IoT Data Stream Novelty Detection: Design, Implementation and Evaluation [DRAFT] *

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

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 and, following this increase, the attack opportunities for hostile agents also increases, requiring more from the network administrator role and the need for tools to detect and react to those threats. One such tool are the Network Intrusion Detection Systems (NIDS) which captures and analyses network traffic, acting when a known attack or a new pattern is detected. For a network security tool to operate in the context of edge and IoT it has to comply with processing time, storage space and energy requirements alongside traditional requirements for stream and network analysis like accuracy and scalability. This work addresses the construction and evaluation of a prototype distributed IDS using a Novelty Detection (ND) algorithm. Using a previously defined IDSA-IoT architecture, we discuss the algorithm steps, how it can be deployed in a distributed environment, the impacts on the accuracy of the ND and evaluate its performance and scalability using a cluster of constrained devices commonly found in IoT scenarios. We found an increase of *A 0.0* processed network flow descriptors per core added to the cluster. Also *B 0.0%* and *C 0.0%* change in *F1Score* in the tested datasets when stream was unlimited in speed and limited to *0.0 z MB/s* respectively.

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 running on the back-end with little human intervention or supervision [**?**,**?**,**?**,**?**].

Security and privacy is a major concern in the IoT, specially regarding devices like smart home assistants and wearable devices have access to user related

* Thanks to CNPq and FAPESP for the support.

information, including location, health data, and many other sensitive data [?]. 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. As an additional concern, IoT devices likely have long lifespan and less frequent software patches, and the ever growing mesh of devices of diverse technologies considerably increases the attack surface.

A study reveals that nearly 20% of organizations have experienced at least one IoT-based attack in the last three years. The study shows that most organizations have no control over the origin and nature of software and hardware used by the connected devices. To protect against these threats, the IoT industry's spending on security is estimated to be around USD $ 3.1 billion in 2021 [?], which includes the development of tools and services to improve asset discovery and management, software security evaluation, hardware and penetration testing.

muito próximo de [?] Because most IoT devices have limited resources (i.e., battery, processing power, bandwidth, and memory), configurable and expensive algorithm-based security techniques may not fit [?]. Machine Learning (ML) techniques have been studied for years to detect attacks from known patterns or to discover new attacks at an early stage [?,?]. A recent survey in [?] shows that ML based methods are a promising alternative which can provide potential security tools for the IoT devices which make them more reliable and accessible than before. Despite the promising use of ML to secure IoT systems, studies presented in the literature [?,?,?] 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 [?]. Unlike traditional methods, stream mining algorithms can be applied to intrusion detection with several advantages, such as: (*i*) working with limited memory, which allows the implementation in small devices (commonly employed in edge services); (*ii*) processing traffic data with a single read; (*iii*) producing real-time response; and (*iv*) detecting novelty and changes in concepts already learned.

Given the successful[1] use of Novelty Detection in network data streams, this paper shows the results of adapting these mechanisms for use in IoT environments. First, we instantiated and experimentally validated the IDSA-IoT architecture [?] on a distributed system composed of small devices with limited resources on the edge of the network. We evaluated through experimental method, how the distribution of the chosen novelty detection algorithm affects the capability to detect changes (novelty) in traffic patterns, and the 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 ?? presents the related works. Section ?? reviews the chosen novelty detection algorithm MINAS. A distributed extension of the novelty detection algorithm, including its implementation and

[1] citation needed

evaluation are presented in Section **??**. Finally, Section **??** summarizes the main findings and presents possible future work.

2 Related Work Nao mexer por enquanto

Recent works explored those areas, to name a few: BigFlow [**?**] employing Apache Kafka and Apache Flink for distributed stream processing evaluating with package stream dataset, CATRACA [**?**,**?**] uses Apache Kafka and Apache Spark for stream processing and

[**?**]

muito próximo de [**?**] 6LoWPAN is a standard defined by the IETF in RFC 6282, to transmit data with IpV6 and other protocols on low power wireless devices using IEEE 802.15.4 in the lower layers. However, this technology still lacks protection and security mechanisms. For instance, in [**?**], signature detection is used to detect DoS and UDP flood attacks. The architecture uses a probe to promiscuously listen the whole traffic of a 6LoWPAN network and sends the data to analysis on a non-constrained host. The work in [**?**] proposed an hybrid IDS which focuses on specific routing attacks, such as sink-hole and selective-forwarding. Higher complexity tasks which demand more computational resources are executed on the border router, while simpler tasks execute on constrained nodes. Results were expressed by metrics as recall and memory and energy consumption. The work in [**?**] proposed the use of anomaly detection to identify internal routing attacks, and signature detection to identify external attacks. Anomaly detection was tested with simulated attacks, while signature detection used a subset of NSL-KDD. They used the recall and FAR as metrics.

