. \texttt{SendMsgAndForget(msg, neuronActor)}
17. \texttt{end for}
18. \texttt{learnRate} ← \texttt{UpdateLearnRate()}
19. \texttt{neighborhoodRate} ← \texttt{UpdateNeighborRate()}
20. if \texttt{hitTrainingCondition} then
21. for each \texttt{weight ∈ dataSet} do
22. if \texttt{filter.contains(weight)} then
23. \texttt{winnerActor} ← \texttt{FindWinner(winner, neuronActorList)}
24. \texttt{msg} ← \texttt{toMsg(UpdateBloomFilter, weight)}
25. \texttt{SendMsgAndForget(msg, winnerActor)}
26. \texttt{end if}
27. \texttt{end for}
28. \texttt{neuronMapMQE} ← \texttt{CalculateMQE(neuronActorList)}
29. \texttt{superneuronMapMQE} ← \texttt{GetNeuronMapMQE(neuronMapMeta)}
30. \texttt{maxCycle} ← \texttt{SizeOf(dataSet) × maxCycleMultiplier}
31. if \texttt{neuronMapMQE} ≤ \texttt{\tau}_1 × \texttt{superneuronMapMQE} or \texttt{cycle} ≥ \texttt{maxCycle} then
32. \texttt{shouldRun} ← \texttt{False}
33. \texttt{else}
34. \texttt{furtherstNeighborPoint} ← \texttt{GetFurthestPoint(maxPoint, neuronMapMeta)}
35. \texttt{neuronMapMeta} ← \texttt{InsertRowOrColumn(neuronMapMeta)}
36. for each \texttt{neuronActor ∈ neuronActorList} do
37. \texttt{KillActor(neuronActor)}
38. \texttt{end for}
39. \texttt{neuronActorList} ← \texttt{BuildActors(neuronMapMeta)}
40. \texttt{end if}
41. \texttt{cycle} ← \texttt{cycle} + 1
42. \texttt{end while}
43. \texttt{layerMQE} ← \texttt{GetLayerMQE(neuronMapMeta)}
44. for each \texttt{neuronMeta ∈ GetNeurons(neuronMapMeta)} do
45. \texttt{neuronMQE} ← \texttt{GetNeuronMQE(neuronMeta)}
46. if \texttt{layerMQE} × \texttt{\tau}_2 < \texttt{neuronMQE} then
47. \texttt{CreateNeuronMapByNeuron(neuronMeta)}
48. \texttt{newExpander} ← \texttt{CreateNeuronMapActor(ghsomTreeActor)}
49. \texttt{msg} ← \texttt{toMsg(ExpandneuronMap, newneuronMapMeta)}
50. \texttt{SendMsgAndForget(msg, newExpander)};
51. \texttt{end if}
52. \texttt{end for}
53. \texttt{msg} ← \texttt{toMsg(GrownResult, neuronMapMeta)}
54. \texttt{SendMsgAndForget(msg, ghsomTreeActor)}
55. \texttt{end if}
56. \texttt{end if}

Algorithm 5 NextWeight(filter)

1. \texttt{lastRecord} ← \texttt{lastRecord} \texttt{or} 0
2. \texttt{dataSetAfterLastRecord} ← \texttt{Filter(dataSet, lastRecord)}
3. for each \texttt{weight ∈ dataSetAfterLastRecord} do
4. if \texttt{filter.contains(weight)} then
5. \texttt{return} \texttt{weight}
6. \texttt{end if}
7. \texttt{end for}

In Algorithm 3, when neuron actors receive CalculateQE message, they calculate the Euclidean distance of the ran-
dom weight from message and its own weight, and then send the distance back to expander. As for neuron actors receive AdaptWeight message, each neuron will update its own weight.

In Algorithm 5, we randomly select an input data which exists in the bloom filter and its weight that has not been processed.

