JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

Machine Learning Based Log Monitoring and Threat Detection Using Random Forest with a Persistent Alert Dashboard

Ramesh B E, Poral Nagaraj, Pranav H Origanti, Syed Abdullah, Md Anas
Department of Computer Science and Engineering
SJM Institute of Technology, Chitradurga, Karnataka, India
Affiliated to Visvesvaraya Technological University (VTU), Belagavi

Abstract—A security analyst who has to read thousands of log records every day will sooner or later miss something. In this paper we present SentinelAI SOC (Security Operations Center), a prototype that classifies structured network log records as Normal or Attack using a Random Forest classifier and shows the result on a Streamlit dashboard. The dataset used is a KDD Cup 99 style collection with 41 input features. Categorical fields are converted with One-Hot Encoding, and the encoder and the classifier are saved together as one Scikit-learn pipeline so that training and prediction always use the same transformation. Two operating modes are provided, batch analysis of a CSV (Comma Separated Values) file and repeated monitoring of a designated live log file. Every detected attack gets a confidence value and an application-level severity, and it is stored in an SQLite database after a SHA-256 (Secure Hash Algorithm, 256 bit) fingerprint check, so that the same event is not inserted twice. On a held-out test split of 98,805 records the classifier made only 23 errors, giving an accuracy of about 99.98 percent. The paper describes the architecture, the data flow, the implementation and the results, and it also states clearly what the prototype does not do.

Keywords—log monitoring, threat detection, intrusion detection, Random Forest, KDD Cup 99, Streamlit, SQLite, security operations center.

I. INTRODUCTION

Every application, server and network service leaves behind a trail of event records, which we call logs. A single record of a network connection may carry the protocol, the service, the state of the connection, the number of bytes sent and the duration. Looked at one by one, such a record says very little. Looked at together, thousands of them tell us what normal behaviour is and where something has gone wrong.

The difficulty is volume. An organisation can produce millions of records in a day, and a monitoring operator must find the few suspicious ones among them. Traditional rule-based monitoring still has an important place because rules are easy to understand, but rules have to be written and maintained by people, and they may not cover every combination of feature values [1], [2]. A supervised Machine Learning (ML) model offers a second layer. It learns from labelled examples and gives a class for every new record, which helps the analyst to decide what to look at first. It does not replace the analyst.

Many academic works stop at the model. They train a classifier in a notebook, report an accuracy figure and end there. We wanted to go one step further and put the model inside a working application, so that data preparation, prediction, alert handling, storage and display are all connected. This is the reason for building SentinelAI SOC. The contributions of this paper are as follows.

(a) A Random Forest pipeline that keeps One-Hot Encoding and classification together, so that the same transformation is applied in training and in prediction.

(b) A dashboard with two modes, batch analysis of a CSV file and repeated monitoring of a live log file, both using the same prediction path.

(c) An alert module that adds confidence and severity to every attack, stores it in SQLite, and prevents duplicates through a SHA-256 fingerprint.

(d) An honest evaluation on a 98,805 record test split, with the limits of the study stated openly.

The rest of the paper is arranged as follows. Section II reviews the related work, Section III gives the system architecture, Section IV explains the data flow, and Section V describes the implementation. Section VI presents the results, Section VII discusses the limitations, and Section VIII concludes the paper.

II. RELATED WORK

Intrusion detection means watching the events of a computer system or a network and analysing them for signs of unauthorised activity. The National Institute of Standards and Technology (NIST) describes the common classes of intrusion detection and prevention systems and gives guidance on their planning and operation [1]. Detection methods are usually grouped into signature-based and anomaly-based ones. A signature method compares events with known patterns, and an anomaly method flags deviation from learned normal behaviour. Liao et al. review both groups and their variants [3]. The work reported here belongs to a third, related group, where a classifier is trained on labelled examples of both normal and attack records.