A three-layer architecture (composed of WSN, Fog and Cloud) with focus on fault tolerance in disaster scenarios is proposed in [**?**]. Fog computing is used to execute ML functions and data aggregation. Experiments used real data collected from sensors. The metrics used included precision, recall and accuracy. The work in [**?**] proposed an hybrid IDS which collects information about the environment and activates specific modules to mitigate each kind of attack. Experiments were made in a real environment and metrics used were recall, precision and resource consumption (CPU and RAM). Smart cities scenarios also employ IoT to measuring and monitoring tasks. In [**?**], the authors propose an architecture that uses three stream mining methods based on ML to characterize water and energy consumption behavior, predict consumption, and detect incidents. The metrics used to express results include water and energy consumption. The work in [**?**] also aimed to identify anomalies in a water distribution network and proposes a three layer architecture (sensors, base stations and datacenter). The second layer performs time-sensitive tasks, thus reducing latency, while third layer provides storage and aggregates the results of the second layer with historical data to generate more accurate information. Water distribution measures were used, comparing the values of the predictions with the actual measurements. Intrusion detection for smart cities, based on data mining techniques running on an unrestricted devices is proposed in [**?**]. Experiments using KDD99 data are presented

and the metrics used were precision, Kappa, memory consumption, time, FAR, and FNR.

Table **??** summarizes the discussion on the related work. Note that some works use data from KDD99 or derived from this dataset. Collected two decades ago, KDD is no longer representative of current attack patterns and IoT environments. Some works used traces captured from local infrastructure, which provide realistic evaluation, but lack of reproducibility. Some works use data produced by intentional attacks simulated, designed by the same people who designed the detection techniques. This can bring unrealistic advantages to the detection methods. Also, it is worth noting that most articles used metrics like FAR, recall, and accuracy. Although widely adopted in classical scenarios, such metrics are inaccurate for stream processing [**?**].

Table 1: Summary of related works

Work	Platforms	Technique	DataSet	Metrics
[**?**]	6LoWPAN	Suricata	Real data with metasploit	Accuracy, packets/second
[**?**]	6LoWPAN	Hybrid - MR OPF	NSL-KDD and simulated attacks	FAR and recall
[**?**]	6LoWPAN	DAG analysis	No information	Recall, energy and memory consumption
[**?**]	WSN	MLP of Weka	Real data	MAE, RMSE, R^2, R, accuracy, recall, precision, specificity
[**?**]	WSN	Independent modules, each with one technique	Trace replay and attack injection	Recall, accuracy, memory and CPU consumption
[**?**]	Smart City	Clustering, MLP and statistical models	Real data from meters	Water and energy consumption
[**?**]	Smart City	LiSA, smoothing function	Real data	Outliers, comparison between simulation and collected data
[**?**]	Smart Grid	7 MOA classifiers	KDD99	Accuracy, Kappa, memory consumption, time, FAR and FNR

3 MINAS

MINAS is an offline-online clustering algorithm, which means it has two distinct phases. The first phase generates the model by creating several micro-clusters based on a separate training set that is processed offline. Each micro-cluster can be associated with only one class of the problem, but each class can have many micro-clusters. The online phase is where the model performs three tasks in (near) real-time. We describe the online phase in more detail since we focus on its tasks.

In summary, MINAS executes the classification, novelty detection, and model update tasks in the online phase.

MINAS tries to classify each incoming unlabeled instance according to the current decision model. Instances unexplained by the model receive an *unknown*

label and are stored in temporary memory for future analysis. When this temporary memory reaches a parameterizable size, MINAS groups the instances to form new micro-clusters. Each micro-cluster is validated to discard the non-cohesive or unrepresentative ones. Valid micro-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 micro-clusters and starts using them to classify new instances. MINAS also has a mechanism to forget micro-clusters that became obsolete and unrepresentative of the current data stream distribution. Besides, MINAS also cleans the temporary memory to eliminate ungrouped unknown instances as they represent noise [**?**].

