Distributed Novelty Detection at the Edge for IoT Network Security *

Luís Puhl[1 [0000−0003−2118−9992]], Guilherme Weigert Cassales[1 [0000−0003−4029−2047]], Helio Crestana Guardia[1 [0000−0001−5010−747X]], and Hermes Senger[1 [0000−0003−1273−9809]]

Universidade Federal de São Carlos, Brasil
https://www2.ufscar.br/

Abstract. The ongoing implementation of the Internet of Things (IoT) is sharply increasing the number and variety of small devices on edge networks. Likewise, the attack opportunities for hostile agents also increases, requiring more effort from network administrators and strategies to detect and react to those threats. For a network security system to operate in the context of edge and IoT, it has to comply with processing, storage, and energy requirements alongside traditional requirements for stream and network analysis like accuracy and scalability. Using a previously defined architecture (IDSA-IoT), we address the construction and evaluation of a support mechanism for distributed Network Intrusion Detection Systems based on the MINAS Data Stream Novelty Detection algorithm. We discuss the algorithm steps, how it can be deployed in a distributed environment, the impacts on the accuracy and evaluate performance and scalability using a cluster of constrained devices commonly found in IoT scenarios. The obtained results show a negligible accuracy loss in the distributed version but also a small reduction in the execution time using low profile devices. Although not efficient, the parallel version showed to be viable as the proposed granularity provides equivalent accuracy and viable response times.

Keywords: novelty detection · intrusion detection · data streams · distributed system · edge computing · internet of things

1 Introduction

The Internet of Things (IoT) brings together a wide variety of devices, including mobile, wearable, consumer electronics, automotive and sensors of various types. Such devices can either be accessed by users through the Internet or connect to other devices, servers and applications, with little human intervention or supervision [15,1,8,13]. Security and privacy is a major concern in the IoT, especially regarding devices having access to user personal data like location,

* The authors would like to thank Brazilian funding agencies FAPESP and CNPq for the financial support.

health and many other sensitive data [12]. Furthermore, if compromised, such devices can also be used to attack other devices and systems, steal information, cause immediate physical damage or perform various other malicious acts [9]. As an additional concern, IoT devices likely have a long lifespan, less frequent software patches, growing diversity of technologies combined with lack of control over the software and hardware of such devices by the host organization (where they are deployed), which considerably increases the attack surface.

Because most IoT devices have limited resources (i.e., battery, processing, memory and bandwidth), configurable and expensive algorithm-based security techniques are not usual, giving way to network based approaches [17]. Machine Learning (ML) techniques, for instance, have been studied for years to detect attacks from known patterns or to discover new attacks at an early stage [2,11]. A recent survey [15] shows that ML based methods are a promising alternative which can provide potential security tools for the IoT network making them more reliable and accessible than before.

Despite the promising use of ML to secure IoT systems, studies found in the literature [2,11,15] are limited to traditional ML methods that use static models of traffic behavior. Most existing ML solutions for network-based intrusion detection cannot maintain their reliability over time when facing evolving attacks [16,10]. Unlike traditional methods, stream mining algorithms can be applied to intrusion detection with several advantages, such as: (*i*) processing traffic data with a single read; (*ii*) working with limited memory (allowing the implementation in small devices commonly employed in edge services); (*iii*) producing real-time response; and (*iv*) detecting novelty and changes in concepts already learned.

Given the recent [16,10,4] use of Data Stream Novelty Detection (DSND) in network data streams, this paper shows the effects of adapting these mechanisms to edge services for use in IoT environments. Our proposal, called *MFOG*, adapted the IDSA-IoT architecture [3] using the DSND algorithm MINAS [5,7], making it suitable to run on a distributed system composed of small devices with limited resources on the edge of the network. Using our newer version of the MINAS algorithm, we have experimentally evaluated how the distribution affects the capability to detect changes (novelty) in traffic patterns and its impact on the computational efficiency. Finally, some distribution strategies and policies for the data stream novelty detection system are discussed.

This paper is organized as follows: Section 2 reviews the chosen DSND algorithm MINAS. A distributed extension of MINAS, including its implementation and evaluation are presented in Section 3 and in Section 4 we show how we evaluated *MFOG* and the discuss results we found. Finally, Section 5 summarizes the main findings and presents possible future work.

2 MINAS

MINAS [5,7] is an offline-online DSND algorithm, meaning it has two distinct phases. The first phase (offline) creates an initial model set with several clusters

based on a clustering algorithm with a labeled training set. Each cluster can be associated with only one class of the problem, but each class can have many clusters.

During its online phase, which is the main focus of our work, MINAS performs three tasks in (near) real-time over a potentially infinite data stream: classification, novelty detection and model update, as shown in Algorithm 1.

MINAS attempts to classify each incoming unlabeled instance according to the current decision model. Instances not explained by the current model receive the *"unknown"* label and are stored in the unknowns-buffer. When the unknowns-buffer size reaches a preset threshold, MINAS executes the Novelty Detection function. After a set interval, samples in the unknowns-buffer are considered to be noise or outliers and removed. The algorithm also has a mechanism to forget clusters that became obsolete and unrepresentative of the current data stream distribution, removing them from the Model and storing in a Sleep Model for recurring patterns detection [7].