Machine learning for intrusion detection has been surveyed widely. Buczak and Guven discuss data mining

and ML methods for cyber security and point out that the quality of the data decides much of the result [4]. Ferrag et al. compare deep learning approaches on several intrusion datasets [5]. At the same time, Sommer and Paxson warned that detection systems based on learning are harder to use in real networks than the laboratory results suggest, because real traffic is varied and the cost of a wrong alert is high [2]. We keep this warning in mind throughout the paper.

The KDD Cup 1999 data was prepared for an intrusion detection competition and describes network connections with 41 features and a label [6]. It is old, and its weaknesses are well known. Tavallaee et al. analysed it and found many redundant records that make results look better than they are [7]. Surveys of intrusion datasets by Ring et al. and by Hindy et al. also discuss how current datasets should be chosen and read [8], [9]. We therefore use the dataset only as a benchmark to demonstrate a complete workflow, and we do not make any claim about live networks.

Random Forest, introduced by Breiman, builds many decision trees on randomised views of the data and combines their votes [10]. The accuracy of the forest depends on the strength of the single trees and on how little they are correlated with one another. Structured log records mix numerical fields such as byte counts with categorical fields such as protocol and service, and tree ensembles handle such a mixture without much tuning. The practical side is supported by the Python tools Pandas, NumPy and Scikit-learn [11]–[13].

III. PROPOSED SYSTEM ARCHITECTURE

The proposed architecture is shown in Fig. 1. Structured records come in from a labelled dataset, a test log file or a live log file. They are prepared by the preprocessing stage and passed to the ML pipeline, which returns a class label. For a record classified as Attack, the threat analysis stage adds the confidence, the severity and the event metadata, and computes a fingerprint. The result goes to two places, the SQLite alert store and the Streamlit dashboard. The responsibilities of data source, preprocessing, inference, threat analysis, storage and presentation are kept separate, so that each part can be changed without disturbing the others.

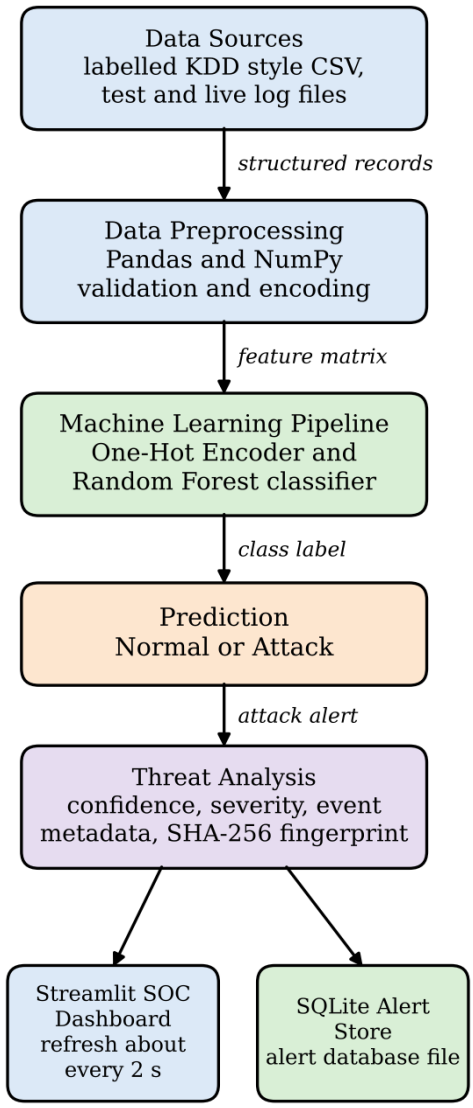

Fig. 1. Architecture of the log monitoring and threat detection system.

A. Data Sources and Preprocessing

The system accepts structured records from two routes. In the first, the user gives a CSV file and all rows are processed. In the second, the application repeatedly reads a designated live log file so that newly appended rows are analysed. The live mode is based on a file. It must not be mistaken for raw packet capture or for direct inspection of network traffic. For demonstration, a bundled stream of test security events can be written into the monitored file.