Authors in [**?**] validated the usage of the MINAS algorithm for the Intrusion Detection and deploying it on edge. However, they used the sequential version of the algorithm, where classification stops completely to execute novelty detection.

Formally, a data stream S is a massive sequence of data elements $x_1, x_2, \ldots, x_n$ that is, $S = \{x_i\}_{i=1}^{n}$, which is potentially unbounded (n $\rightarrow \infty$). Compared to traditional (batch) data mining, stream processing algorithms have additional requirements. For instance, with a potentially infinite data stream, storing data for late processing is not a choice due to memory constraints. Algorithms need to incrementally process incoming data instances in a single pass while operating under memory and response time constraints. Furthermore, as data streams present transient behavior, prediction models often need to be incremented to adapt to concept drift observed in data.

```
Parameters: cleaningWindow as CW,
noveltyDetectionTrigger as NDT,
minExamplesPerCluster as MEPC,
noveltyFactor as NF
Function NoveltyDetection(p, ModelSet,
  ModelSleepSet, UnkownSet):
    newModelSet = ∅ , ;
    newClusters = clustering (UnkownSet);
    foreach cl in newClusters do
        if sizeOf (cl.sampleSet) < MEPC then
          continue;
        (distance, near) = mearestCluster (cl,
          ModelSet ∪ ModelSleepSet);
        if distance < near.radius × NF then
            cl.label = near.label;
            cl.type = extension;
        else
            cl.label = p.noveltyIndex;
            p.noveltyIndex = p.noveltyIndex
              +1;
            cl.type = novelty;
        UnkownSet = UnkownSet ⊖
          cl.sampleSet;
        newModelSet = newModelSet ⊕ cl;
    return newModelSet
```

```
Input: ModelSet, Sample Stream
Output: Classified Stream as out
Function MinasOnline(p, ModelSet,
  SampleStream):
    UnkownSet = ∅, ModelSleepSet = ∅,
      lastCleanupTime = now ();
    foreach sampe from SampleStream do
        sample.label = unknown;
        (distance, cluster) = mearestCluster
          (sample, ModelSet);
        if distance < cluster.radius then
            sample.label = cluster.label;
            out ← sample;
            continue;
        UnkownSet = UnkownSet ∪ sample;
        if sizeOf (UnkownSet) ≥ NDT then
            novelties = NoveltyDetection (p,
              ModelSet, ModelSleepSet,
              *UnkownSet);
            ModelSet = ModelSet ⊕ novelties;
        if now () > lastCleanup + CW then
            ModelSet = handleModelSleep
              (ModelSet, ModelSleepSet);
            UnkownSet = removeOldSamples
              (UnkownSet, lastCleanup);
        out ← sample;
```

Algorithm 1: Our interpretation of MINAS [**?**,**?**,**?**]

4 Proposal

In this work, we investigate the use of the novelty detection techniques and strategies presented by the MINAS algorithm and the IDSA-IoT architecture for IoT. However, given the distributed nature and the typical use of low-end devices in envisioned and currently deployed IoT scenarios, new constraints apply: (i) the classification phase of the algorithm must occur in parallel, at different spots at the same time; (ii) the novelty detection phase, which provides the model evolution, must also occur in parallel, at different spots; (iii) both phases must also operate in parallel, possibly at different places/nodes of the system; (iv) the algorithm complexity must allow it to be processed by modest computer devices.

Thus we propose a NIDS using MINAS [**?**] (a Novelty Detection algorithm) to effectively identify previous and new intrusion threats, implemented over a architecture [**?**] using parallel and distributed techniques leveraging edge and cloud for efficient computing.

NIDS monitor the packet network traffic, aggregate into flow descriptors and analyze to identify any intrusion or misbehavior. However, this problem requires both fast and accurate response: the former 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; the latter is required as to not misidentify harmless with harmful and vice-versa. 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 our NIDS scenario. Even though we also capture data produced in the edge, sending this data to the cloud would in the worst case double the internet communication requirements of the overall system. Fog computing infrastructure aims to offload computing resources from cloud providers by placing edge devices closer to end-users and/or data sources. But two MINAS steps limit this fog offload, the processing intensive novelty detection and, long term model storage and distribution of the internal model. Those steps surpass the capabilities of common fog hardware and therefore need to be at least shared to a cloud where such requirements are easy and cheap to fulfill.

In our proposed NIDS, fog and cloud computing resources are employed as to minimize the time elapsed between a flow descriptor ingestion and intrusion alarm, allocating the classification step of MINAS in a MPI cluster 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, and those steps require more resources and thus are divided in fog and cloud.