Input: ModelSet, inputStream
Output: outputStream
Parameters: cleaningWindow, noveltyDetectionTrigger

```
1  Function MinasOnline(Model, inputStream):
2      UnknownsBuffer ← ∅; SleepModel ← ∅ ;
3      lastCleanup ← 0; noveltyIndex ← 0;
4      foreach sample_i ∈ inputStream do
5          nearest ← nearestCluster (sample, Model);
6          if nearest.distance ≤ nearest.cluster.radius then
7              sample.label ← nearest.cluster.label;
8              nearest.cluster.lastUsed ← i ;
9          else
10             sample.label ← unknown;
11             UnknownsBuffer ← UnknownsBuffer ∪ sample;
12             if | UnknownsBuffer | ≥ noveltyDetectionTrigger then
13                 novelties ← NoveltyDetection (Model ∪ SleepModel,
                     *UnknownsBuffer);
14                 Model ← Model ∪ novelties;
15             if i > ( lastCleanup + cleaningWindow) then
16                 Model ← moveToSleep (Model, *SleepModel, lastCleanup);
17                 UnknownsBuffer ← removeOldSamples (UnknownsBuffer, lastCleanup);
18                 lastCleanup ← i;
19         outputStream.append(sample);
```

Algorithm 1: Our interpretation of MINAS [7].

The Novelty Detection function, illustrated in Algorithm 2, groups the instances to form new clusters, and each new cluster is validated to discard the non-cohesive or unrepresentative ones. Valid clusters are analyzed to decide if they represent an extension of a known pattern or a completely new pattern. In

both cases, the model absorbs the valid clusters and starts using them to classify new instances.

```
   Parameters: minExamplesPerCluster, noveltyFactor
 1 Function NoveltyDetection(Model, Unknowns):
 2 |  newModelSet ← ∅;
 3 |  foreach new in clustering (Unknowns) do
 4 |  |  if ( |new.sampleSet| ≥ minExamplesPerCluster ) ∧ (new.silhouette > 0)
   |  |   then
 5 |  |  nearest ← nearestCluster (new, Model);
 6 |  |  if nearest.distance < (nearest.cluster.radius × noveltyFactor) then
 7 |  |  |  new.label ← nearest.cluster.label;
 8 |  |  |  new.type ← "extension";
 9 |  |  else
10 |  |  |  new.label ← noveltyIndex;
11 |  |  |  noveltyIndex ← noveltyIndex +1;
12 |  |  |  new.type ← "novelty";
13 |  |  Unknowns ← Unknowns − new.sampleSet;
14 |  |  newModelSet ← newModelSet ∪ new;
15 |  return newModelSet;
```

Algorithm 2: MINAS [7] Novelty Detection task.

3 Proposal

In this work we investigate an appropriate architecture for performing DSND at the edge, allowing small IoT devices to detect undesirable network behavior. To that end, we propose and evaluate *MFOG*, a distributed DSND system following the IDSA-IoT architecture [3] and based on a distributed version of the algorithm MINAS [7]. Our approach explores distributed computing and a trivial load balancer to enable low profile devices to classify and detect unwanted traffic in a scalable, edge focused way.

However, given the distributed nature and the typical use of small computing devices in IoT scenarios as well as the need handle network speeds, new challenges arise: (*i*) the classification phase of the algorithm must occur in parallel at different nodes; (*ii*) the novelty detection phase, which provides the model evolution, must also be asynchronous; (*iii*) the algorithm complexity (time and space) must allow it to be processed by modest computing devices (i.e., small memory and low processor performance).

NIDS monitor network traffic, and analyze the characteristics of each flow to identify any intrusion or misbehavior. However, this problem requires both fast and accurate response [4]: fast response is needed to have a proper reaction before harm can be cast to the network and to cope with the traffic without imposing loss or delay in the NIDS or observed network; accurate response is

required as not to misidentify, especially the case of false positive that leads to false alarms. To achieve those goals, we leverage fog computing.

In common IoT scenarios, data is captured by small devices and sent to the cloud for any compute or storage tasks, but this is not feasible in a NIDS scenario. Fog computing infrastructure aims to offload processing from the cloud providers by placing edge devices closer to end-users and/or data sources.

In our proposal, fog and cloud computing resources are combined to minimize the time elapsed between a flow descriptor ingestion and intrusion alarm, performing the classification step of MINAS running multiple classifier instances. After the initial classification, the resulting label can be used immediately, but if the sample is labeled as *unknown*, this sample must be stored and the novelty detection step will be triggered.