The preprocessing stage loads the data with Pandas, assigns the column names, separates the features from the target, and converts the original labels into two classes, Normal as 0 and Attack as 1. The three categorical fields, protocol type, service and flag, are encoded with the One-Hot Encoder, which turns every category into a group of indicator columns. The encoder is configured to ignore unknown categories, so that a value not seen in training does not stop the prediction.

B. Machine Learning Pipeline

The encoder and the Random Forest classifier are placed in one Scikit-learn pipeline. This is a deliberate choice. When preprocessing is done by hand outside the model, the columns used in training and the columns used in prediction can quietly differ, and the result is wrong without any error message. With one pipeline, a record goes in as raw feature values and the same fitted transformation is applied before the classifier is called. The fitted pipeline is saved with Pickle and loaded when the application starts, so the model need not be trained again every time the dashboard is opened.

C. Threat Analysis, Storage and Dashboard

When the pipeline predicts Attack, the application builds an alert with an event identifier, the detection time, the severity, the confidence, the protocol, the service, the connection flag and the source mode. Severity is one of four application-level labels, shown in Table I. It is separate from the model output, which stays binary. The confidence value is displayed with each detection. The alert is stored in SQLite, an embedded database that keeps everything in a single file and needs no server [14]. Because of this, the alert history remains available after a page refresh or an application restart.

The dashboard is built with Streamlit and shows the state of the AI engine, the preprocessor and the model, the dataset in use, the monitoring mode, the detections and the alert history. In live mode it refreshes about every two seconds, which gives a near real-time feel for a file-based prototype [15].

TABLE I. Severity labels used by the dashboard

Severity	Meaning	Use in dashboard
LOW	Lower priority	Routine review
MEDIUM	Needs attention	Review as per local policy
HIGH	Higher priority	Review promptly
CRITICAL	Highest priority	Review first

IV. DATA FLOW DESIGN

Fig. 2 shows the data flow at Level 1. At Level 0 the whole system is one process, where the user selects or supplies structured log records and receives the prediction results and the alert history. Level 1 opens this process into five steps. Process 1.0 prepares the input by reading the CSV data or the monitored live log records from data store D1. Process 2.0 applies the fitted encoder and arranges the feature matrix. Process 3.0 runs the Random Forest classifier, using the saved model file in data store D2, and produces a class label with a confidence value. Process 4.0 builds the event metadata for attack records and computes the fingerprint. Process 5.0 inserts the non-duplicate alerts into the alert database D3 and refreshes the dashboard views, and the results are returned to the user.

