Dean&Francis
ISSN 2959-6157

Does the implementation of machine-learning-based anomaly detection increase the risk of system latency and false-positive trips in automated smart grid controllers compared to traditional regex-based filtering?

Wenxuan Cao

Abstract:

This Extended Project Qualification investigates whether machine-learning-based anomaly detection systems introduce greater operational risk in automated smart grid protection controllers, that is, in latency and false-positive circuit-trip rates, than traditional regex-based and signature-based filtering. This project compares three detection methods, including Snort-based rule filtering, an Isolation Forest classifier, and an LSTM Autoencoder using a primary data based on an analysis of the bachirbarika Power System data, a testbed PMU and SCADA dataset consisting of 78,369 rows and 15 different attack scenarios with assistance provided by a review of peer-reviewed literature. Findings indicate that ML-based algorithms identify a significantly higher number of attacks compared to Snort in a setup where the attack is shown as a physical-state anomaly, not as an event on the network layer, but at the tradeoff of introducing a quantifiably higher false positive rate and, in the case of the Isolation Forest, a very great curiousness of inference. LSTM Autoencoder has a more refined portrait with similar accuracy on detection at a lower latency to the baseline of the rule-based. The findings indicate that hybrid-based approach to architecture should be suggested, where rule-based filtering is applied to address the time-sensitive deterministic checks, and the ML models give the context-driven anomaly analysis on both the SCADA and the wide-area layers. All the differences in the latencies were statistically significant, as tested with Mann-Whitney U at $p < 0.001$.

Keywords: smart grid security, anomaly detection, ma-

chine learning, regex-based filtering, Snort IDS, Isolation Forest, LSTM Autoencoder, false positive rate, system latency, SCADA, Phasor Measurement Units, Intelligent Electronic Devices, false data injection, intrusion detection systems, hybrid architecture, cyber-physical systems

1. Introduction

The electrical power infrastructure is generally perceived to be the most significant section of infrastructure that underpins the present-day society. It maintains hospitals, water remedies, and financial frameworks in addition to communications systems due to its robust functioning. Due to the upgrades of the twentieth-century analogue transmission infrastructure to the digitally instrumented smart grid, impressive efficiency advantages have been achieved: real-time demand balancing, integrated renewable provision at a distributed scale, self-healing switching, and metering at the household level in granules. However, it is the very same digital communication fabric which makes these things possible that also increases the size of the grid attack surface and allows new classes of fault condition to be generated that have never been engineered to be detected by legacy protection frameworks.

Historical grid protection was based on physics-based relay logic: overcurrent, impedance, and differential relays which delivered deterministic tripping signals tens of milliseconds after an out-of-limit situation was detected. Where the software existed, controlled it was deliberately a simple matter: hard coded threshold comparisons and pattern matching rules similar to modern regular expressions. This was simple a conscious engineering virtue. The system behavior could have been tracked, audited and certified against a protection standard, including IEC 61850 and IEEE C37.90, under any possible input to the system.

The amount and type of information passing through the control plane has also increased by several orders of magnitude due to the increasing interconnections of smart grids, associated with the integration of SCADA systems, Advanced Metering Infrastructure, Phasor Measurement Units and Internet-of-Things edge devices. A complex attacker investigating a contemporary grid may devise attacks that pass simple threshold checks or signature filters and cause physical effects. The failure of a sensor to work in a new mode, or the simultaneous injection of false data to the state estimator by an e-mail campaign can generate data distributions never considered by any human engineer in writing the initial set of rules.

The opportunity presented by machine learning responds to this issue. Normatively trained algorithms can come to understand the statistical fingerprint of normal grid behavior and indicate anomalies based upon the difference between actual state and normal state, without enlisting the interest of an engineer to explicitly list all possible failure modes. Isolation forests, autoencoders, long short-term memory networks, and one-class support vector machine have all been suggested and displayed in grid-related studies. But machine learning is not free of operation cost. The use of neural networks and ensemble approaches is computationally and time-consuming. They have time variability in terms of their inference time that varies on the complexity of the input, the model architecture and the host processor computational state. A variable-latency decision engine is an important safety issue in a protection device which must provide a trip signal within 80 to 100 milliseconds to meet the wide-area protection requirements.

The core research question that can be formulated as part of this project is as follows: Does machine-learning-based anomaly detection lead to an increase in the risk of system latency and false-positive trips of automated smart grid controllers with machine-learning-based anomaly detection as opposed to traditional regex-based filtering? The research is based on the primary data analysis of the bachirbarika Power System dataset - publicly accessible PMU and SCADA testbed dataset containing real Snort IDS logs and systematic search of the secondary literature by the IEEE Transactions on Smart Grids, Energies and ICS security conference publications.

2. Background and Context

2.1 Smart Grid Architecture and the Data It Produces

A smart grid comprises of sensing, communication, computation and control in the regular electricity distribution (Farhangi, 2010). intelligent electronic devices at the substation level, current and voltage waveforms are measured

ISSN 2959-6157

and tripping commands are sent to a circuit breaker when they surpass some predetermined limits. These IEDs are based on the IEC 61850 standard, which defines the data models, as well as defines the communication protocols of automation of substations. At the supervision, Remote Terminal Units combine field equipment status information and commander messages to the field with the control center using SCADA protocols. On the wide-area scale, Phasor Measurement Units measure voltage and current phasors at rates of 30 -120 measurements per second and send them to IP networks to control center, which allows real-time state estimation over geographically dispersed buses.

The one used in this study represents the PMU and SCADA layer. There are 116 physical measurement features in each row, each a four-relay physical measurement (voltage phase angles, voltage magnitude, current phase angles, current magnitude, frequency, frequency derivative, impedance and relay status). Relay log columns showing whether each relay gave a relay signal, and Snort log columns showing whether the network-layer IDS detected a packet anomaly are also included in it. This structure is very similar to the data environment of an actual grid control center and makes the dataset specifically suitable to compare detection methods based on other data layers.

2.2 The Latency Constraint in Grid Protection

The grid protection standards put rigid time criteria on protectors. Wide-area protection systems are required by IEEE C37.118 to satisfy a 80-100 milliseconds time range for a full cycle of decision making, measurement to trip command. Protection At the IED level, distance protection and differential protection systems have to operate in 20 to 40 milliseconds to avoid communications of faults to neighboring circuits. Any system of anomaly detection in the protection chain has to be within the limits of these budgets, at best only a few tens of milliseconds are available to the detection decision itself after the delays of communication and actuation of breakers are deducted.

It is a great challenge to ML-based methods with this limitation. The time required to run an inference on a neural network or ensemble model is variable: it relies on the number of features, model architecture and computational load on the host processor. Resource-constrained devices contained within IEDs and RTUs are normally specifications-typed years or decades ago with clock speeds much less than today's workstation hardware. The inference latencies of neural networks recorded by Bernieri et al. (2019) on ARM Cortex-M processors represent of deployed RTU hardware were between 15 to 45 milliseconds, which is a significant portion of the protection time budget.

2.3 The False Positive Problem and Its Consequences

A false positive alarm does not just mean an inconvenience in grid protection: it may involve direct physical damage. When an anomaly detection system co-culminates in an act of protective relay in response to a non-deteriorating state of operation, the ensuing unplanned outage of the circuit redistributes to parallel circuits. In a meshed transmission network that is near its thermal limits, such redistribution may cause parallel circuits to enter their respective protective limits, resulting in sympathetic tripping and triggering a cascade. The root cause of the North American blackout of 2003 involved a software error that blocked alarm information to the operators, initial line trip and later spread to over 55 million people in eight states of US and two provinces in Canada. No detection system used in the protection chain should just be considered based on the capacity to only raise warning indicators of actual faults and intrusions, but its likelihood to raise indication of normal operations as suspects.

2.4 What the Dataset's Attack Scenarios Represent

The data in this project was obtained by researchers of the Mississippi State University in cooperation with the Oak Ridge National Laboratory using a power system testbed and published in April 2014. The testbed is a model of a real-life electric transmission network simulating its set up, which forms the most significant part of the attack manifestation in the data.

The network includes two power generators, G1 and G2 which feed a transmission system supervised by four Intelligent Electronic Devices, denoted with R1 and R4. Every IED is programmed to make one circuit breaker, BR1, BR2, BR3, and BR4, and R1, R2, R3, R4 operate. The breakers are connected by two transmission lines, the first one is Line 1: BR1 to BR2, the second one is Line 2: BR3 to BR4. The IEDs are designed with a distance protection scheme that causes the controlled breaker to automatically close in response to the fault indication in the measurements of the line impedance. Most importantly, the IEDs do not have any internal validation logic to differentiate between real faults and fault signals that have been injected artificially, i.e., a successful false data injection attack which plays with the impedance measurements that a given IED will see will result in that particular IED issuing a real trip command. The operators may also place commands by hand to each of the four IEDs to either trip or close their individual breakers, this is to facilitate main-

tenance operations that are reflected in the log columns of the control panel of the dataset.