To have a better overview of our proposal and how it integrate in existing IoT environments, Figure **??** 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 router to the internet or be more complex by connecting to private clouds or containing more devices providing fog computing capabilities; lastly, available over the internet, traditional public cloud

```
Input: mpiParams as mp, params as p,
       ModelSet, Sample Stream
Output: Classified Stream as out
Function Mfog(mp, p, ModelStream,
  InputStream):
    ModelSet = ∅;
    ModelSetLock = new Lock ();
    if mp.rank == 0 then root
        new Thread (Detector, [mp, p,
          ModelSet]);
        Sampler (mp, p, InputStream);
    else leaf
        new Thread (modelReceiver, [mp, p,
          ModelSet]);
        Classifier (mp, p);
Function Classifier(mp, p):
    while True do
        sampe = receive(SampleType, root);
        if sample == EndOfStream then
          break;
        sample.label = unknown;
        with readLock(ModelSetLock)
            (distance, cluster) =
              mearestCluster(sample,
              ModelSet);
        if distance < cluster.radius then
            sample.label = cluster.label;
        send(root, SampleType, sample);
Function modelReceiver(mp, p, ModelSet):
    while True do
        cl = receive(ClusterType, root);
        if cl == EndOfStream then break;
        with writeLock(ModelSetLock)
            ModelSet = ModelSet ⊕ cl;
Function Sampler(mp, p, InputStream):
    destination = 1;
    foreach sample from InputStream do
        if typeOf(sample) is Cluster then
            broadcast(ClusterType, sample,
              root);
            with writeLock(ModelSetLock)
                ModelSet = ModelSet ⊕
                  sample;
            continue;
        send(destination, SampleType,
          sample);
        destination = destination +1;
        if destination > mp.size then
          destination = 1;
Function Detector(mp, p, ModelSet):
    while True do
        sampe = receive(SampleType, any);
        if sample == EndOfStream then
          break;
        out ← sample;
        if sample.label == unknown then
            UnkownSet = UnkownSet ∪
              sample;
            if sizeOf(UnkownSet) ≥
              p.noveltyDetectionTrigger then
                novelties =
                  NoveltyDetection(p,
                  *ModelSet, ∅, *UnkownSet);
                with
                  writeLock(ModelSetLock)
                    ModelSet = ModelSet ⊕
                      novelties;
                foreach cl in novelties do
                    broadcast(ClusterType,
                      cl, root);
```

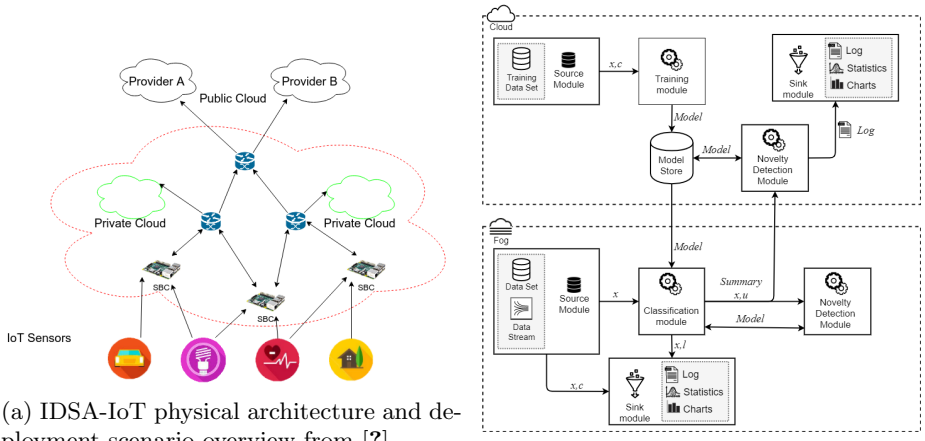
Algorithm 2: MINAS implemented over MPI.

provides inexpensive computing and storage on demand. In this scenario the further apart resources are, more network resources need to be dedicated and, as any networked system, more latency between action and reaction is observed.

While our proposal focuses on fog computing resources, those resources are often limited and they do not have the same reach and availability as traditional public cloud. For that reason we also leverage the public cloud for model storage and distribution, global novelty detection and alarm forwarding.