To have a better overview of our proposal and how it integrates with existing IoT environments, Figure 1 depicts such scenario showing from bottom to top: IoT devices directly connected to a (local) gateway network; this gateway network could be as simple as a single Internet router or be more complex by connecting to private clouds or containing more devices providing fog computing capabilities; lastly, available over the internet, the traditional public cloud provides inexpensive computing and storage on demand. In this scenario, the further apart resources are, the more network resources need to be employed, and, as with any networked system, the higher is the latency.

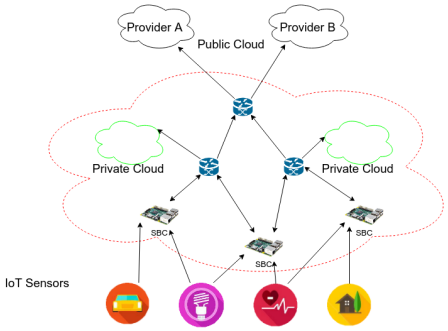


Fig. 1: IDSA-IoT [3] physical architecture and deployment scenario overview.

The overall *MFOG* architecture has two main modules, Classification and Novelty Detection, which implement the MINAS main tasks. The Classification Module performs the same task of the MINAS Online phase and is the focal point for parallelism and distribution in our proposal. It is replicated in the fog and runs on each cluster node, using a configurable number of threads (limited to the node CPU core count).

The Novelty Detection Module can also be replicated, the choice being one instance per local network, one global cloud instance, or both. This module also handles the homonymous task of MINAS Online phase, receiving all the samples

labeled with *unknown*, storing them in an internal *unknown-buffer*, and, when this buffer is full, performing the MINAS Novelty Detection task (clustering followed by validation).

3.1 Policies

The design of our distributed DSND architecture includes partitioning the functionalities of MINAS and establishing the appropriate data flows between different actors. Changes to placement and behavior can have different impacts and should be chosen with care. The decisions following these discussions can be organized in several policies, some of them were recurring during our implementation discussions and are:

- Regarding the allocation of the Novelty Detection Module:
  - At each fog node: patterns will be only detected if sufficient samples of them occur in the local observed network, use of the local node processing power, and a model synchronization mechanism between networks must be added;
  - In the cloud: detect patterns even when scattered on each local network, each sample with *unknown* label must be sent from edge to cloud implying increased internet link usage and increased delay between the appearance of a pattern, its detection and propagation to fog classifiers;
  - On both: local *unknown* buffer is maintained and novelty detection is local as well, once a sample is considered as noise or outlier it shall be sent to the cloud where the process repeats but with global data. This choice needs an even more complex model synchronization mechanism.
- Regarding the model cleanup (forget mechanism): Even when a global novelty detection is used, local models can be optimized for faster classification using the local model statistics by sorting by (or removing) least used clusters;
- Lastly, reclassification of *unknowns*: In the novelty detection task in MINAS, the *unknown-buffer* is effectively classified using the new set of clusters. In Algorithm 2 line 13, the new valid cluster (novelty or extension) includes the set of samples composing that cluster, thus, if this new label assignment was put forth to the system output it would introduce delayed outputs, more recent and perhaps more accurate. Also, it would change the system data stream behavior from a *map* (meaning each input has one output) to a *flatMap* (each input can have many outputs).

3.2 Implementation

The original MINAS algorithm has a companion implementation[1] (*Ref*) written in Java using MOA library base algorithms such as K-means and CluStream, but our implementation only used K-means. Another difference between *Ref* and

[1] Available at `http://www.facom.ufu.br/~elaine/MINAS`.

MFOG is the definition of cluster radius derived from the distances of elements forming the cluster and the cluster's center. *Ref* uses the maximum distance while *MFOG* uses the standard deviation of all distances as described in [7].

The stream formats for input and output are also of note. As input, the algorithm takes samples ($\vec{v}$), which are a sequence of numbers with dimension d. In addition to $\vec{v}$, for both training and evaluation, the class identifier is provided as a single character, along with a unique item identifier (*uid*), which can otherwise be determined from the sample index in the input stream.

As its output, the algorithm returns the original sample $\vec{v}$ followed by the assigned label. Adjustments can easily be made to provide the output results as a tuple containing *uid* and the assigned label.

For evaluation purposes, an *MFOG* implementation[2] was made using MPI (*Open MPI 4.0.4*). The program is organized in a single program multiple data (SPMD) programming model, so a single version of the *MFOG* program was initiated on all nodes, being that one of them would perform the root role, while the others ran as leaves, the program entry point is illustrated on Algorithm 3 and the overall sequence of interactions is shown in Figure 2.

Each leaf node runs a model adjustment thread and multiple (up to the number of cores) classifier threads. The leaf tasks are illustrated in Algorithm 4.

On the root process, illustrated in Algorithm 5, a sampler thread is responsible for distributing the sampled flow information ($\vec{v}$) to the classifier nodes, using a round-robin load balancing scheme. The other thread on the root process is responsible for receiving the classification results and for processing the unknown samples in the search for novelties.