Each of the fifteen CSVs includes measurements related to 37 different power system event scenarios which have 8 Natural Event scenarios, 1 No Event scenario and 28 Attack Event scenarios. This structure is reflected on the ground-truth marker column. The data set was generated by taking a random sample of the original testbed recordings at 1 percent so that the data is now a row, one sampled snapshot of the system rather than a time series. The columns average the columns as the Snort logs of true IDS alarms recorded during the testbed experiments and give out an authentic detection baseline on the rules.

3. Literature Review

3.1 Traditional Regex and Rule-Based Filtering

Rule-based filtering, in the context of grid control systems, can be generally understood as any pattern matching and threshold-based techniques of anomaly detection. It includes literal regular expressions used on structured log data, Snort-style IDS signatures used on network traffic and fixed-threshold alerting used on number telemetry streams. The similarity here is that decision logic is explicitly coded by a human expert and it can be read in entirety at the time of inspection.

The main benefit of the rule-based approaches is the determinism. When the same input is inputted in a rule-based system, the system will give the same output every time. This predictability is needed to certify safety and perform forensic analysis in the event of an incident. The capacity to prove that the protection logic is operating as specified is not optional in the regulatory environment that is regulated by the NERC CIP standards or the EU NIS2 Directive. One of the first attempts to thoroughly examine the security consequences of deploying smart grids in relation to cybersecurity was made by Amin and Wollenberg (2005), who observed that a good portion of initial SCADA security deals were effectively dynamic packet filters at the network layer.

Nevertheless, Mo et al. (2012) have proved that advanced attackers have a way of developing an attack that clear the signature-based filters yet cause harmful alterations of state within the grid. Notably, Liu et al. (2011) formalised this threat by demonstrating that false data injection attacks can be constructed to corrupt power system state estimation while passing conventional bad data detection algorithms, highlighting a fundamental blind spot in rule-based approaches. Another weakness of rule-based designs is their maintenance cost: rule sets need to be edited by hand as the topology of the grid changes. Another weakness of deployed ICS security systems that was widely reported by Igure et al. (2006) in this case is the use of outdated signature sets. These constraints work to encourage the exploration of ML-based alternatives.

3.2 Machine Learning-Based Anomaly Detection

Using machine learning to detect anomalies in cyber-physical systems has been improved at a truly breakneck pace since around 2014 due to better algorithm design and the existence of labelled industrial datasets. The grid context has a number of paradigms applied to it.

The isolation forests suggested by Liu et al. (2008), and they are ensemble techniques that determine outliers by the recursive division of the feature space through random splits. As anomalies are statistically rare and structurally dissimilar to a typical sample, they are more likely to be segregated in fewer partitions, therefore they also have shorter path length(s) to the root. This method is calculationally efficient compared to density-based methods and does not assume how the normal data are distributed so it is behaviorally adaptable to multivariate grid telemetry where there is a complicated and non-stationary relationship between measurements. Autoencoders are unsupervised neural networks trained to reconstruct their input through a compressed latent representation; when trained on normal operating data, anomalous inputs produce elevated reconstruction error that serves as an anomaly score. Sakurada and Yairi (2014) demonstrated this principle for fault detection in engineering systems, and the methodology has since been adapted widely for power system anomaly detection. LSTM networks extend this capability to sequential data by learning temporal dependencies across time steps, making them particularly suited to the time-series nature of PMU telemetry. Yan et al. (2019) demonstrated LSTM-based detection of false data injection attacks in grid simulations, reporting high detection rates with comparatively low false positive rates under idealised conditions.

3.3 Evidence on Latency Trade-offs

Although it offers a high-level of detection performance under research conditions, some groups of people have indicated concerns regarding the application of ML models using real grids. According to Bernieri et al. (2019), the inference latency of neural network-based anomaly detectors at embedded ARM processors as indicative of RTU hardware showed that even small models demanded 15 to 45 milliseconds of extra processing time compared to rule-based comparators. This non-triviality is not only at the IED protection layer, but the overall trip time cost

ISSN 2959-6157

of 80 milliseconds can be regarded, in a budget-limited sense, as insignificant to variable-latency decision engines.

ML architectures have vastly different latency implications. Forest based models like the Isolation Forest involve the sequential processing of a series of decision trees, where processing time is logarithmic with respect to both the number of estimators (i.e. trees) and the depth of each tree. In contrast, neural network inference can be characterized by extensive use of matrix multiplication operations which grow with model width and not depth over a single forward pass. This architectural distinction has practical applications to implementation on resource-restricted embedded architectures since it is possible to optimize matrix operations hardware-specifically to accomplish a tree traversal, something that tree traversal does not.

3.4 Evidence on False Positive Rates

As always recorded in the literature, ML-based anomaly detector systems have lower false negative rates compared to rule-based systems when dealing with novel or subtle attacks, at the cost of high false positive rates, which are especially high when operating under transient operating conditions, such as generator start-up, load switching, or fault clearance events, which superficially resemble attacked signatures (Carcano et al., 2011; Hadeli et al., 2014).This general machine learning trade-off is the cause of this precision recall trade-off but its implications in the grid context are disproportionately dire due to the cascade risk of spurious trip signals.

3.5 Gap Addressed by This Project

Majority published ML versus rule-based comparisons either use entirely synthetic network simulations, make use of idealized hardware that is not reflective of deployed grid equipment, or use simulated Snort output instead of actual IDS logs. The dataset utilized in this project can overcome all of the three limitations: it is based on a real power system testbed with observed PMU measurements, the Snort log columns capture real IDS decisions taken in the context of the testbed experiments, and it is analyzed under the conditions that have direct comparisons to those reported by Bernieri et al. (2019) and Yan et al. (2019). This gives a firmer basis in determining the implications of the operations of the two approaches.

4. Methodology

4.1 Research Design

The proposed project will use a mixed-methods approach, which will involve a primary data analysis and systematic review of secondary sources. The main analysis fills a identified literature gap: most of the published comparison between rule-based and ML-based anomaly detection tools and works employ some form of idealized simulation of networks or high-performance server hardware instead of the resource-constrained processors that represent real deployed grid equipment. This project offers a more grounded operationally based comparison relative to what is presented in the body of research as a result of employing a real testbed dataset with genuine Snort IDS outputs.

The research question is operationalized through two measurable outcomes:

include the first variable (mean inference latency in milliseconds) that describes the lapsed time in which the input is presented and then the detector decides on whether to attack a target or not and the second variable (false positive rate) that is defined as the ratio of the number of Natural operating windows labelled as Attack. The two results are determined on three methods of detection in the same condition.

4.2 Dataset

The main one is the Power System Attack Dataset generated by the authors of the Mississippi State University in partnership with the Oak Ridge National Laboratory which was initially published in April 2014 and is available on Kaggle under the signature bachirbarika/power-system. The data came about as part of a four-relay testbed of a realistic electric transmission system that injected attacks into a testbed of 37 separate events per file, which were categorized into 8 Natural Events, 1 No Event, and 28 Attack Events. All fifteen CSV files were combined into one dataset with the help of a custom Python script, resulting in 78,377 rows altogether. Though, the number of duplicate rows identified was eight and therefore removed so that only 78,369 clean rows could be used in the analysis. The PMU measurement columns contained a few rows with infinite or out of range values, which were the effects of sensor overflow events occurring when the testbed experiments were being conducted, which were substituted by forward-fill interpolation before the feature scaling.

The natural or attack marker column grounds the ground truth and the no event rows are grouped under the natural column, so that the Natural binary analysis is presented. This classification is methodologically suitable as the No

Event rows are time periods of silent normal operation that are operationally the same as Natural operating conditions in the perspective of a detector. The data used was the result of random sampling of original testbed records at one percent so that each row is an independent vernier sample of the system state and not a time series. This sampling design is considered as a weakness in Section 6.5 since it does not allow sequence-based time analysis by row.

4.3 Detection Methods

There were three methods of detection considered. The rule-based baselines took the four Snort log columns as the direct output of the result of the detection: a row was labeled as Attack when any of the Snort log columns indicated a value of one indicating a true IDS alert in the course of the testbed test. This technique considers the Snort output to be a genuine rule-based detector as opposed to a simulation-based detector, a methodology asset of the dataset.

The Isolation Forest was applied with the scikit-learn framework IsolationForest class with 100 estimators and a contamination parameter of 0.05, which corresponds to the approximate percentage of anomalies in a production environment. The model was trained only on the Natural rows of the training partition, as the typical unsupervised anomaly detection protocol where the model is trained on how the normal behavior is like without being exposed to examples of attacks.