In Algorithm 6, we get neuron’s mqe through ShowNeuronInfo message, then add every neuron’s mqe and divide by neuron’s number in this neuron map. As a result, we can get the MQE of the neuron map. The SOM training process ends when MQE of this neuron map satisfies $\tau_1$ constraint (line 31), if not, the expander creates a column or a row of neuron actors referring to the growing of neuron map and keeps training till it accomplishes. For memory concern, after new neuron map growing, neurons in old neuron map will be killed. Then the expander would evaluate each status of neuron and create a metadata of new neuron map according to the neuron which satisfied $\tau_2$ constraint (line 47). With metadata of new neuron maps, the expander creates more expanders for them and sends the metadata as message to each new expander. Finally, the expander sends the result of the trained neuron map as to GHSOMTreeBuilder and finishes the expansion (line 55).

Algorithm 6 CALCULATEMQE(neuronActorList)
1: layerMQE ← 0
2: count ← 0
3: for each neuronActor ∈ neuronActorList do
4:  msg ← taMsg(ShowNeuronInfo, neuronActor)
5:  neuronQE ← SendMsgAndWait(msg, neuronActor)
6:  if neuronQE > 0 then
7:    layerMQE ← layerMQE + neuronQE
8:  end if
9: end for
10: return layerMQE ÷ count

E. Performance Evaluation

Table II-E shows the performance improvement with parallel-actor computation. We use a 6-core, 32GB device for GHSOMTreeBuilder and two 2-core, 4GB devices for Expander. In all case, by adopting the parallel computation, we improve the performance of the ghsom algorithm compared to the sequential execution. The advantage is more significant when we have more actors against a large size of samples. The computation could be scaled out by increasing the number of actors.

III. MALWARE DETECTION WITH CLUSTERS

We discuss how to derive the detection rules with the clustering results for malware detection in this section. We have implemented the VISO [14] for monitoring online VM behaviors. It collects the sequential system call data of VMs. However, we need find the rules that can be used to distinguish malware from benign ones. Note that malware may have benign behaviors during the execution, and hence it is essential to find the unique periods that are distinguished from others. To this aim, we apply the clustering algorithms presented in this work. We collect executions of various kinds of malware and benign software in VMs. We use types of system calls that are observed in the executions as the attributes of a vector. We split the execution in periods. Each period has the same window, say n system calls, and is treated as an individual sample (labelled with period id and the corresponding malware/benign-ware) in the clustering. The value of the attributes of that sample is then the count of that system call type within the period. That is to say, each sample is represented as a vector with call counts as the attribute values and is associated with a label indicating its malware id and its period index. We then apply the presented distributed GHSOM algorithm to cluster these samples.

For all the clusters that contain samples with benign labels (despite of having samples with malware labels), we treat them as benign behavior clusters. The intuition is that samples (even labeled with malware) that fall in the benign clusters are the periods that may appear in benign executions and cannot be used to distinguish malware from benign ones. For the clusters that have only samples labelled with malware (could be one or many types), we can use them to derive detection rules for the labelled malware. We have three kinds of detection rules. The first one is using a single cluster. The later two kinds are the composition rule of multiple clusters.

- Single-cluster rule: The simplest rule is using a single cluster. We say a sample falls in the cluster if the winning neuron of the given sample is selected from this cluster. An execution of a VM that has any its system call period falls in a malware cluster raises an alarm. Figure 2 shows a sample vector of system call counts that fits a detection rule for Worm and Virus. The most common detected samples are from executions of VMs that execute Win32/Korgo, Win32/Conficker and Win32/Sality family type malware. We will discuss the false positive/negative rates of these detection rules in the experiment section.

- Multi-cluster Rule: When a malware contributes multiple clusters (all these clusters have samples that are labelled with malware), the detection rule is composed form all the clusters. An execution of a VM that has its system call periods fall in all of these malware cluster raises an alarm. That is to say, a VM has to meet all the malware clusters to be considered as a malicious behavior. The restriction can be released as fitting any (rather than all) of the malware cluster in a sound manner. Figure 3 shows a detection rule for the specific malware family type Win32/Conficker.X. If all the malware clusters are met, the execution of that VM is malicious.
• Temporal-cluster rule: When a malware contributes multiple clusters and these clusters have temporal dependency relations, i.e., a cluster happens before another (according to the period id), the detection rule is composed from all the clusters in order. A cluster has to happen before another cluster if all its samples (labelled with the same malware) have smaller period ids than those of the samples in the other cluster. An execution of a VM that has its system call periods fall in all of these malware clusters in the order that meets the happen-before relation raises an alarm. That is to say, a VM has to meet all the malware clusters to be considered as a malicious behavior. The restriction can be released as fitting any (rather than all) of the malware cluster in a sound manner. Figure 4 shows the sequence of malware clusters for the detection of another type of Win32/Conficker.X.