In Figure **??** we depict the overall architecture cut down to individual modules: three main functional modules being Training, Classification and Novelty Detection handling MINAS main tasks; and three auxiliary Source, Sink and Model Store, addressing external and internal connections as well as providing facilities to our tests and experiments.

Source Module provides the input data stream for Offline and Online phases of MINAS and is deployed in cloud and in fog for each respective phase and a variant for testing on our experiments. Model Store is another trivial module handling the initial Model storage and distribution for Classification and Novelty Detection modules. Last of the auxiliary modules is the Sink module, it denotes the consumer of labeling output stream such as an alarm system, however in our



(a) IDSA-IoT physical architecture and deployment scenario overview from [**?**].

(b) *MFOG* components and communications overview,

Fig. 1: Architecture overview.

implementation it is much more complex, handling all tests metrics extraction and evaluations for our experiments, aggregating all output (stream and logs) in files for proper analysis and later comparisons. This module also differs in its software stack from the C language and MPI library, for the ease of implementing such analysis needed by our experiments, courtesy of Pandas and NumPy libraries, we employed Python for this module.

Training Module encapsulates the Offline phase of MINAS and its output, being the initial model, is stored by the Model Store. Classification Module houses the homonymous task of MINAS Online phase and is the focal point for parallelism in our proposal, being replicated in the fog on each local network containing a cluster with one or more nodes and each node multiple processes (limited to the individual CPU core count). Novelty Detection Module can be also replicated, one instance per local network and one global instance, also handling the homonymous task of MINAS Online phase. This modules takes as input all samples labeled with *unknown*, stores them in a internal *unknown* buffer and when this buffer is full, triggers MINAS Novelty Detection task (clustering followed by validation).

Each communication in Figure **??** shows the direction of the data flow and identifies the data contained: $Model$ is MINAS internal Model containing a set of cluster data structures, x, c identifies a sample with the real class, x is the sample without the real class, x, l identifies a sample with the assigned label, x, u is sample with the *unknown* label, *summary* is a statistical summary of model usage.

4.1 Polices

The distribution of steps and tasks in various modules opens data distribution and its impacts to discussion. The decisions following these discussions can be organized in several policies, some of them are:

- Regarding the allocation of the Novelty Detection Module:
  - It can be located at each fog node meaning novelties will be only detected if sufficient patterns occurs in the local observed network, it also spends the local node processing power and a model sharing mechanism must be added;
  - It can be located in the cloud and thus detect patters even when their footprint is small in each local network, also a model sharing mechanism is not needed as the model has a single instance stored in the cloud, the penalty of this choice is increased internet usage as any sample with *unknown* label must be sent from edge to cloud, implying some delay between the appearance of a novel pattern, its detection and its propagation to fog classifiers;
  - It can be located in both, meaning that a local *unknown* buffer is maintained, novelty detection is performed on that buffer, and once a sample is to be discarded as noise or outlier it must be sent to the cloud where the process repeats but with global data. This choice also need the model sharing mechanism and is clearly the more complex.
- Regarding the model cleanup (forget mechanism): Even when a global novelty detection is used, local models can be optimized for faster classification using the its local model statistics, sorting last or removing clusters that are not in frequent use;
- Lastly, a feature not explicitly shown in the original MINAS is the reclassification of *unknowns* after the detection of a novelty pattern with the newfound label: As the last step in the novelty detection step in MINAS, the *unknown* sample buffer is classified using the newfound subset of clusters, if the sample can be explained by a new cluster it is removed from the *unknown* sample buffer, however, this new labeling is not put forward to the systems output restraining the system data-stream behavior to a *map* (meaning each input has one output), whereas if this feature was enabled the behavior would be a *flatMap* (each input can have many outputs) and introduce new outputs, more recent and perhaps more accurate but later.

4.2 Implementation

The original MINAS algorithm has a companion implementation (*Ref*) written in Java using MOA library base algorithms such as K-means and CluStream, however in the new implementation only K-means is used. Another difference between *Ref* and *MFOG* is cluster radius calculation from the distances of elements forming the cluster and the cluster's center, where the former uses the maximum distance, the latter uses the standard deviation of all distances as described in [**?**].

The stream format for input and output also of note. Input information needed is the samples value ($\vec{v}$), this $\vec{v}$ is a number sequence of length d (dimension). In addition to the $\vec{v}$ for evaluation and training purposes the class identifier as single character, optionally an unique item identifier (*uid*) can be provided. For output information and format the decision isn't so clear as we can't predict future system integrations needs like only novelty alarms or every samples original $\vec{v}$ with assigned label so, we have a compromise and put only enough information for the Sink Module (where the full information from the testing file or stream can be accessed) meaning the format can be defined as a tuple containing *uid* and assigned label.