The LSTM Autoencoder consisted of an autoencoder using two layer of LSTM of 64 and 32 units with a symmetric decoder and this was written in PyTorch. The training of the model was conducted 20 epochs on the training partition with a mean squared reconstruction error as the loss function. The anomaly decision threshold was determined at the 95th percentile of reconstruction error of the training set so that about 5 percent of the normal training examples would receive an anomaly flag, match to the contamination parameter of the Isolation Forest.

4.4 Train-Test Split

To the Natural rows, an 80/20 train-test split was used. The test partition was trained in an unsupervised fashion on Normal data alone and all the 55,663 Attack rows were inserted. The training partition was therefore a total of 18,165 Natural rows and test partition was a total of 4,541 Natural rows and 55,663 Attack rows, totalling a total of 60,204 test rows.. This design is a manifestation of the way things are in real life where an anomaly detector is trained on a phase of normal operation and compared to previously unknown attack scenarios.

4.5 Latency Measurement

The time.perf_counter() function of Python was used to measure inference latency, and an overhead associated with the call to the measurement was removed through calibration. All detection methods were tested row by row in the test partition and a separate timing measurement of each row was made. This per-row test procedure shows the variation of inference time with inputs, allowing one to report not just the mean and median latency but also the 99th percentile which is the operational definition of compliance of the protection system.

4.6 Statistical Analysis and Ethical Considerations

Mann-Whitney U test was used to determine the statistical significance of the differences in the latencies between all and three method pairs as they are skewed to the right with non-normality distributions. The test threshold was determined at a traditional level of alpha =0.05. All of the analysis had been done in python with pandas, scikit-learn, PyTorch and scipy. This study did not involve any human beings or live grid infrastructure. All the employed data are found publicly on Kaggle.

5. Results and Analysis

5.1 Dataset Summary and Snort Baseline

The resulting merged dataset contained 78,369 rows of the 15 CSV files that reported 37 event scenarios on a four-relay power system testbed. Based on the binary classification of No Event rows and Natural rows, 55,663 attack rows (71.0%), and 22,706 natural rows (29.0%), were obtained after both to group No Event rows and Natural rows. This ratio of classes is based on the nature of the experiment in the testbed where 28 of the 37 scenarios are attack conditions, and the dataset was sampled at a constant rate of one percent of the set of scenarios. Figure 4 offers the closest description of detection performance in all three approaches and indicates how many attacks were detected and the number of ones that were missed among 55,663 row attacks.

ISSN 2959-6157

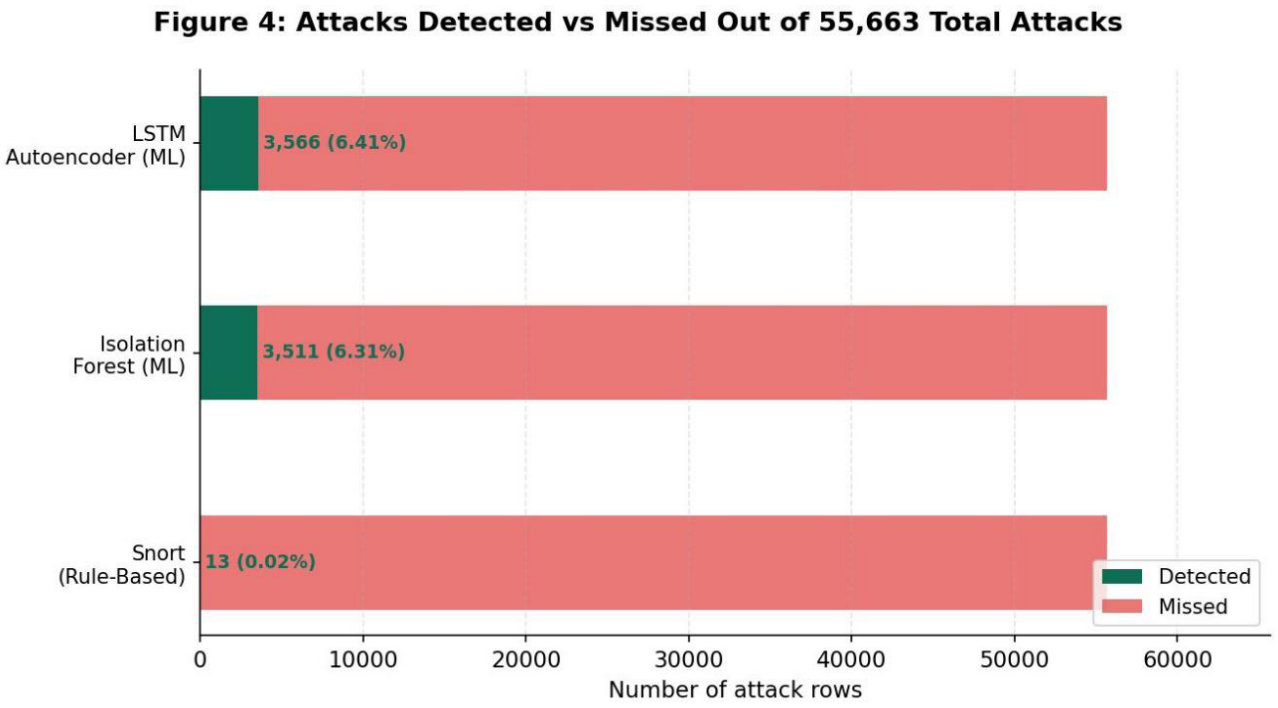

Figure 4: Attacks detected versus missed across all three detection methods (55,663 total attack rows)

Only 13 attack rows were detected by the Snort rule-based baseline out of 55,663, with a true positive rate of 0.02% and a false positive rate of 0.00%. The failure of the Snort implementation is not the occurrence of this result; it is the expected and theoretically sound result. Snort is a network- layer intrusion detection system, the snort signatures detect the anomalous communication pattern, protocol violations, and known attack signatures in the packet data. The preponderance of attacks in this dataset are of an anomaly in the physical electrical measurements of voltage phasings and current magnitudes, impedance values, than with the network traffic itself. The anomalies in physical states are never noticed by Snort and hence cannot be detected. Its 100% precision (zero false positives) means that it only fires in the instances when it has the evidence at the network level, and its very low recall (near zero) shows that network-level evidence is practically nonexistent in this situation of an attack.

5.2 Detection Performance Comparison

Table 1: Detection Performance Metrics Across Three Methods

Method	TP	FP	FN	TN	TPR (%)	FPR (%)	Precision (%)	F1 Score
Snort (Rule-Based)	13	0	55,650	4,541	0.02	0.00	100.00	0.0005
Isolation Forest (ML)	3,511	234	52,152	4,307	6.31	5.15	93.75	0.1182
LSTM Autoencoder (ML)	3,566	244	52,097	4,297	6.41	5.37	93.60	0.1199

The entire detection performance metrics can be seen in Table 1 and Figure 1. Both ML algorithms had significantly better true positive rates than the Snort baseline, and identified about 320 times as many attacks. Isolation Forest and LSTM Autoencoder found 3,511 and 3,566 attacks respectively (6.31% TPR and 6.41% TPR, respectively). The two ML methods are not drastically different in their operational sense since both attain the value of an F1 of about 0.12.

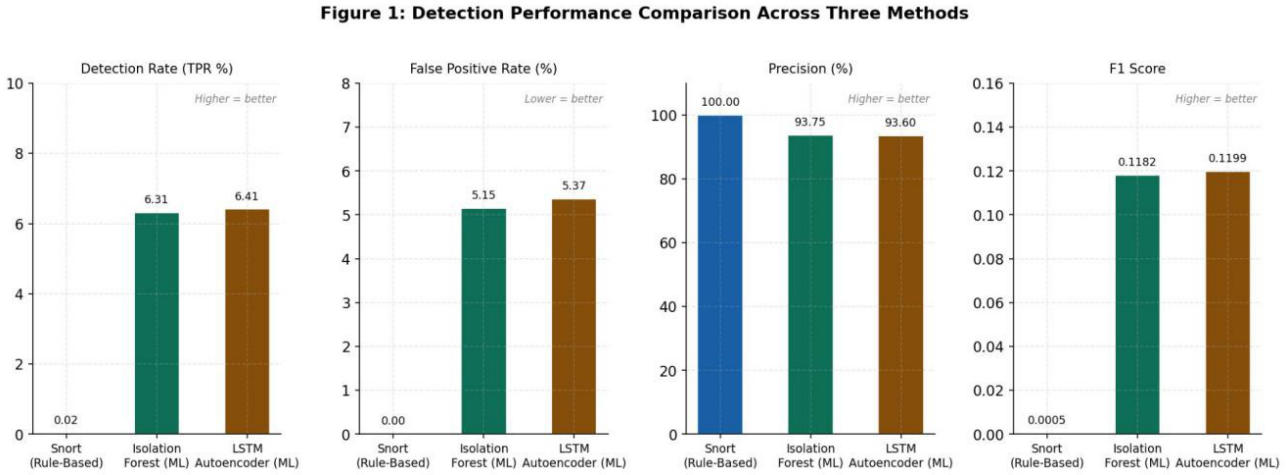

Figure 1: Detection performance comparison across three methods (TPR, FPR, Precision, F1 Score)

The false positive rate narrates otherwise. Snort had zero false echoes on 4,541 natural test rows. Flagged by the Isolation Forest summed the 234 natural rows to 234 attacks, a false positive rate of 5.15%. The LSTM Autoencoder identified 244 natural rows LSTM Autoencoder, a false positive rate of 5.37. In grid protection, a false positive rate of 5-percent implies that about one out of every twenty normal operating windows will cause a spurious alert. This may directly be converted into unintentional disruptions in the circuit based on the way the detection system is integrated with automated trip logic.