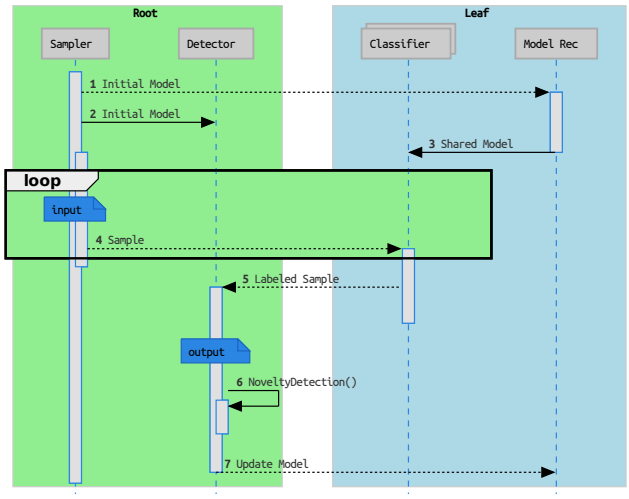


Fig. 2: *MFOG* life line overview.

[2] Available at `https://github.com/luis-puhl/minas-flink`.

```
   Parameters: mpiRank
   Input: inputStream
   Output: outputStream
1  Function Mfog(inputStream, outputStream):
2  |  Model ← ∅; ModelLock ← new Lock ();
3  |  if mpiRank = 0 then root
4  |  |  new Thread (Detector, [outputStream, Model, ModelLock]);
5  |  |  Sampler (inputStream, Model, ModelLock);
6  |  else leaf
7  |  |  new Thread (modelReceiver, [Model, ModelLock]);
8  |  |  Classifier (Model, ModelLock);
```

Algorithm 3: *MFOG*: main MPI entry-point.

```
1  Function Classifier(Model, ModelLock):
2  |  while True do
3  |  |  sampe ← receive (SampleType, root);
4  |  |  if sample = EndOfStream then break;
5  |  |  sample.label ← "unknown";
6  |  |  with readLock (ModelLock)
7  |  |  |  nearest ← nearestCluster (sample, Model);
8  |  |  if nearest.distance ≤ nearest.cluster.radius then
9  |  |  |  sample.label ← nearest.cluster.label;
10 |  |  send (root, SampleType, sample);
11 Function modelReceiver(Model, ModelLock):
12 |  while True do
13 |  |  cl ← receive (ClusterType, root);
14 |  |  if cl = EndOfStream then break;
15 |  |  with writeLock(ModelLock)
16 |  |  |  Model ← Model ∪ cl;
```

Algorithm 4: *MFOG* Leaf Tasks: Model Receiver and Classifier.

4 Experiments and Results

Aiming to evaluate our proposal for the effects of distributed novelty detection in a IoT NIDS scenario, we implemented an experimental setup, composed of three Raspberry Pi 3 model B single board computers connected via Ethernet Switch. The idea was to create a simple cluster simulating an IoT network with constrained resources at the edge of the network. This cluster stored all source code, binaries (compiled and linked in place) and data sets. In our setup, the data set is stored in the root's node SD card and is read for each experiment. All experiments were executed in this cluster for isolation of otherwise unforeseen variations and for safe software comparison with constant hardware.

The data set used is the December 2015 segment of Kyoto 2006+ data set[3] (Traffic Data from Kyoto University's Honeypots) [14] containing 7 865 245 sam-

[3] Available at `http://www.takakura.com/Kyoto_data/`.

```
   Parameters: mpiSize
 1 Function Sampler(inputStream, Model, ModelLock):
 2 |  dest ← 1;
 3 |  foreach sample ∈ inputStream do
 4 |  |  if typeOf (sample) is Cluster then
 5 |  |  |  broadcast (ClusterType, sample, root);
 6 |  |  |  with writeLock (ModelLock)
 7 |  |  |  |  Model ← Model ∪ sample;
 8 |  |  |  continue;
 9 |  |  send (dest, SampleType, sample);
10 |  |  dest ← dest +1;
11 |  |  if dest > mpiSize then dest ← 1;
   Parameters: cleaningWindow, noveltyDetectionTrigger
12 Function Detector(outputStream, Model, ModelLock):
13 |  UnknownSet ← ∅; lastCleanup ← 0;
14 |  while True do
15 |  |  sample ← receive (SampleType, any);
16 |  |  if sample = EndOfStream then break;
17 |  |  outputStream.append(sample);
18 |  |  if sample.label = "unknown" then
19 |  |  |  UnknownSet ← UnknownSet ∪ sample;
20 |  |  |  if | UnknownSet | ≥ noveltyDetectionTrigger then
21 |  |  |  |  novelties ← NoveltyDetection (Model, *UnknownSet);
22 |  |  |  |  with writeLock (ModelLock)
23 |  |  |  |  |  Model ← Model ∪ novelties;
24 |  |  |  |  foreach cluster ∈ novelties do
25 |  |  |  |  |  broadcast (ClusterType, cluster, root);
26 |  |  |  if sample.uid > ( lastCleanup + cleaningWindow) then
27 |  |  |  |  UnknownSet ← removeOldSamples (UnknownSet, lastCleanup);
28 |  |  |  |  lastCleanup ← sample.uid;
```

Algorithm 5: *MFOG* Root Tasks: Sampler and Detector.

ples. From the original data set, we filtered only samples associated with normal traffic or known attack types identified by existing NIDS, and attack types with more than 10 000 samples for significance, as previously done by [3]. The remaining samples then were normalized so each feature value space (e.g., IP Address, Duration, Service) is translated to the Real interval $[0, 1]$.