Another implementation decision related to the output stream is whether or not to reprocess, and add to the output stream, examples in the unknown buffer after the novelty detection procedure, meaning one item can be classified once as unknown and again with a label. Our preliminary tests using this technique had increased true positives when compared to not using it. However this changes the stream operator behavior from a *Map* to a *FlatMap* having duplicate entries on the output stream as previously mentioned. Regardless of choice the classification of the unknown buffer after a model update, using the full model or just the added set of clusters, is done to remove the examples "consumed" in the creation of a new cluster in the internals of the clustering algorithm.

For *MFOG* the Message Passing Interface (MPI, from *Open MPI 4.0.4*) library was used. In MPI programming, multiple processes of the same program are created by the run-time and each process instance receives a rank parameter, for *MFOG* this parameters indicate if the process is root, rank 0, or leaf otherwise. Beyond this division, each process also operates two threads, for the root there is a sampler and detector threads, for the leafs each has a model receiver thread and multiple classifier threads. The overall sequence of interactions is shown in Figure **??**.

The Sink Module was also build following reference techniques like multi-class confusion matrix with label-class association [**?**] to extract classification quality metrics.

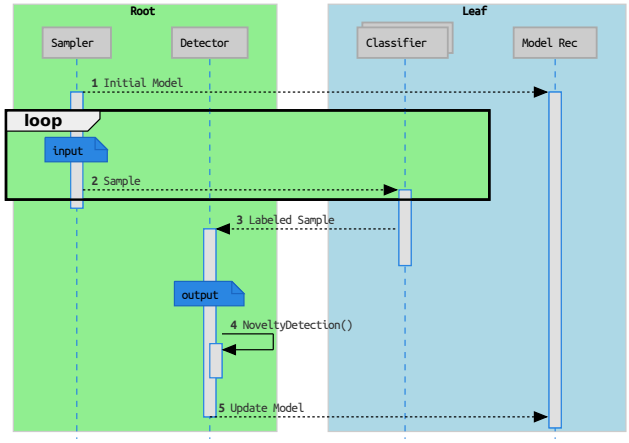


Fig. 2: *MFOG* life line overview.

5 Experiments and Results

For the experimental setup we dedicated three Raspberry Pi 3 model B single board computers connected via Gigabit Ethernet Switch forming a simple cluster. This cluster stored all source code, binaries (compiled and linked in place) and datasets, being accessed via our laboratory network over Secure Shell (SSH). All experiments were executed in this cluster for isolation of otherwise unforeseen variations.

The dataset used is the December 2015 segment of Kyoto 2006+ Dataset[2] (Traffic Data from Kyoto University's Honeypots) [**?**]. This segment was filtered (from 7 865 245 instances) to only examples associated to known attack types identified by existing IDS, and attack types with more than 10 000 instances for significance as done by [**?**]. The remaining instantes then were transformed by normalization, transforming each feature value space (e.g. IP Address, Duration, Service) to the Real interval $[0, 1]$. The result is stored in two sets, training set and test set, using the holdout technique filtering in only normal class resulting in 72 000 instances for training set and 653 457 for test set, containing 206 278 N (normal) class and 447 179 A (attack) class.

5.1 Metrics and Visualizations

There are two broad evaluation metrics for each experiment: a time mesure extracted by using *GNU Time 1.9* and, a set of qualitative mesures extracted by a python program. The first metric is straightforward and is the time measure of the full program execution. The latter metric is not as simple and for its extraction required a purposely build python program. This program takes

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

two inputs, the test dataset and the captured output stream, and outputs the confusion matrix, label-class association, final quality summary with: Hits (accuracy), Misses (Err), Unknowns (UnkR); and stream visualization chart with per example instance summary with novelty label markers.

For clarity, it is necessary to detail how to interpret and compare each metric, as for some it is trivial but others are not so straightforward.