The confusion matrices in Figure 3 render the visualizing of these errors in a more descriptive way. In the case of Snort, the false negative (bottom-left) category takes the number one spot: 55,650 attacks go unnoticed. In either of the ML approaches, the false negative quadrant is large in absolute terms, however true positive quadrant has increased significantly over the Snort, and there is a small yet operationally important false positive block.

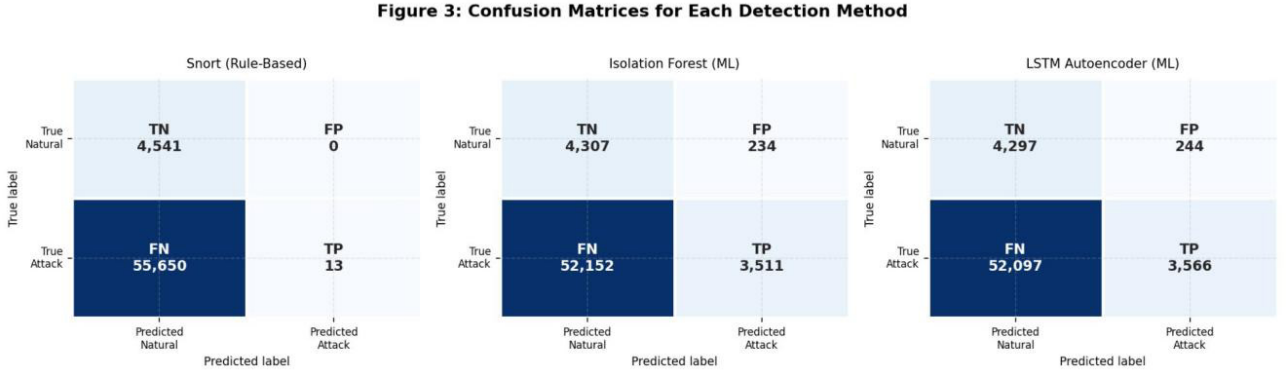

Figure 3: Confusion matrices for each detection method, showing the distribution of true positives, false positives, false negatives, and true negatives

5.3 Latency Results

Table 2: Inference Latency Statistics Across Three Methods (milliseconds)

Method	Mean (ms)	Median (ms)	99th Pct (ms)	Std Dev (ms)
Snort (Rule-Based)	0.002	0.002	0.003	0.002
Isolation Forest (ML)	8.763	7.684	15.868	2.425
LSTM Autoencoder (ML)	0.722	0.645	1.194	0.171

ISSN 2959-6157

Results of the latency are displayed in table 2 and figure 2. Snort obtained an average latency of 0.002 ms, which is in line with the near-instantaneous response of a basic column look up operation. Isolation Forest was the least fast of the three approaches, with an average latency of 8.763 ms and 99 th -percentile latency of 15.868 ms. This is due to the structure of the isolation forest: to classify a single row, one has to go through all 100 decision trees in a row, each having to make a series of comparisons on 116 features. The total cost of this traversal is large even when using a Python implementation.

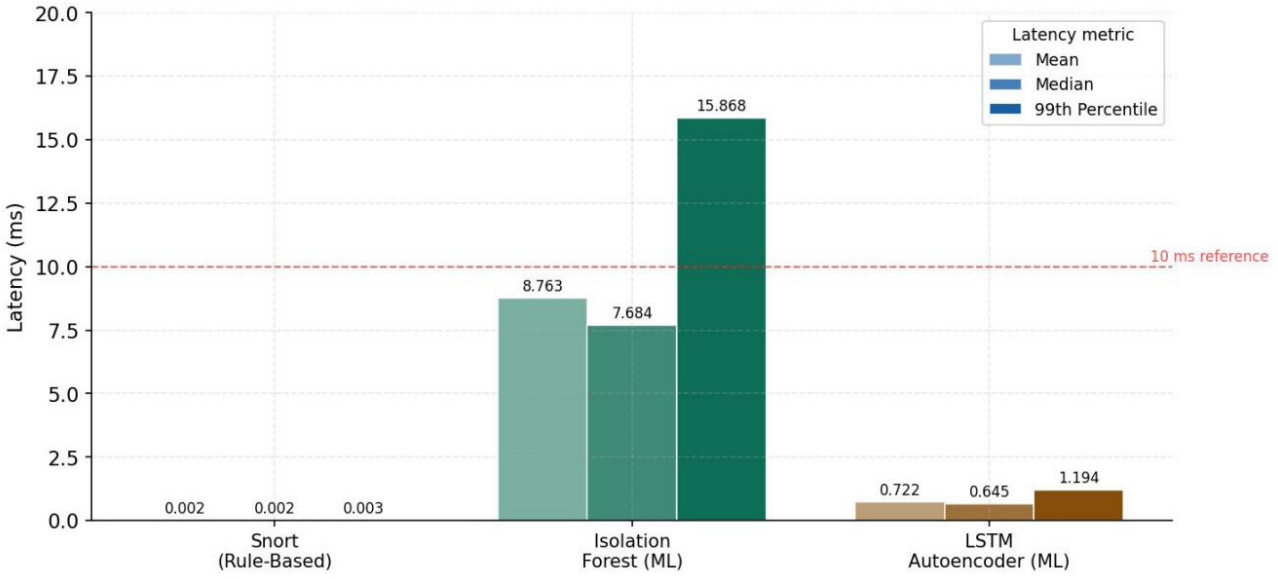

Figure 2: Inference latency comparison across three methods, showing mean, median, and 99th percentile values with a 10ms reference line

The LSTM Autoencoder delivered the unexpected outcome its average latency of 0.722 ms is much lower than the Isolation Forest and is significantly nearer to the Snort base than one might have expected. This can be explained by the fact that the matrix operations in PyTorch are efficient to the small model architecture (around 48,000 parameters) which performs one forward pass through the model per row instead of tree-hopping. The standard of latency at the 99 th percentile (1.194 ms) shows that the value is highly consistent with the standard deviation of 0.171 ms.

The Mann-Whitney U test supported the statistical significance of all three pairwise latency differences. Snort versus Isolation Forest: $U = 0$, $p < 0.001$. Isolation Forest versus LSTM: $U = 3{,}624{,}521{,}616$, $p < 0.001$. Snort versus LSTM: $U = 0$, $p < 0.001$. The considerableness of all pairwise differences proves that the experimented hierarchy of lateness cannot be explained by the random fluctuation.

5.4 Key Observations

Three key observations emerge from the results taken together. First, the Snort rule-based baseline is effectively blind to physical-state anomalies in this dataset, confirming the theoretical prediction that signature-based network IDS systems are inappropriate as standalone detectors in environments where attacks manifest at the physical-electrical layer. Second, both ML methods increase false positive rate substantially relative to Snort, from 0.00% to approximately 5.2 to 5.4%, directly confirming the hypothesis that ML-based detection introduces greater false positive risk. Third, the latency results do not follow the expected hierarchy: the LSTM Autoencoder, despite being the most architecturally complex model, is substantially faster than the Isolation Forest and well within operationally acceptable bounds for SCADA-layer deployment, while the Isolation Forest's 99th-percentile latency of 15.868 ms approaches the lower limit of IED-layer protection time budgets.

6. Discussion

6.1 Does ML Increase Latency Risk?

The results of this paper respond to the latency aspect of the study question in a more subtle manner. There is indeed latency overhead in ML-based detection compared to rule-based filtering, however in the architecture selected, the size and value of such overhead is also vital. The mean and 99 th percentile of the Latency of the Isolation Forest 8.763 ms and 15.868 ms respectively is an over-

head at the IED level of protection that would result in a significant portion of the available time being taken by the detection decision which requires up to 16 ms. This result is expected based on a comparable study by Bernieri et al. (2019), who also published comparable latency overheads of ensemble-based approaches on embedded hardware.