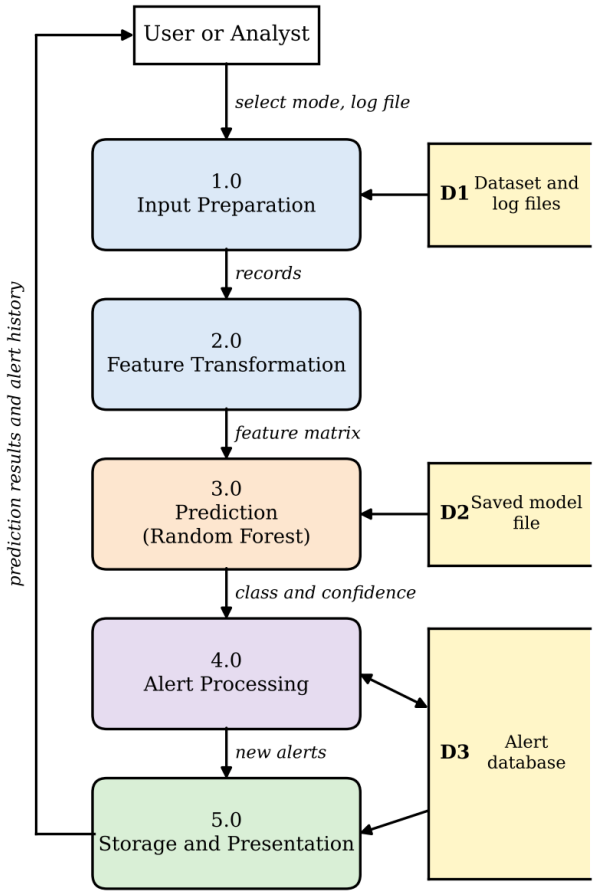

Fig. 2. Level 1 data flow diagram with data stores D1, D2 and D3.

The detailed logic of the alert processing step is shown as a flowchart in Fig. 3. A record that is predicted as Normal is shown as such and does not enter the alert path. For an Attack, the severity and the confidence are assigned and the SHA-256 fingerprint of the event is computed. The database is then searched for this fingerprint. If it is found, the event is treated as an alert that already exists and is not inserted again. If it is not found, a new alert is stored, and in both cases the result is shown in the dashboard. This check matters in live mode, where the same file is read again and again and the same rows would otherwise be reported repeatedly.

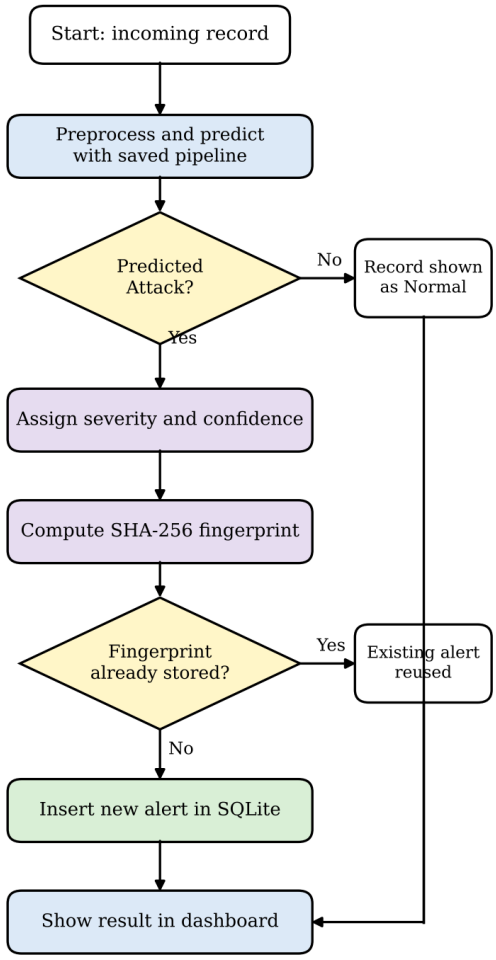

Fig. 3. Flowchart of alert generation and duplicate prevention.

SHA-256 is a secure hash algorithm specified by NIST, and it produces a fixed length digest from any input [16]. The same event always gives the same digest, and this gives each event a stable identity. The exact fields that are combined before hashing are not documented in the material available to us, and we do not describe them here.

V. IMPLEMENTATION

A. Dataset and Preprocessing

The dataset is a KDD Cup 99 style collection with 41 input features and one target label, which makes 42 columns. The features describe the protocol, the service, the state of the connection, the duration, the byte counts in both directions, connection counts and error-related indicators. The original attack categories are merged into one class. This binary design keeps the dashboard output simple, but it also means that the family of an attack, such as a denial of service or a probe, is not reported. The data is divided into 80 percent for training and 20 percent for testing with stratification, so that the share of Normal and Attack records is similar in both parts.

B. Model Configuration

The classifier is the RandomForestClassifier of Scikit-learn. The settings are listed in Table II. A fixed random state makes a repeated experiment give the same result in the same software environment. The balanced class weight adjusts the weights according to the class frequencies in the training data, and the parameter n jobs set to minus one lets the estimator use all processor cores.