The resulting derived data set is then stored in two sets, training set and test set, using the holdout technique. However, for the training set we filter in only normal class resulting in 72 000 instances. For the test set we use 653 457 instances with 206 278 instances with "N" (normal) class and 447 179 instances with "A" (attack) class. Note that this choice results in overfitting for the normal class and, under-fitting for the attack class as the system first needs to detect a novel class and then add it to the model.

4.1 Measurements and Visualizations

We have used two types of evaluation measurements for each experiment: a measure of the full experiment execution time and, a set of qualitative measurements extracted by a Python script.

Our evaluation script was build following reference techniques like multi-class confusion matrix with label-class association [7] to extract classification quality measurements. This script takes two inputs, the test data set and the captured output stream, and gives as outputs the confusion matrix, label-class association, final quality summary with: *Hits* (true positive), *Misses* (Err), *Unknowns* (UnkR); and stream visualization chart with per example instance summary with novelty label markers.

In the confusion matrix $M = m_{ij} \in \mathbb{N}^{c \times l}$, computed by our evaluation script, each row denotes the actual class c and each column denotes the predicted label l present in the captured output stream. Thus, each cell $M_{c,l}$ contains the count of examples from the test data set of class c, found in the output stream with the label l assigned by the experiment under evaluation.

For the data set under use, original classes are $c \in \{N, A\}$, and for the labels we have the training class *"N"*, *unknown* label *"-"* and the novelties $i \in \mathbb{N}$.

Added to the original confusion matrix M are the rows *Assigned* and *Hits*. *Assigned* row represents which original class c (or if *unknown*, *"-"*) the label l is assigned to, this is computed by using the original class if $c = l$ or by associated novelty label to original class as described in [6] section 4.1 (class from where the most samples came from). *Hits* row shows the true positive count for each label l with assigned class c, being the same value as cell $M_{c,l}$. The *Hits* row is also used to compute the overall true positive in the summary table and stream visualization chart. One complete matrix is shown in Tab. 1a.

For the measurements summary table, six measurements from two sources are displayed. Three measures *Hits*, *Unknowns* and *Misses* represented as ratio of the captured output stream, extracted from the evaluation python program, computed as follows: *Hits* (true positive rate) is the sum of the *Hits* row in the extended confusion matrix; *Unknowns* is the count of examples in the captured output stream marked with the *unknown* label (*"-"*); *Misses* is the count of all examples in the captured output stream marked with a label distinct from the *Assigned* original class and are not marked as unknown.

Furthermore in the measurement summary table, *Time*, *System* and *Elapsed* represented in seconds, are extracted from *GNU Time 1.9*. *Time* is the amount of CPU seconds expended in user-mode (indicates time used doing CPU intensive computing, e.g., math). *System* is the amount of CPU seconds expended in kernel-mode (for our case, it indicates time doing input or output). *Elapsed* is the real-world (wall clock) elapsed time and indicates how long the program took to complete. Our four main experiments are shown in Tab. 2.

Lastly, the stream visualization chart shows the summary quality measurement (*Hits*, *Unknowns*, *Misses*) computed for each example in the captured output stream. The Horizontal axis (x, domain) plots the index of the example

Table 1: Confusion Matrices and Qualitative measurements

(a) Reference implementation

Labels	-	N	1	2	3	4	5	6	7	8	9	10	11	12
Classes														
A	3774	438750	123	145	368	8	52	165	1	1046	161	2489	71	26
N	8206	193030	0	79	44	0	0	0	229	181	154	4066	289	0
Assigned	-	N	A	A	A	A	A	A	N	A	A	N	N	A
Hits	0	193030	123	145	368	8	52	165	229	1046	161	4066	289	26

(b) Sequential implementation

Labels	-	N	0	1	2	4	5	6	7	8	10
Classes											
A	16086	429765	94	995	104	0	23	3	29	46	34
N	12481	193642	3	94	0	47	0	0	0	11	0
Assigned	-	N	A	A	A	N	A	A	A	A	A
Hits	0	193642	94	995	104	47	23	3	29	46	34

(c) Parallel single-node

Labels	-	N	0	1	2	3	4
Classes							
A	12282	433797	147	952	0	0	1
N	3088	203019	40	99	27	5	0
Assigned	-	N	A	A	N	N	A
Hits	0	203019	147	952	27	5	1