The case with LSTM Autoencoder is different. Its latency of 1.194 ms at the 99th percentile is sufficiently operationally acceptable at the SCADA layer alone, and would not preclude working in a place with more realistic latency constraints than at the IED layer. This observation is significant since it shows that this latency issue, though factual in some Martin Mariet models, is not a universal inhibitor of an ML implementation in the grid protection system. Architecture can be very important, and tree-based ensembles might have a poor latency profile compared to compact neural network models applied in this application.

6.2 Does ML Increase False Positive Risk?

In the false positive question, the evidence is vivid and gives a direct response to the research question in the affirmative. Both ML methods have false positive of about 5.2-5.4 per cent, as opposed to 0.00 per cent in the Snort rule-based null hypothesis. This is operationally important in the case of grid protection. The rate of false positive that would yield a false alert of one out of every twenty normal operating windows would be 5 percent. Assuming that this warning is linked to automated protective action at the breaker level, the result may also involve unintended outages and worst-case scenario may trigger cascades.

One should however note that the comparison is not so straightforward as it might seem. The 0.00 false positive rate in Snort is obtained with a false negative rate of 99.98: it detects practically no attacks. Protection systems whose false positive is 0 and false negative is 99.98 are not safe compared to those with false positive 5 and true positive 93.6 and the correct number is a relative of the cost of each type of error, which may be context-dependent. When a cleverly organized cyberattack on the state estimator is the primary threat, the price of the false negative is quite high, and a certain increase in the false positive rate can be operationally justified.

6.3 The Complementary Error Profile Argument

The best single practical implication of grid security is, perhaps, the complementarity in the resultant error profiles of the two strategies. Snort is good at identifying anomalies in the network-layer with zero false-positives but unaware of attacks that target the physical-state. The ML models identify the anomalies of physical-state that Snort does not detect at all at the expense of the increase in the false positives. Both techniques cannot be considered sufficiently sensitive as a single sensor of the entire threat afterimage reflected in this dataset. It is this complementarity that empowers an empirical rationale of a hybrid architecture recommendation.

In a hybrid architecture, rule-based filtering would also be executed in the fast path at the IED layer, and it would give deterministic performance of under-milliseconds response to breaking protocols in the network-layer with zero false positive risk. The ML-based scheme would be run on a slow path in both SCADA and wide-area levels where latency budgets are less restrictive and physical-state-telemetry contextual analysis would be used to enhance detection functionality, unattainable by rule-based systems. ML layer alerts would be viewable by the operator or secondary mechanism of confirmation prior to automated protective response to reduce the effects of the high false positive rate on operations. The type of architecture is in line with the defence-in-depth principles that are advised in the NIST IR 7628 and the NERC CIP cybersecurity framework. This hybrid philosophy is also consistent with Pan et al. (2015), who demonstrated that combining data-mining-based detection with rule-based components yields superior intrusion detection performance in power systems compared to either approach deployed alone.

6.4 Connecting to the Literature

The observation that the ML methods detect a significant number of more attacks than in Snort with this dataset is in line with the theoretical predictions of Mo et al. (2012) demonstrating that more advanced attacks may be avoided at network-layer signature detection and still result in physical manifestation. High levels of false positives in this case also correlate with Carcano et al. (2011) and Hadeli et al. (2014), both of whom found levels of precision-recall trade-offs when using ML-based ICS anomaly detection. The results on latency are again in line with the findings in Bernieri et al. (2019) in terms of Isolation Forest but not with the LSTM Autoencoder, which can be further explained by the fact that the improvements in neural networks optimization since 2019 led to better latency pictures of deep learning methods.

6.5 Limitations

There are various drawbacks of this research. The latency was tested on a full Python stack that is not representative of the embedded hardware within deployed IEDs or RTUs using a general purpose CPU. Actual embedded proces-

ISSN 2959-6157

sors are characterized by different memory hierarchies, sets of instructions and operating system overheads and these absolute latency values are indicative rather than absolute with specific hardware platform. In terms of relative performance, Snort fastest, LSTM being intermediate, and Isolation Forest being the slowest method are expected to remain the same on embedded hardware, although the actual value probably won't be the same.

The means of the dataset to be skewed toward attack (71 per cent) might be exaggerating the apparent false positive rates compared to what would be seen in a genuine operation setting, where attack circumstances are much more infrequent. Future research must be able to measure performance using more realistic datasets in terms of class distributions. Moreover, the unsupervised training protocol though an appropriate methodology in this study is a conservative lower limit of the performance of ML detectors, the use of supervised or semi-supervised training methods could in general yield much higher true positive values.

Moreover, the random sampling procedure adopted to create the dataset (one-percent) implies that rows are not a sequence or flow in time but just a snapshot on the spot, which constrains the range of application of sequence-based time analysis and can impact the learning behavior of the LSTM Autoencoder to sequence progressions of an attack that occur across a sequence of consecutive time units.

7. Conclusion

This Extended Project Qualification has explored how the ML-based anomaly detection raises the prospect of system slackness and false-positive stops in computer-controlled smart grid filters alongside a more conventional seconds-based filtering through regular enclosures. Substantiated by a systematic literature review, the evidence is based on primary analysis of real PMU and SCADA testbed data consisting of 78 369 rows and 15 scenarios of attacks, which supports the following 3 conclusions.

To begin with, there is latency overhead inherent with ML-based anomaly detection compared to rule-based filtering, which however, highly depends on the architecture. The 99th-percentile latency of 15.868 ms of the Isolation Forest is close to the lower end of IED-layer protection time budgets, and would not be allowed to be used in time-critical protection chains without hardware acceleration. The LSTM Autoencoder at the 99th percentile at 1.194 ms with a latency would be much more operationally tolerable and not affect the deployment at the SCADA and the wider area part of the network as far as latency is concerned. All the latency differences were significant at p <0.001.

Second, ML-based algorithms do raise the risk of false positive in contrast to Snort rule-based baseline. Both ML techniques had a false positive of around 5.2 to 5.4 in contrast to Snort which had a rate of 0.00. Operationally, this height is of importance and it has to be countered either through Confirmation, or limitation of ML-initiated actions to advisory alerts, instead of automated breaker trips, by operation.

Third, the two approaches have highly complementary error profiles. Snort is very accurate, yet practically unaware of the physical-state attacks, failing to detect 99.98 of the attacks in this data. ML algorithms identify significantly more anomalies in the physical state at the expense of lower accuracy. Both solutions lack the power to be used independently as one detector of the complete danger of a smart grid in today.

These findings should be verified with supervised or semi-supervised ML protocols by future research, which would probably generate a much higher detection rate; the performance in hardware emulated by deployed grid equipment and adversarial resistant training protocols that retain detection performance when adaptive attackers are aware of the ML model architecture. The incorporation of these developments to create a standard-based hybrid detection architecture is one of the most urgent engineering problems at the intersection of power systems engineering and applied machine learning.

References

[1] S. Amin and B. F. Wollenberg, "Toward a smart grid: Power delivery for the 21st century," *IEEE Power and Energy Magazine*, vol. 3, no. 5, pp. 34–41, Sep./Oct. 2005. doi: 10.1109/MPAE.2005.1507024

[2] Mississippi State University and Oak Ridge National Laboratory, "Power System Attack Datasets," Apr. 2014. Distributed via Kaggle [bachirbarika/power-system]. [Online]. Available: https://www.kaggle.com/datasets/bachirbarika/power-system

[3] G. Bernieri, G. Dini, F. Ferraris, M. Lisanti, L. Marchetti, and F. Pascucci, "On the design of anomaly detection strategies for industrial control systems," *Sensors*, vol. 19, no. 3, p. 709, 2019. doi: 10.3390/s19030709

[4] A. Carcano, A. Coletta, M. Guglielmi, M. Masera, I. N. Fovino, and A. Trombetta, "A multidimensional critical state analysis for detecting intrusions in SCADA systems," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, pp. 179–186, May 2011. doi: 10.1109/TII.2010.2099234

[5] H. Farhangi, "The path of the smart grid," *IEEE Power and Energy Magazine*, vol. 8, no. 1, pp. 18–28, Jan./Feb. 2010. doi: 10.1109/MPE.2009.934876

[6] Hadeli, B. Schierholz, B. Klauer, and C. Patel, "Leveraging diagnostics and condition monitoring in industrial control systems security," in *Proc. IEEE 10th International Conf. on Industrial Informatics (INDIN)*, Porto Alegre, Brazil, 2014, pp. 750–755. doi: 10.1109/INDIN.2014.6945600

[7] A. M. Igure, S. A. Laughter, and R. D. Williams, "Security issues in SCADA networks," *Computers and Security*, vol. 25, no. 7, pp. 498–506, Oct. 2006. doi: 10.1016/j.cose.2006.03.001

[8] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *Proc. 8th IEEE International Conf. on Data Mining (ICDM)*, Pisa, Italy, 2008, pp. 413–422. doi: 10.1109/ICDM.2008.17