In the confusion matrix $M = m_{ij} \in \mathbb{N}^{c \times l}$, computed by our evaluation program, each row denotes one of the datasets original (actual) class and each column denotes the marked (predicted) label present in the captured output stream. Thus, each cell $M_{c,l}$ contains the count of examples from the test dataset of class c found in the output stream with the label l assigned by the under evaluation experiment. For the dataset under use, original classes are *"N"* and *"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 C are the rows *Assigned* and *Hits*. The former 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 [**?**] section 4.1. The latter row, *Hits*, shows the true positive count for each label, computed by coping the value of the cell $M_{c,l}$ where the label is the same and the class c is the value in the above *Assigned* row. The *Hits* row is also used to compute the overall accuracy. The complete matrix is shown in Tab. **??**.

(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

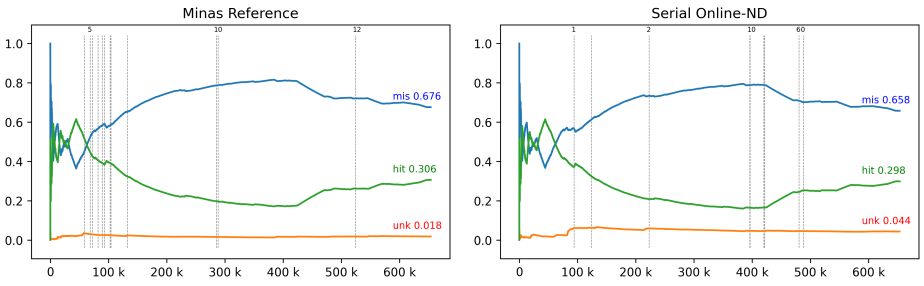
(b) Serial implementation

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

Table 2: Confusion Matrix and Qualitative Metrics

For the metric summary table, six metrics from two sources are displayed. Three metrics *Hits Unknowns Misses* represented as ratio of the captured output stream, extracted from the evaluation python program, computed as follows: *Hits* (overall accuracy) is the summation of the homograph 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. Lastly, *Time*, *System* and *Elapsed* metrics represented in seconds, are extracted from *GNU Time*. *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 (indicates how long another system or person had to wait for the result). To compare the time metric is simple, the lower time taken, the better. Our four main experiments are shown in Tab. **??**.

Lastly, the stream visualization chart shows the summary quality metrics (*Hits Unknowns Misses*) computed for each example in the captured output stream. This summary is computed for each example but it uses the *Assigned* row computed previously to evaluate *Hits*, other metrics are derived as described before. Therefore, horizontal axis (x, domain) plots the index of the example and the vertical axis (y, image) shows the metric computed until that example index on the captured output stream. Adding to the summary metrics, 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 as if showing every label would turn this feature unreadable due to overlapping. Figure **??** shows two complete stream visualization charts.



(a) Reference Implementation (b) Serial Implementation

Fig. 3: Validation Comparison: Stream hits and novelties visualization

5.2 Results Discussion

Four main experiments need detailed discussion: (*a*) reference implementation of Minas (*Ref*) [**?**]; (*b*) new implementation in serial mode; (*c*) new implementation in single-node, multi-task mode and (*d*) new implementation in multi-node, multi-task mode. Each experiment uses the adequate binary executable, initial model (or training set for the reference implementation) 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 **??**.

Table 3: Collected Metrics Summary. deixo os tempos com ou sem a soma do tempo offline?

	Exp. (a)	Offline	Exp. (b)	Exp. (c)	Exp. (d)
Hits	0.305618		0.298438	0.312416	0.312478
Misses	0.676049		0.657843	0.664061	0.663802
Unknowns	0.018333		0.043717	0.023521	0.023718
Time	2761.83	194.12	274.91	716.22	401.26
System	7.15	0.075	11.585	47.845	157.685
Elapsed	2772.07	194.27	287.3	339.31	289.65

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

As stated, to validate our implementation we compare it to *Ref* (the original MINAS companion implementation), so we extracted the same metrics using same process for both *a* and *b*, they can be viewed on Tables **??**, **??** and for ease of comparison on Table **??** the summary can be compared side by side.

In general the observed classification quality metrics are very similar, they 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 small changes to MINAS parameters, MINAS internals like K-means ordering, cluster edge inclusion and cluster radius formula as stated in Subsection **??**. As for the efficiency metrics on Table **??** our implementation used less time to analyze the test data set, this is due to small optimizations on the minimal sample to cluster center on the classifier task and more importantly on the stop condition on the internal K-means algorithm, while *Ref* uses a fixed iteration limit of 100, our implementations adds the "no improvment" check and thus stops earlier on most cases and this in turn reduces time taken on the Novelty Detection step. Also note that *Ref* time includes the Offline phase while our implementation runs it once and reuses the initial model for *b*, *c* and *d*.