IV. EXPERIMENTS

A. Data Sets of Malware

We evaluate the clustering algorithm against real malware cases in this section. We have tested two malware sets: 1) we download malware executables from the OWL database [20] and record their system call executions with VISO. The number of attributes of these system calls is less than a hundred. 2) we collect execution of Windows APIs from malware executables presented in [21]. These APIs include parameters and are considered to be high-dimensional data (with more than two thousand attributes). For OWL’s data set, we split the executions of 3013 malwares and 16 benign/normal behaving VMs into periods of 1000 system calls as samples. This yields 1,228,904,821 samples in total. We label these samples according to the analysis of VirusTotal [22]. There are 31 different types of malware families in total. We have a large amount of Worm type malware. Others are Trojan and Virus type malware. Our goal is to generate detection rules for these malware families. For the API data, we split the executions into periods of 300 windows API calls as samples. We report the analysis result using GHSOM clusters.

B. A Data Clustering Example

Figure 5 is a sample GHSOM cluster result for the high dimensional API executions collected from 96 malwares and 5 benign programs. The GHSOM consists of 5 layers. There are 6 neurons in layer 1, 20 neurons in layer 2, 34 neurons in layer 3, 34 neurons in layer 4 and 12 neurons in layer 5, with total 106 neurons.

C. Rule Accuracy Evaluation

To evaluate rule accuracy, we report the following rates for each detection rule that is derived from the GHSOM clustering results. For the OWL data set, the GHSOM consists of 3 layers shown in Figure 6. There are 6 neurons in layer 1, 41 neurons in layer 2 and 105 neurons in layer 3, with total 152 neurons. We found 19 malware clusters for the OWL data set and 26 malware clusters for the API data set. The false positive rate is computed by dividing the number of false positives, e.g., a benign execution is recognized as the malware, to the sum of false positives and true positives. The false negative rate is computed by diving the number of false negatives, e.g., no periods of an execution of the malware fit, to the sum of false negatives and true negatives. The false negative rate is the percentage of a detection rules omission. We report the false positive and negative rates for each detection rule of the OWL data set and of the API data set in Figure 7.

D. Accuracy Comparison with K-means

For comparison on K-means and GHSOM, we apply the detection rule derivation using the clustering results of K-means. We use the API executions as training inputs to K-means. The K is set to 106 (according to the GHSOM result). Though we observed that the training time using K-means is much less than GHSOM, we found the clusters have bias such as high similar samples may be clustered in different clusters and samples that fall in the same cluster are not quite similar. As for the detection rule, there are 50 malware clusters that do not contain any benign samples. For each of them, we derive its detection rule accordingly. Figure 8 shows the accuracy evaluation on these detection rules. The figure shows that the presented GHSOM algorithm can help us achieve much lower false rates compared to the K-means rules.

V. CONCLUSION

Unsupervised clustering is needed for data that have limited information. We present a new scalable algorithm for GHSOM constructions with distributed computation in this work and
Fig. 5. The clustering map of API (high-dimensional) data set

Fig. 6. The clustering map of OWL data set

Fig. 7. OWL and API (high-dimension) data accuracy evaluation
evaluate the presented approach against malware executions, showing that unsupervised clustering that separates benign and malicious samples can be applied to extract essential malicious execution periods and derive effective detections rules to prevent malware executions. We also show that adopting K-means with randomly selecting the number of clusters in advance may result in useless clusters.

ACKNOWLEDGMENT

We thank anonymous reviewers for their valuable comments. This work was supported in part by Taiwan Information Security Center (TWISC), Academia Sinica, and Ministry of Science and Technology, Taiwan, under the grant MOST 103-2221-E-004-006-MY3 and 105-2923-E-002-016-MY3.

REFERENCES