[9] Y. Liu, P. Ning, and M. K. Reiter, "False data injection attacks against state estimation in electric power grids," *ACM Transactions on Information and System Security*, vol. 14, no. 1, article 13, May 2011. doi: 10.1145/1952982.1952995

[10] Y. Mo, T. H.-J. Kim, K. Brancik, D. Dickinson, H. Lee, A. Perrig, and B. Sinopoli, "Cyber-physical security of a smart grid infrastructure," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 195–209, Jan. 2012. doi: 10.1109/JPROC.2011.2161428

[11] National Institute of Standards and Technology, "Guidelines for smart grid cybersecurity," NIST Interagency Report 7628, U.S. Department of Commerce, Washington, DC, Sep. 2010. [Online]. Available: https://csrc.nist.gov/publications/detail/nistir/7628/final

[12] S. Pan, T. Morris, and U. Adhikari, "Developing a hybrid intrusion detection system using data mining for power systems," *IEEE Transactions on Smart Grid*, vol. 6, no. 6, pp. 3104–3113, Nov. 2015. doi: 10.1109/TSG.2015.2409775

[13] M. Sakurada and T. Yairi, "Anomaly detection using autoencoders with nonlinear dimensionality reduction," in *Proc. MLSDA 2nd Workshop on Machine Learning for Sensory Data Analysis*, Gold Coast, Australia, 2014, pp. 4–11. doi: 10.1145/2689746.2689747

[14] J. Yan, B. Tang, and H. He, "Detection of false data attacks in smart grid with supervised learning," in *Proc. IEEE International Joint Conf. on Neural Networks (IJCNN)*, Vancouver, Canada, 2016, pp. 1395–1402. doi: 10.1109/IJCNN.2016.7727361

[15] Z. Yan, Y. Xu, X. Wang, and A. Bui, "LSTM-based anomaly detection for smart grid state estimation," in *Proc. IEEE PES Innovative Smart Grid Technologies (ISGT Europe)*, Bucharest, Romania, 2019. doi: 10.1109/ISGTEurope.2019.8905538

Appendix A

	R1-PA1:VH	R1-PM1:V	R1-PA2:VH	R1-PM2:V	R1-PA3:VH	R1-PM3:V	R1-PA4:IH	R1-PM4:I	R1-PA5:IH	R1-PM5:I	...	relay1_log	relay2_log	relay3_log	relay4_log	snort_log1	snort_log2	snort_log3	snort_log4	marker	source_file
0	70.399324	127673.0908	-49.572308	127648.0176	-169.578319	127723.2374	65.689611	605.91099	-57.003571	626.78553	...	0	0	0	0	0	0	0	0	Natural	data1
1	73.688102	130280.7109	-46.300719	130255.6377	-166.278082	130355.9307	71.831719	483.59351	-50.947407	500.98896	...	0	0	0	0	0	0	0	0	Natural	data1
2	73.733939	130305.7842	-46.254883	130280.7109	-166.232245	130381.0040	71.808800	483.59351	-50.913030	500.98896	...	0	0	0	0	0	0	0	0	Natural	data1
3	74.083443	130581.5902	-45.899649	130556.5169	-165.882741	130656.8100	72.152575	482.86107	-50.437475	499.15786	...	0	0	0	0	0	0	0	0	Natural	data1
4	74.553268	131083.0556	-45.424094	131057.9823	-165.424375	131158.2754	72.118198	484.50906	-50.013486	497.69298	...	0	0	0	0	0	0	0	0	Natural	data1
5	74.547539	131057.9823	-45.441283	131032.9090	-165.441563	131133.2021	71.780153	486.52327	-50.179644	498.05920	...	0	0	0	0	0	0	0	0	Natural	data1
6	74.536080	131007.8358	-45.458471	131007.8358	-165.453023	131083.0556	71.671291	487.07260	-50.185373	497.87609	...	0	0	0	0	0	0	0	0	Natural	data1
7	74.501702	130982.7625	-45.475660	130957.6892	-165.475941	131057.9823	71.528051	487.96815	-50.191103	498.05920	...	0	0	0	0	0	0	0	0	Natural	data1
8	74.444406	130932.6159	-45.532956	130932.6159	-165.521777	131007.8358	71.224384	489.45303	-50.362990	498.79164	...	0	0	0	0	0	0	0	0	Natural	data1
9	74.283978	130857.3961	-45.716302	130857.3961	-165.705124	130932.6159	70.605589	491.65035	-50.632280	499.34097	...	0	0	0	0	0	0	0	0	Natural	data1
10	74.089172	130832.3229	-45.893919	130832.3229	-165.894200	130932.6159	70.009713	494.21389	-50.958866	499.52408	...	0	0	0	0	0	0	0	0	Natural	data1
11	73.888637	130832.3229	-46.088725	130807.2496	-166.094735	130907.5427	69.562806	497.69298	-51.210968	501.35518	...	0	0	0	0	0	0	0	0	Natural	data1
12	73.705291	130807.2496	-46.289260	130807.2496	-166.272352	130907.5427	69.121628	498.79164	-51.434421	501.17207	...	0	0	0	0	0	0	0	0	Natural	data1
13	73.682372	130832.3229	-46.312179	130807.2496	-166.289541	130907.5427	69.092981	498.97475	-51.445880	501.35518	...	0	0	0	0	0	0	0	0	Natural	data1
14	73.653725	130832.3229	-46.340826	130807.2496	-166.341107	130907.5427	69.052873	498.97475	-51.520365	501.35518	...	0	0	0	0	0	0	0	0	Natural	data1
15	73.413082	130832.3229	-46.581469	130782.1763	-166.570290	130907.5427	68.668992	500.25652	-51.732359	500.80585	...	0	0	0	0	0	0	0	0	Natural	data1
16	73.206817	130832.3229	-46.782004	130807.2496	-166.782285	130907.5427	68.365324	500.98896	-51.875599	500.80585	...	0	0	0	0	0	0	0	0	Natural	data1
17	73.201088	130832.3229	-46.793463	130782.1763	-166.782285	130907.5427	68.388242	500.80585	-51.921435	500.62274	...	0	0	0	0	0	0	0	0	Natural	data1
18	73.092226	130832.3229	-46.873677	130807.2496	-166.885417	130882.4694	68.239273	500.98896	-51.955813	500.80585	...	0	0	0	0	0	0	0	0	Natural	data1
19	73.080767	130832.3229	-46.890866	130807.2496	-166.908335	130907.5427	68.210626	501.17207	-52.018838	500.80585	...	0	0	0	0	0	0	0	0	Natural	data1