As for the effects of running a MPI cluster with our implementation we observe an increase of time when e go from ~~we have to say it is pretty shitty because of our choice of distribution using round-robin, use some load balancing~~

~~and micro-batching for better results.~~ Nevertheless, we can also show the effects of delay in the Classify, Novelty Detect, Model Update and Classify loop, as in the first non-serial experiment b we observe a reduction in Novelty Labels on the Confusion Matrix (Tables **??** and **??**) from 10 to 4. The same effect is observed on the stream visualization as well, where Figure **??**

When observing the stream visualization on figure **??**

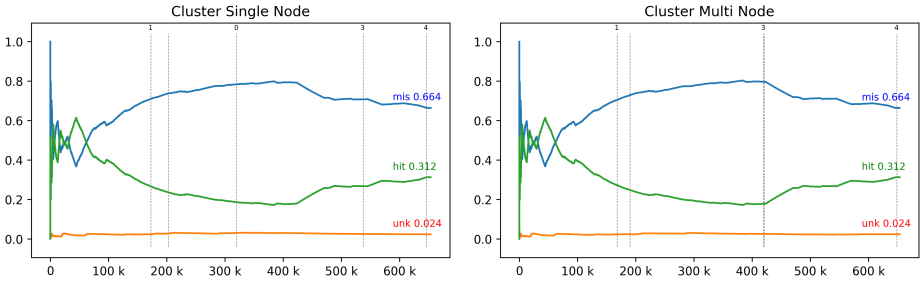
(a) Parallel single-node

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

(b) 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

Table 4: Confusion Matrix and Qualitative Metrics for MPI Clusters.



(a) Parallel single-node (b) Parallel multi-node

Fig. 4: Parallelism Comparison: Stream hits and novelties visualization

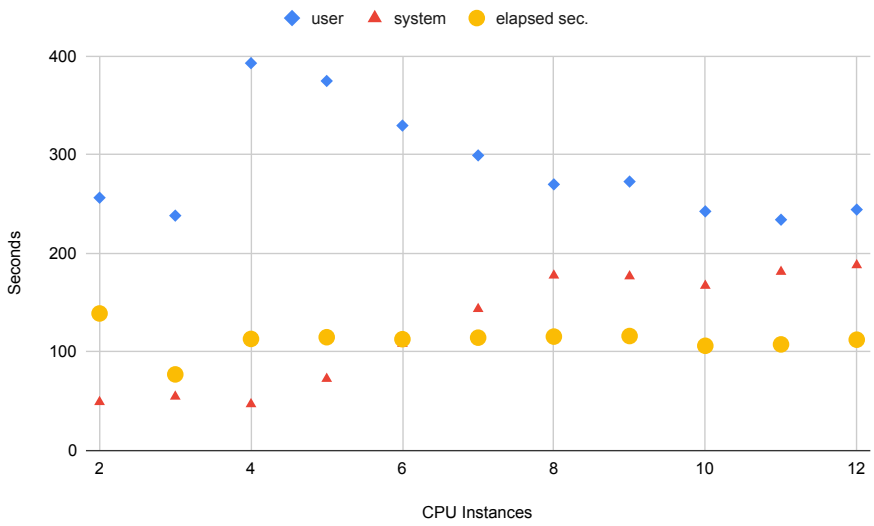


Fig. 5: Time measurements per added instance

6 Conclusion Nao mexer por enquanto

[3] While there is some impact on the predictive results this is not reflected on classification quality metrics indicating that distribution of Novelty Detection algorithms incurs some penalty but is overshadow by the benefits of scalability. Beside that, in our experiments, a months worth of traffic flow descriptors is processed in around 300 seconds even with non-optimal load sharing strategy (single sample round-robin that incurs the bigger networking overhead), sparing use of memory (unknown buffer limited to a fraction of the available memory), and potential system slowdown due to cascading effects on networking queue (caused by the Detector Module when its unknown buffer is full and novelty detection task is under way as it still a non-stream oriented algorithm limited to a single batch of fixed memory).

Aplicação em cenário real.

Acknowledgment

The authors thank CNPq (Contract 167345/2018-4). Hermes Senger also thanks CNPq (Contract 305032/2015-1) and FAPESP (Contract 2018/00452-2, and Contract 2015/24461-2) for their support.

[3] pode ser a conclusão do abstract