(d) Parallel multi-node

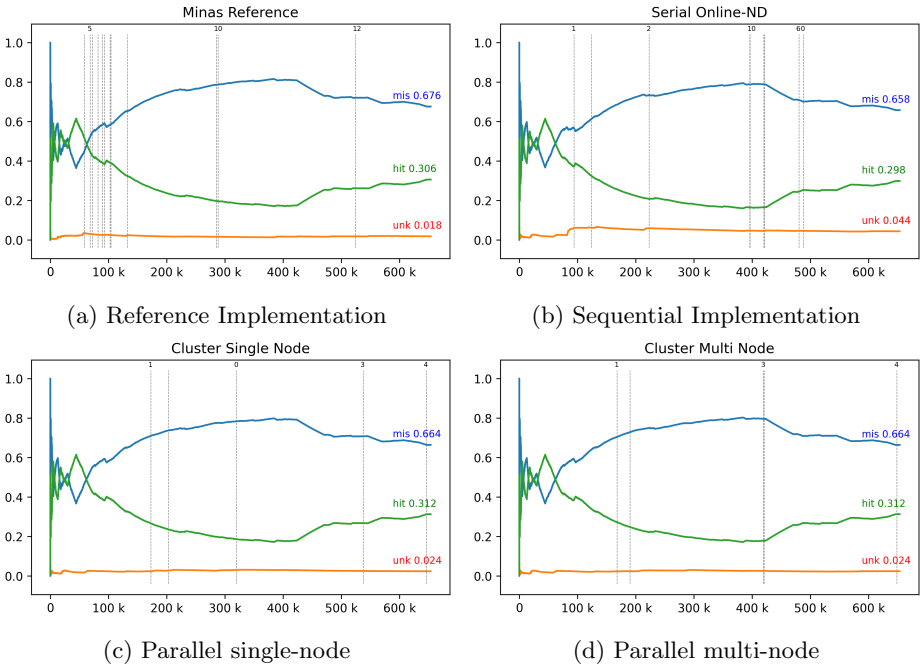
Labels	-	N	0	1	2	3	4
Classes							
A	12378	433631	117	886	0	162	5
N	3121	202916	40	96	105	0	0
Assigned	-	N	A	A	N	A	A
Hits	0	202916	117	886	105	162	5

and the vertical axis (y, image) shows the measurement computed until that example index on the captured output stream.

Adding to the stream visualization chart, novelty label markers are represented as vertical lines indicating *when* in the captured output stream a new label first appeared. Some of the novelty label markers include the label itself ($l \in \mathbb{N}$) for reference (showing every label would turn this feature unreadable due to overlapping). Figure 3 shows complete stream visualization charts.

4.2 Discussion

Four main experiments are presented for discussion: (a) reference implementation of MINAS (*Ref*) [7]; (b) new implementation in sequential mode; (c) new implementation in single-node, multi-process mode (1×4) and (d) new implementation in multi-node, multi-process mode (3×4). Each experiment uses the adequate binary executable, initial model (or training set for *Ref*) and test set to compute a resulting output stream which is stored for qualitative evaluation. The summary of all four experiments is shown in Table 2.



(a) Reference Implementation

(b) Sequential Implementation

(c) Parallel single-node

(d) Parallel multi-node

Fig. 3: Stream hits and novelties visualization.

The comparison of the first two experiments (*a* and *b*) provides a validation for our implementation, while the latter three (*b*, *c* and *d*) serve as showcase for the effects of distribution.

As stated, to validate our implementation we have compared it to *Ref* (the original MINAS companion implementation), so we extracted the same measurements using same process for both *a* and *b*, which can be viewed in Tables 1a, 1b and for ease of comparison in Table 2 the summary can be compared side by side.

In general, the observed classification quality measurements are very similar, and only diverge slightly where *a* has more *Hits* and *Misses* whereas *b* shifted those to *Unknowns*. This phenomenon was watched very closely during development and we found that it was due to small changes to MINAS parameters, MINAS internals like K-means ordering, cluster edge inclusion and cluster radius formula as stated in Subsection 3.2.

As for the time measurements in Table 2 our implementation used less time to analyze the test data set. This is mostly due to the stop condition on the internal K-means algorithm; while *Ref* uses a fixed iteration limit of 100, our implementations adds the "no improvement" check and stops earlier in most cases, which in turn reduces the time taken on the *NoveltyDetection* function. There are also small optimizations on the *nearestCluster* function (minimal distance from sample to cluster center in the set) affecting the *classifier* task and

Table 2: Collected Measures Summary.

Experiment Metric	*Ref* (a)	Offline	Sequential (b)	Single Node (c)	Multi Node (d)
unk	11980		28567	15370	15499
	0.018333		0.043717	0.023521	0.023718
hit	199708		195017	204151	204191
	0.305618		0.298438	0.312416	0.312478
err	441769		429873	433936	433767
	0.676049		0.657843	0.664061	0.663802
Time (s)	2761.83	194.12	80.79	522.10	207.14
System (s)	7.15	0.075	11.51	47.77	157.61
Elapsed (s)	2772.07	194.27	93.03	145.04	95.38
Latency (s)	$4.24 \cdot 10^{-3}$		$1.42 \cdot 10^{-4}$	$2.22 \cdot 10^{-4}$	$1.46 \cdot 10^{-4}$
Processors	1	1	1	4	12
Speedup				0.6414092	0.9753617
Efficiency				0.1603523	0.0812801

NoveltyDetection function. One can also note that *Ref* time in *a* includes the Offline phase while our implementation runs it once and reuses the initial model for *b*, *c* and *d*. In the table the offline time this is shown as a separate column.