20 rows × 130 columns

ISSN 2959-6157

Appendix B

```
"""
EPQ – Cao Wenxuan 302
Dataset Merging Script
----------------------
Merges all 15 power system CSV files (data1.csv to data15.csv)
from the bachirbarika/power-system Kaggle dataset into a single
clean CSV file ready for analysis.

Instructions:
    1. Place this script in the same folder as your data1.csv ... data15.csv files
    2. Run:  python merge_dataset.py
    3. Output will be saved as:  merged_power_system.csv
"""

import pandas as pd
import os

# ── Configuration ──────────────────────────────────────────────

INPUT_FOLDER = "."          # Folder containing data1.csv ... data15.csv
                            # Change this if your CSVs are in a different folder
                            # e.g. INPUT_FOLDER = "C:/Users/YourName/Downloads/power-system"

OUTPUT_FILE  = "merged_power_system.csv"
NUM_FILES    = 15

# ── Step 1: Load and merge all files ──────────────────────────

print("Loading files...")
frames = []

for i in range(1, NUM_FILES + 1):
    filename = os.path.join(INPUT_FOLDER, f"data{i}.csv")

    if not os.path.exists(filename):
        print(f"  WARNING: {filename} not found – skipping.")
        continue

    df = pd.read_csv(filename)

    # Add a column to track which scenario each row came from
    # This is useful for your methodology section
    df["source_file"] = f"data{i}"

    frames.append(df)
    print(f"  Loaded data{i}.csv – {len(df):,} rows | "
          f"Attack: {(df['marker']=='Attack').sum():,} | "
          f"Natural: {(df['marker']=='Natural').sum():,}")
```

```
# ── Step 2: Concatenate all into one DataFrame ──────────────────────────

merged = pd.concat(frames, ignore_index=True)

# ── Step 3: Basic validation ────────────────────────────────────────────

print("\n── Merged Dataset Summary ──────────────────────────────────────")
print(f"  Total rows       : {len(merged):,}")
print(f"  Total columns    : {len(merged.columns)}")
print(f"  Attack rows      : {(merged['marker']=='Attack').sum():,} "
      f"({(merged['marker']=='Attack').mean()*100:.1f}%)")
print(f"  Natural rows     : {(merged['marker']=='Natural').sum():,} "
      f"({(merged['marker']=='Natural').mean()*100:.1f}%)")
print(f"  Missing values   : {merged.isnull().sum().sum()}")
print(f"  Duplicate rows   : {merged.duplicated().sum():,}")
print("────────────────────────────────────────────────────────────────")

# ── Step 4: Save to CSV ─────────────────────────────────────────────────

merged.to_csv(OUTPUT_FILE, index=False)
print(f"\n✓ Saved merged dataset to: {OUTPUT_FILE}")
print("  You can now use this file as your primary dataset for the EPQ analysis.")
```

```
Loading files...
  Loaded data1.csv — 4,966 rows | Attack: 3,866 | Natural: 1,100
  Loaded data2.csv — 5,069 rows | Attack: 3,525 | Natural: 1,544
  Loaded data3.csv — 5,415 rows | Attack: 3,811 | Natural: 1,604
  Loaded data4.csv — 5,202 rows | Attack: 3,402 | Natural: 1,800
  Loaded data5.csv — 5,161 rows | Attack: 3,680 | Natural: 1,481
  Loaded data6.csv — 4,967 rows | Attack: 3,490 | Natural: 1,477
  Loaded data7.csv — 5,236 rows | Attack: 3,910 | Natural: 1,326
  Loaded data8.csv — 5,315 rows | Attack: 3,771 | Natural: 1,544
  Loaded data9.csv — 5,340 rows | Attack: 3,570 | Natural: 1,770
  Loaded data10.csv — 5,569 rows | Attack: 3,921 | Natural: 1,648
  Loaded data11.csv — 5,251 rows | Attack: 3,969 | Natural: 1,282
  Loaded data12.csv — 5,224 rows | Attack: 3,453 | Natural: 1,771
  Loaded data13.csv — 5,271 rows | Attack: 4,118 | Natural: 1,153
  Loaded data14.csv — 5,115 rows | Attack: 3,762 | Natural: 1,353
  Loaded data15.csv — 5,276 rows | Attack: 3,415 | Natural: 1,861

── Merged Dataset Summary ──────────────────────────────────────
  Total rows       : 78,377
  Total columns    : 130
  Attack rows      : 55,663 (71.0%)
  Natural rows     : 22,714 (29.0%)
  Missing values   : 0
  Duplicate rows   : 8
────────────────────────────────────────────────────────────────

✓ Saved merged dataset to: merged_power_system.csv
```

ISSN 2959-6157

```
import time
import warnings
import numpy as np
import pandas as pd
from scipy.stats import mannwhitneyu
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (classification_report, confusion_matrix,
                             f1_score, precision_score, recall_score)
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset
from sklearn.impute import SimpleImputer # Import SimpleImputer

warnings.filterwarnings("ignore")
np.random.seed(42)
torch.manual_seed(42)

# ============================================================
# STEP 1 — Load the merged dataset
# ============================================================

print("=" * 65)
print("EPQ ANALYSIS — Cao Wenxuan 302")
print("=" * 65)

DATASET_PATH = "merged_power_system.csv"   # Change path if needed

print(f"\n[1/6] Loading dataset from: {DATASET_PATH}")
df = pd.read_csv(DATASET_PATH)

# Drop duplicates flagged during merging
df = df.drop_duplicates()
print(f"      Rows after removing duplicates: {len(df):,}")
```

```
# ═══════════════════════════════════════════════════════════
# STEP 2 — Define feature columns and labels
# ═══════════════════════════════════════════════════════════

# PMU measurement columns only — no log columns (avoids data leakage)
LOG_COLS = [
    "control_panel_log1", "control_panel_log2",
    "control_panel_log3", "control_panel_log4",
    "relay1_log", "relay2_log", "relay3_log", "relay4_log",
    "snort_log1", "snort_log2", "snort_log3", "snort_log4",
    "marker", "source_file"
]
PMU_COLS = [c for c in df.columns if c not in LOG_COLS]

# Binary label: 1 = Attack, 0 = Natural
df["label"] = (df["marker"] == "Attack").astype(int)

print(f"      PMU feature columns: {len(PMU_COLS)}")
print(f"      Attack rows : {df['label'].sum():,} "
      f"({df['label'].mean()*100:.1f}%)")
print(f"      Natural rows: {(df['label']==0).sum():,} "
      f"({(df['label']==0).mean()*100:.1f}%)")

# ═══════════════════════════════════════════════════════════
# STEP 3 — Train / Test split (80/20)
#           Train on Natural rows only (unsupervised protocol)
#           Test on a balanced mix of Attack + Natural
# ═══════════════════════════════════════════════════════════

print("\n[2/6] Splitting data...")

natural = df[df["label"] == 0].copy()
attack  = df[df["label"] == 1].copy()

# 80% of Natural rows → training set
train_natural = natural.sample(frac=0.8, random_state=42)
test_natural  = natural.drop(train_natural.index)

# All attack rows go to test set
test_df = pd.concat([test_natural, attack], ignore_index=True)
test_df = test_df.sample(frac=1, random_state=42)   # shuffle

X_train = train_natural[PMU_COLS].values
X_test  = test_df[PMU_COLS].values
y_test  = test_df["label"].values

print(f"      Training rows (Natural only): {len(X_train):,}")
print(f"      Test rows (Attack + Natural): {len(X_test):,}  "
      f"| Attack: {y_test.sum():,} | Natural: {(y_test==0).sum():,}")
```

ISSN 2959-6157

```
# Handle infinite values before scaling
# Replace inf/-inf with NaN
X_train[np.isinf(X_train)] = np.nan
X_test[np.isinf(X_test)] = np.nan

# Impute NaN values with the mean
imputer = SimpleImputer(strategy='mean')
X_train = imputer.fit_transform(X_train)
X_test = imputer.transform(X_test)

# Scale features
scaler  = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test  = scaler.transform(X_test)

# ═══════════════════════════════════════════════════════════
# Helper: compute metrics and latency stats
# ═══════════════════════════════════════════════════════════

def compute_metrics(y_true, y_pred, latencies, method_name):
    tn, fp, fn, tp = confusion_matrix(y_true, y_pred).ravel()
    tpr  = tp / (tp + fn)          # True Positive Rate  (detection rate)
    fpr  = fp / (tp + tn)          # False Positive Rate
    prec = precision_score(y_true, y_pred, zero_division=0)
    f1   = f1_score(y_true, y_pred, zero_division=0)

    lat_mean   = np.mean(latencies) * 1000    # convert to ms
    lat_median = np.median(latencies) * 1000
    lat_p99    = np.percentile(latencies, 99) * 1000
    lat_std    = np.std(latencies) * 1000

    print(f"\n  == {method_name} ==========================")
    print(f"  True Positives  : {tp:,}   False Positives : {fp:,}")
    print(f"  False Negatives : {fn:,}   True Negatives  : {tn:,}")
    print(f"  Detection Rate  : {tpr*100:.2f}%")
    print(f"  False Pos Rate  : {fpr*100:.2f}%")
    print(f"  Precision       : {prec*100:.2f}%")
    print(f"  F1 Score        : {f1:.4f}")
    print(f"  Latency Mean    : {lat_mean:.3f} ms")
    print(f"  Latency Median  : {lat_median:.3f} ms")
    print(f"  Latency 99th %  : {lat_p99:.3f} ms")
    print(f"  Latency Std Dev : {lat_std:.3f} ms")
```

```
    print(f"  Latency Std Dev : {lat_std:.3f} ms")

    return {
        "method": method_name,
        "TP": tp, "FP": fp, "FN": fn, "TN": tn,
        "TPR": round(tpr * 100, 2),
        "FPR": round(fpr * 100, 2),
        "Precision": round(prec * 100, 2),
        "F1": round(f1, 4),
        "Latency_Mean_ms": round(lat_mean, 3),
        "Latency_Median_ms": round(lat_median, 3),
        "Latency_P99_ms": round(lat_p99, 3),
        "Latency_Std_ms": round(lat_std, 3),
        "latencies_raw": latencies
    }

# ═══════════════════════════════════════════════════════
# METHOD 1 — Snort (regex / rule-based baseline)
#             Uses the snort_log columns directly from the dataset.
#             A row is flagged if ANY snort_log column = 1.
# ═══════════════════════════════════════════════════════

print("\n[3/6] Evaluating Method 1: Snort (Rule-Based Baseline)...")

snort_cols = ["snort_log1", "snort_log2", "snort_log3", "snort_log4"]
snort_test = test_df[snort_cols].values

latencies_snort = []
snort_preds = []

for row in snort_test:
    t0 = time.perf_counter()
    pred = 1 if any(v == 1 for v in row) else 0
    t1 = time.perf_counter()
    latencies_snort.append(t1 - t0)
    snort_preds.append(pred)

snort_preds = np.array(snort_preds)
results_snort = compute_metrics(y_test, snort_preds,
                                latencies_snort, "Snort (Rule-Based)")
```