TABLE II. Model and split configuration

Parameter	Value	Parameter	Value
n estimators	100	random state	42
class weight	balanced	n jobs	-1 (all cores)
Train-test split	80/20 stratified	Target classes	Normal = 0, Attack = 1

C. Software Environment

The prototype is written in Python. Pandas is used for loading and handling the tables, NumPy for numerical operations and Scikit-learn for the encoder, the pipeline, the Random Forest and the evaluation measures. Streamlit builds the dashboard, SQLite keeps the alerts, and Git with GitHub is used for version control. No graphics processor is needed, since the whole workflow is CPU based tabular learning, and an ordinary laptop with a multi-core processor and 8 GB of memory is enough for development. The Pickle format deserves a caution. Loading a tampered Pickle file can execute arbitrary code, so the model file should be created by the team, kept in a trusted place, and never loaded from an unknown source [17].

VI. RESULTS AND DISCUSSION

The classifier was evaluated on the held-out test split with accuracy, precision, recall and the F1-score. With Attack as the positive class, these are calculated from the numbers of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN).

$$Accuracy = (TP + TN) / (TP + TN + FP + FN) \quad (1)$$

$$Precision = TP / (TP + FP), \quad Recall = TP / (TP + FN) \quad (2)$$

$$F1 = 2 \times Precision \times Recall / (Precision + Recall) \quad (3)$$

The confusion matrix on the 98,805 test records is given in Table III. Of the 19,456 Normal records, 19,446 were classified correctly and 10 were wrongly flagged as Attack. Of the 79,349 Attack records, 79,336 were detected and 13 were missed. This is 23 errors in total.

TABLE III. Confusion matrix on the test split

Actual class	Predicted Normal	Predicted Attack	Total
Normal	19,446	10	19,456
Attack	13	79,336	79,349
Total	19,459	79,346	98,805

The measures calculated from these counts are shown in Table IV together with the rounded values reported by the project team. The small difference in the third decimal is only due to rounding.

TABLE IV. Evaluation measures on the test split

Measure	Value (%)	Reported (%)
Accuracy	99.977	99.98
Precision (Attack)	99.987	99.99
Recall (Attack)	99.984	99.98
F1-score (Attack)	99.985	99.99

Both kinds of error matter in practice. The 10 false positives would take some time of an analyst, and the 13 false negatives are attacks that could stay unreviewed. For a security tool the second kind is usually the more serious one, and a deployment should therefore watch the recall of the Attack class closely.

We would like to be careful in reading these figures. The test split comes from the same benchmark as the training data, and the KDD Cup 99 collection is known to contain

many duplicated records and to be far from current traffic [7]. Very high scores on such a collection are therefore not a promise of the same performance on a real network, and we report them only as test-split observations. The prototype was also not tested for throughput, latency or memory use, and no such numbers are claimed. The functional checks of the workflow, which cover the loading of the data, the label mapping, the encoding, the saving and loading of the pipeline, file analysis, live monitoring, alert generation, persistence and duplicate prevention, are described in the project report as behaviours of the implemented system.

VII. LIMITATIONS AND FUTURE SCOPE

The system has several limits, which we prefer to state plainly. It works only on structured CSV records, and it does not capture raw packets or connect to Zeek, Suricata or an enterprise Security Information and Event Management (SIEM) platform. Its output is binary, so the type of attack is not given. The formula behind the confidence value and the rules that map a detection to a severity level are not documented, and they should be written down and calibrated in a later version. The alert database has no access control or retention policy, and the model file must be protected from replacement.

There is clear scope for improvement. The model should be tested on newer and independently collected datasets, and a multi-class classifier could name the attack family. A controlled connector to a network monitoring tool, streaming ingestion for higher volumes, role-based access control, model version tracking and drift monitoring would move the prototype closer to real use. An option for analysts to mark an alert as confirmed or as a false positive would also give useful feedback for retraining.