As for the effects of running the classification processes on the small devices as MPI nodes with our implementation, we observe an increase of time when we go from 1 to 4 instances in a single node (*b* and *c* respectively), hinting that our choice of load distribution is not as effective as we expected. Further experiments were conducted with the number of instances varying from 1 (sequential) to 12 (3 nodes with 4 CPUs each), but that caused no impact on the true positive rate (*Hits*) and elapsed time. More detailed time measurements can be seen in Figure 4, where we observe near constant time for *elapsed* (near 100s), the *system* increases gradually while *user* decreases at the same rate. We interpret this behavior as a display of potential for gains using a better load balancing than our choice of round-robin such as micro-batching for better *compute-to-communication ratio* (CCR). In general, Figure 4 shows no speedup but also no penalty for scaling to more than 4 instances.

Nevertheless, we can also show the effects of delay in the Classify, Novelty Detection, Model Update and Classify feedback loop. Comparing *b* and *c* we observe a reduction in Novelty labels on the Confusion Matrix (tabs. 1b and 1c) from 9 to 5. The same effect is observed on the stream visualization (figs. 3b and 3c) where our sequential implementation has fewer novelty markers, and they appear later, but the measures keep the same "shape". Comparing *c* and *d* the difference is even smaller, (figs. 3b and 3c) as they both suffer the expected delay in the feedback loop due to asynchronous task execution.

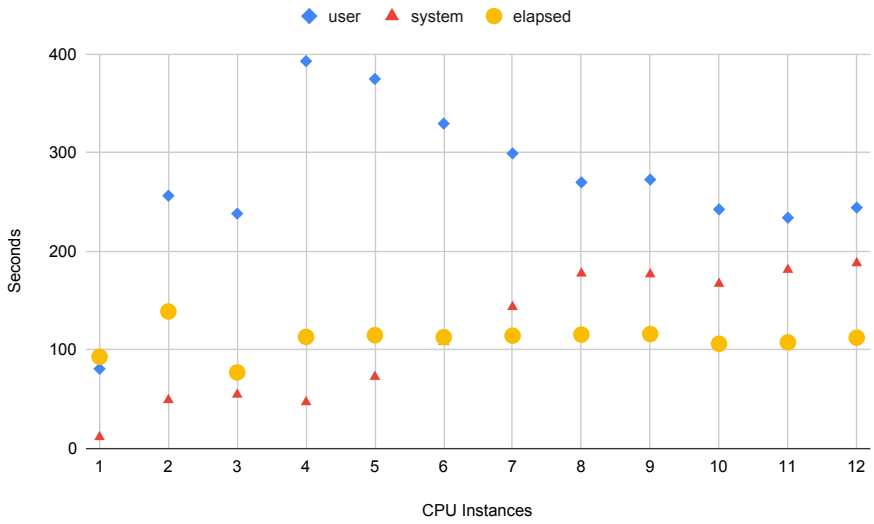


Fig. 4: Time measurements per added instance.

5 Conclusion

Data Stream Novelty Detection (DSND) can be a useful mechanism for Network Intrusion Detection (NIDS) in IoT environments. It can also serve other related applications of DSND using continuous network or system behavior monitoring and analysis. Regarding the tremendous amount of data that must be processed in the flow analysis for DSND, it is relevant that this processing takes place at the edge of the network. However, one relevant shortcoming of the IoT, in this case, is the reduced processing capacity of such edge devices.

In this sense, we have put together and evaluated a distributed architecture for performing DSND applied at network flow descriptors at the edge. Our proposal, *MFOG*, is a distributed DSND implementation based on the MINAS algorithm and the main goal of this work is to observe the effects of our approach to a previously sequential only algorithm, especially in regards to time and quality metrics.

While there is some impact on the predictive metrics, this is not reflected on overall classification quality metrics indicating that distribution of MINAS has a negligible loss of accuracy. In regards to time and scale, our distributed executions was faster than the previous sequential implementation of MINAS, but efficient data distribution was not achieved as the observed time with each added node remained near constant.

Overall, *MFOG* and the idea of using distributed flow classification and novelty detection while minimizing memory usage to fit in smaller devices at the edge of the network is a viable and promising solution. Further work include the

investigation of other DSND algorithms, other clustering algorithms in MINAS and analysis of varying load balancing strategies.

Acknowledgment

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, and Programa Institucional de Internacionalização – CAPES-PrInt UFSCar (Contract 88887.373234/2019-00). Authors also thank Stic AMSUD (project 20-STIC-09), FAPESP (contract numbers 2018/22979-2, and 2015/24461-2) and CNPq (Contract 167345/2018-4) for their support.