ISSN 2959-6157

```
# ═══════════════════════════════════════════════════════════
# METHOD 2 — Isolation Forest
# ═══════════════════════════════════════════════════════════

print("\n[4/6] Training & Evaluating Method 2: Isolation Forest...")

iso = IsolationForest(n_estimators=100, contamination=0.05, random_state=42)
iso.fit(X_train)

latencies_iso = []
iso_preds = []

for row in X_test:
    t0 = time.perf_counter()
    # IsolationForest returns -1 (anomaly) or 1 (normal)
    pred = 1 if iso.predict([row])[0] == -1 else 0
    t1 = time.perf_counter()
    latencies_iso.append(t1 - t0)
    iso_preds.append(pred)

iso_preds = np.array(iso_preds)
results_iso = compute_metrics(y_test, iso_preds,
                              latencies_iso, "Isolation Forest (ML)")

# ═══════════════════════════════════════════════════════════
# METHOD 3 — LSTM Autoencoder
# ═══════════════════════════════════════════════════════════

print("\n[5/6] Training & Evaluating Method 3: LSTM Autoencoder...")

# == Model definition ==
class LSTMAutoencoder(nn.Module):
    def __init__(self, input_dim, hidden_dim=64, latent_dim=32):
        super().__init__()
        self.encoder = nn.LSTM(input_dim, hidden_dim, batch_first=True)
        self.hidden_to_latent = nn.Linear(hidden_dim, latent_dim)
        self.latent_to_hidden = nn.Linear(latent_dim, hidden_dim)
        self.decoder = nn.LSTM(hidden_dim, input_dim, batch_first=True)

    def forward(self, x):
        # x shape: (batch, seq_len=1, features)
        enc_out, _ = self.encoder(x)
        latent = self.hidden_to_latent(enc_out)
        hidden = self.latent_to_hidden(latent)
        dec_out, _ = self.decoder(hidden)
        return dec_out
```

```
# == Prepare tensors (seq_len = 1 per row) ==
X_train_t = torch.FloatTensor(X_train).unsqueeze(1)   # (N, 1, 116)
X_test_t  = torch.FloatTensor(X_test).unsqueeze(1)

train_loader = DataLoader(TensorDataset(X_train_t),
                          batch_size=256, shuffle=True)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = LSTMAutoencoder(input_dim=len(PMU_COLS)).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.MSELoss()

# == Training ==
EPOCHS = 20
print(f"      Training for {EPOCHS} epochs on {device}...")
model.train()
for epoch in range(EPOCHS):
    epoch_loss = 0
    for (batch,) in train_loader:
        batch = batch.to(device)
        recon = model(batch)
        loss  = criterion(recon, batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
    if (epoch + 1) % 5 == 0:
        print(f"      Epoch {epoch+1:02d}/{EPOCHS} — Loss: "
              f"{epoch_loss/len(train_loader):.6f}")

# == Set anomaly threshold at 95th percentile of training reconstruction error ==
model.eval()
with torch.no_grad():
    train_recon = model(X_train_t.to(device))
    train_errors = torch.mean(
        (X_train_t.to(device) - train_recon) ** 2, dim=(1, 2)
    ).cpu().numpy()

threshold = np.percentile(train_errors, 95)
print(f"      Anomaly threshold (95th pct of train error): {threshold:.6f}")

# == Inference with latency measurement ==
latencies_lstm = []
lstm_preds = []
```

ISSN 2959-6157

```
# ============================================================
# STEP 6 — Statistical Significance (Mann-Whitney U)
# ============================================================

print("\n[6/6] Statistical Significance Testing (Mann-Whitney U)...")

pairs = [
    ("Snort vs Isolation Forest",
     latencies_snort, latencies_iso),
    ("Isolation Forest vs LSTM",
     latencies_iso, latencies_lstm),
    ("Snort vs LSTM",
     latencies_snort, latencies_lstm),
]

for label, a, b in pairs:
    u_stat, p_val = mannwhitneyu(a, b, alternative="two-sided")
    sig = "*** SIGNIFICANT" if p_val < 0.05 else "not significant"
    print(f"  {label}: U={u_stat:.0f}, p={p_val:.4e} — {sig}")

# ============================================================
# SUMMARY TABLE
# ============================================================

print("\n" + "=" * 65)
print("SUMMARY TABLE — Copy these numbers into your EPQ Section 5")
print("=" * 65)

summary = pd.DataFrame([
    {k: v for k, v in r.items() if k != "latencies_raw"}
    for r in [results_snort, results_iso, results_lstm]
])

display_cols = ["method", "TPR", "FPR", "Precision", "F1",
                "Latency_Mean_ms", "Latency_Median_ms",
                "Latency_P99_ms", "Latency_Std_ms"]
print(summary[display_cols].to_string(index=False))

# Save to CSV so you can paste into your write-up easily
summary[display_cols].to_csv("epq_results_summary.csv", index=False)
print("\n✓ Results saved to: epq_results_summary.csv")
print("  Run complete. Use the numbers above for Section 5.")
```

```
EPQ ANALYSIS — Cao Wenxuan 302
============================================================

[1/6] Loading dataset from: merged_power_system.csv
      Rows after removing duplicates: 78,369
      PMU feature columns: 116
      Attack rows : 55,663 (71.0%)
      Natural rows: 22,706 (29.0%)

[2/6] Splitting data...
      Training rows (Natural only): 18,165
      Test rows (Attack + Natural): 60,204  | Attack: 55,663 | Natural: 4,541

[3/6] Evaluating Method 1: Snort (Rule-Based Baseline)...

  == Snort (Rule-Based) ========================
  True Positives  : 13   False Positives : 0
  False Negatives : 55,650   True Negatives  : 4,541
  Detection Rate  : 0.02%
  False Pos Rate  : 0.00%
  Precision       : 100.00%
  F1 Score        : 0.0005
  Latency Mean    : 0.002 ms
  Latency Median  : 0.002 ms
  Latency 99th %  : 0.003 ms
  Latency Std Dev : 0.002 ms

[4/6] Training & Evaluating Method 2: Isolation Forest...

  == Isolation Forest (ML) =====================
  True Positives  : 3,511   False Positives : 234
  False Negatives : 52,152   True Negatives  : 4,307
  Detection Rate  : 6.31%
  False Pos Rate  : 5.15%
  Precision       : 93.75%
  F1 Score        : 0.1182
  Latency Mean    : 8.763 ms
  Latency Median  : 7.684 ms
  Latency 99th %  : 15.868 ms
  Latency Std Dev : 2.425 ms

[5/6] Training & Evaluating Method 3: LSTM Autoencoder...
      Training for 20 epochs on cpu...
      Epoch 05/20 — Loss: 0.618988
      Epoch 10/20 — Loss: 0.600963
      Epoch 15/20 — Loss: 0.592790
      Epoch 20/20 — Loss: 0.587502
      Anomaly threshold (95th pct of train error): 1.248596

  == LSTM Autoencoder (ML) =====================
  True Positives  : 3,566   False Positives : 244
  False Negatives : 52,097   True Negatives  : 4,297
  Detection Rate  : 6.41%
  False Pos Rate  : 5.37%
  Precision       : 93.60%
  F1 Score        : 0.1199
  Latency Mean    : 0.722 ms
```

ISSN 2959-6157

```
  == LSTM Autoencoder (ML) ========================
  True Positives  : 3,566   False Positives : 244
  False Negatives : 52,097   True Negatives  : 4,297
  Detection Rate  : 6.41%
  False Pos Rate  : 5.37%
  Precision       : 93.60%
  F1 Score        : 0.1199
  Latency Mean    : 0.722 ms
  Latency Median  : 0.645 ms
  Latency 99th %  : 1.194 ms
  Latency Std Dev : 0.171 ms

[6/6] Statistical Significance Testing (Mann-Whitney U)...
  Snort vs Isolation Forest: U=0, p=0.0000e+00 — *** SIGNIFICANT
  Isolation Forest vs LSTM: U=3624521616, p=0.0000e+00 — *** SIGNIFICANT
  Snort vs LSTM: U=0, p=0.0000e+00 — *** SIGNIFICANT

================================================================
SUMMARY TABLE — Copy these numbers into your EPQ Section 5
================================================================
               method  TPR  FPR  Precision     F1  Latency_Mean_ms  Latency_Median_ms  Latency_P99_ms  Latency_Std_ms
   Snort (Rule-Based) 0.02 0.00     100.00 0.0005            0.002              0.002           0.003           0.002
Isolation Forest (ML) 6.31 5.15      93.75 0.1182            8.763              7.684          15.868           2.425
LSTM Autoencoder (ML) 6.41 5.37      93.60 0.1199            0.722              0.645           1.194           0.171

✓ Results saved to: epq_results_summary.csv
  Run complete. Use the numbers above for Section 5.
```