VIII. CONCLUSION

In this paper we have described SentinelAI SOC, a log monitoring and threat detection prototype that joins a Random Forest classifier with a dashboard, a persistent alert store and fingerprint-based duplicate prevention. Keeping the encoder and the classifier in one pipeline removes a common source of silent errors, and the two monitoring modes share one prediction path. On the 98,805 record test split the classifier reached an accuracy of about 99.98 percent with 10 false positives and 13 false negatives. These values describe a benchmark, and the real value of the work lies in showing a complete and connected workflow, from a log record to a stored and displayed alert.

ACKNOWLEDGMENT

The authors thank the Department of Computer Science and Engineering, SJM Institute of Technology, Chitradurga, for the facilities and the support given during this project work.

REFERENCES

[1] K. Scarfone and P. Mell, “Guide to intrusion detection and prevention systems (IDPS),” NIST Special Publication 800-94, National Institute of Standards and Technology, Gaithersburg, MD, USA, Feb. 2007.

[2] R. Sommer and V. Paxson, “Outside the closed world: On using machine learning for network intrusion detection,” in Proc. IEEE Symp. Security and Privacy, 2010, pp. 305–316.

[3] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, “Intrusion detection system: A comprehensive review,” J. Netw. Comput. Appl., vol. 36, no. 1, pp. 16–24, Jan. 2013.

[4] A. L. Buczak and E. Guven, “A survey of data mining and machine learning methods for cyber security intrusion detection,” IEEE Commun. Surveys Tuts., vol. 18, no. 2, pp. 1153–1176, 2nd Quart. 2016.

[5] M. A. Ferrag, L. Maglaras, S. Moschoyiannis, and H. Janicke, “Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study,” J. Inf. Secur. Appl., vol. 50, Art. no. 102419, Feb. 2020.

[6] S. Stolfo, W. Fan, W. Lee, A. Prodromidis, and P. Chan, “KDD Cup 1999 data,” UCI Machine Learning Repository, 1999. DOI: 10.24432/C51C7N.

[7] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, “A detailed analysis of the KDD CUP 99 data set,” in Proc. IEEE Symp. Comput. Intell. Security Defense Appl. (CISDA), 2009, pp. 1–6.

[8] M. Ring, S. Wunderlich, D. Scheuring, D. Landes, and A. Hotho, “A survey of network-based intrusion detection data sets,” Comput. Secur., vol. 86, pp. 147–167, Sep. 2019.

[9] H. Hindy et al., “A taxonomy of network threats and the effect of current datasets on intrusion detection systems,” IEEE Access, vol. 8, pp. 104650–104675, 2020.

[10] L. Breiman, “Random forests,” Mach. Learn., vol. 45, no. 1, pp. 5–32, Oct. 2001. DOI: 10.1023/A:1010933404324.

[11] W. McKinney, “Data structures for statistical computing in Python,” in Proc. 9th Python in Science Conf. (SciPy), 2010, pp. 56–61.

[12] C. R. Harris et al., “Array programming with NumPy,” Nature, vol. 585, no. 7825, pp. 357–362, Sep. 2020.

[13] F. Pedregosa et al., “Scikit-learn: Machine learning in Python,” J. Mach. Learn. Res., vol. 12, pp. 2825–2830, 2011.

[14] SQLite Consortium, “About SQLite.” [Online]. Available: https://www.sqlite.org/about.html

[15] Streamlit Inc., “Streamlit documentation.” [Online]. Available: https://docs.streamlit.io/

[16] National Institute of Standards and Technology, “Secure hash standard (SHS),” FIPS PUB 180-4, Aug. 2015. DOI: 10.6028/NIST.FIPS.180-4.

[17] Python Software Foundation, “pickle: Python object serialization,” Python 3 documentation. [Online]. Available: https://docs.python.org/3/library/pickle.html