References

1. Abane, A., Muhlethaler, P., Bouzefrane, S., Battou, A.: Modeling and improving named data networking over ieee 802.15.4. In: 2019 8th International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN). pp. 1–6 (2019). https://doi.org/10.23919/PEMWN47208.2019.8986906
2. Buczak, A.L., Guven, E.: A survey of data mining and machine learning methods for cyber security intrusion detection. IEEE Communications Surveys & Tutorials **18**(2), 1153–1176 (2016)
3. Cassales, G.W., Senger, H., DE FARIA, E.R., Bifet, A.: Idsa-iot: An intrusion detection system architecture for iot networks. In: 2019 IEEE Symposium on Computers and Communications (ISCC). pp. 1–7 (June 2019). https://doi.org/10.1109/ISCC47284.2019.8969609, `https://ieeexplore.ieee.org/document/8969609/`
4. da Costa, K.A., Papa, J.P., Lisboa, C.O., Munoz, R., de Albuquerque, V.H.C.: Internet of things: A survey on machine learning-based intrusion detection approaches. Computer Networks **151**, 147–157 (2019). https://doi.org/10.1016/j.comnet.2019.01.023
5. Faria, E.R., Gama, J.a., Carvalho, A.C.P.L.F.: Novelty detection algorithm for data streams multi-class problems. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. p. 795–800. SAC '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2480362.2480515, `https://doi.org/10.1145/2480362.2480515`
6. de Faria, E.R., Gonçaalves, I.R., Gama, J., Carvalho, A.C.P.d.L.F.: Evaluation of multiclass novelty detection algorithms for data streams. IEEE Transactions on Knowledge and Data Engineering **27**(11), 2961–2973 (nov 2015). https://doi.org/10.1109/TKDE.2015.2441713, `http://ieeexplore.ieee.org/document/7118190/`
7. de Faria, E.R., Ponce de Leon Ferreira Carvalho, A.C., Gama, J.: Minas: multiclass learning algorithm for novelty detection in data streams. Data Mining and Knowledge Discovery **30**(3), 640–680 (May 2016). https://doi.org/10.1007/s10618-015-0433-y, `https://doi.org/10.1007/s10618-015-0433-y`
8. HaddadPajouh, H., Dehghantanha, A., Parizi, R.M., Aledhari, M., Karimipour, H.: A survey on internet of things security: Requirements, challenges, and solutions. Internet of Things p. 100129 (2019)

9. Kolias, C., Kambourakis, G., Stavrou, A., Voas, J.: DDoS in the IoT: Mirai and Other Botnets. Computer **50**(7), 80–84 (2017). https://doi.org/10.1109/MC.2017.201, http://ieeexplore.ieee.org/document/7971869/
10. Lopez, M.A., Duarte, O.C.M.B., Pujolle, G.: A monitoring and threat detection system using stream processing as a virtual function for big data. In: Anais Estendidos do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos. pp. 209–216. SBC, Porto Alegre, RS, Brasil (2019), https://sol.sbc.org.br/index.php/sbrc_estendido/article/view/7789
11. Mitchell, R., Chen, I.R.: A survey of intrusion detection techniques for cyber-physical systems. ACM Computing Surveys (CSUR) **46**(4), 55 (2014)
12. Sengupta, J., Ruj, S., Bit, S.D.: A comprehensive survey on attacks, security issues and blockchain solutions for iot and iiot. Journal of Network and Computer Applications **149**, 102481 (2020)
13. Shanbhag, R., Shankarmani, R.: Architecture for internet of things to minimize human intervention. 2015 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2015 pp. 2348–2353 (2015). https://doi.org/10.1109/ICACCI.2015.7275969
14. Song, J., Takakura, H., Okabe, Y., Eto, M., Inoue, D., Nakao, K.: Statistical analysis of honeypot data and building of kyoto 2006+ dataset for nids evaluation. Proceedings of the 1st Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, BADGERS 2011 pp. 29–36 (2011). https://doi.org/10.1145/1978672.1978676
15. Tahsien, S.M., Karimipour, H., Spachos, P.: Machine learning based solutions for security of internet of things (iot): A survey. Journal of Network and Computer Applications **161**(November 2019) (2020). https://doi.org/10.1016/j.jnca.2020.102630
16. Viegas, E., Santin, A., Bessani, A., Neves, N.: Bigflow: Real-time and reliable anomaly-based intrusion detection for high-speed networks. Future Generation Computer Systems **93**, 473 – 485 (2019). https://doi.org/https://doi.org/10.1016/j.future.2018.09.051, http://www.sciencedirect.com/science/article/pii/S0167739X18307635
17. Zhou, J., Cao, Z., Dong, X., Vasilakos, A.V.: Security and privacy for cloud-based iot: Challenges. IEEE Communications Magazine **55**(1), 26–33 (2017). https://doi.org/10.1109/MCOM.2017.1600363CM