**Abstract-The world continues to grow more and more connected, even more so due to the Covid-19 pandemic. With people more spending time at home, smart devices are becoming more prevalent. As the market grows, so does the pressure to solve the paradigm of security of what are often simple devices. There is an expanding need for anomaly and intrusion detection with minimal overhead for smart devices. Tracing, the logging of system data, is a prime candidate for a solution. In particular, logs of system calls and the critical path are of interest in our paper. With said data comes with the problem of how to represent the data and how to analyze it. Due to the length of raw data that is produced with a trace, we must focus on key aspects and shorten its representation. As the analysis does not necessarily need to be completed on the device itself, overhead is less of an issue. We take a look at the effectiveness of Natural Language Processing and Long Short-Term Memory neutral networks and their ability to recognize standard system behaviour. Our results show training a neural network to learn standard system call and critical path data prove to be highly accurate.**

Keywords: Intrusion Detection, Anomaly Detection, System Calls, Critical Path, Tracing, LSTM, NLP

I. INTRODUCTION

As society allows digital ‘smart’ devices to wirelessly control even the more mundane aspects of their lives, they assume a level of security is present. The concept of connected household devices is known as the internet of things (IoT). The extra wireless connections provide attackers with extra access point for intrusion. With the amount of potentially sensitive data being sent through these connections, such as daily habits, banking details, internal cameras and microphones, it is paramount that manufactures ensure their devices keep their users’ data safe.

Due to the simplicity of many smart home devices, low overhead is essential as they lack the processing power that more traditional systems can leverage. As these devices are expected to be connected to a network, this only concerns the data collection. A trace is a log of system events within the timeframe of the trace. Tracing requires minimal resources making it ideal for this use case. The trace essentially contains a snapshot of the system during a time frame and its data can be used to detect anomalies. The trace can then be sent to a remote system to be analyzed. The remote machine must also filter data as the entire trace will not be relevant. The components this paper will focus on are the system calls and the critical paths.

A system call is a signal sent from a program that requests a kernel level service. A trace contains the list of system calls made while the tracing occurred with details on its execution. For the purpose of this paper, only the sequence of what system calls were used will be necessary. The critical path is a subgraph of the os execution graph, which displays the status of the various processes. The critical path shows only the processes that are necessary to complete before other processes can start. While the os execution graph contains many sections where the processes are running in parallel, the critical path is a linear graph, hence the reason it is called a path Francis et al[[1]](#endnote-2) . It shares its name with a similar concept in management and scheduling where tasks that must be completed before others can start are mapped out. The critical path also shows the processes that are affecting the execution time since other processes are dependant on those on the critical path. There are 11 possible statuses that can be displayed on the critical path (eg. running, waiting for network, blocked). Our experiment looks at the critical path sequence and the duration of each possible status.

Two methods of analysing the trace data were tested. The first was Natural Language Processing (NLP), where software is attempted to learn and understand and process natural human language. This field provides functionalities such as text comparison and prediction. To use NLP, trace data must be converted to text form, where as data from a trace is represented as a string. NLP is a powerful tool that powers spell check, voice and text recognition, and much more. These same principals could be applied to recognizing standard runtime and anomalies. The second method used was Long Short-Term Memory (LSTM), a Recurrent Neural Network. LSTMs attempt to mitigate issues where neural networks cannot retain data as new data pushes out the old. RNNs can be trained to predict expected critical path and system call data. If properly trained, data that does not match the values the model predicts can be identified as an anomaly and potentially an intrusion.

The paper proceeds as followed. Section II. discuses work done by others using similar concepts for intrusion detection. Section III. Explains the data we used and how it is gathered and represented. Section IV. discusses the uses of NLP to compare datasets. Section V. explains the neural network model based off on LSTM that was used in our experiments. Section VI. shows the results of the LSTM models. Finally, section VII summarizes our work and what possible future work can be done and is followed by section VIII. which has our references.

II. RELATED WORK

Our work certainly is not the first to use system call and traces to detect intrusions. Forrest et al[[2]](#endnote-3) is regarded to have introduced the concept of system call monitoring and Kim et al[[3]](#endnote-4) built upon their work. Du et al[[4]](#endnote-5) tested kernel module tracing on several different Arm processors. Their results required only a 5-6% overhead to record the traces, demonstrating tracing is viable on low power devices. Dymshits et al[[5]](#endnote-6) had previously tested training LSTM to learn off of system calls in a vector format. Kohyarnejadfard et al[[6]](#endnote-7) extracted data from system call streams using a sliding window method and their LSTM model had high accuracy. Wunderlich et al[[7]](#endnote-8) compared system call data with the kernel module as data points and showed that the kernel module was less reliable. Guan et al[[8]](#endnote-9) tested the use of both LSTM and transformers to classify system calls with high precision. Anand et al[[9]](#endnote-10) used NLP to find the difference between traces using the Levenshtein Distance.

III. DATA

A. Traces

In our experiments, we retrieve data from user space through kernel traces generated by LTTng (Linux Trace Toolkit Next Generation). LTTng allows for easy tracing on Linux systems. As programs in user-space are restricted from many functions, they must request a system call to request actions requiring the lower levels of the system. The reliance on communication with the kernel allows the monitoring of a systems status strictly from tracing the kernel space. A shell script was used to generate a dataset of traces. As a single trace could produce thousands of data points in seconds, we elected to capture a simple and short process. The Linux command wget from the GNU Project was chosen. The command sends a web request to an apache web server to retrieve the contents of a specified website. We wrote a shell script that starts an LTTng tracing session, sets it up, runs wget, and close the tracing session. It repeats this process for a user defined number of iterations, each generating a trace. As each request takes less than a second to complete, hundreds of traces can be generated in minutes with the use of the script. Despite its short length, there is plenty of data generated, allowing each trace to be a single record in our analysis.

B. Data Extraction

A trace contains a plethora of data, we use the program Trace Compass with the LTTng add-on to extract key data. Trace Compass is software designed for manual analysis of traces. However, there is an extension that allows for an embedded scripting environment. The Eclipse Advanced Scripting Environment, or EASE, is an add-on that allows users to write and run scripts written in JavaScript and Python. We utilized EASE to run JavaScript code to extract the critical path and the system calls from the traces which are outputted to external files. As thread ids can differ between run times, the script must first find the matching thread id to the corresponding name of the process of interest. There is no native way to get the system calls of a trace. As a result, the script must filter out the system call from the events. The system calls in the event log data consists of an entry and an exit for a system call. Every entry is put into a stack while until the corresponding exit is found. Then the system call can be saved using data from the entry and exit. The critical path can be accessed from the os execution graph. Once the data is read, it can then be processed into text output. In addition, Trace Compass has basic command line functionality, allowing us to import batches of traces and run scripts without the user having to manually open Trace Compass. We created another shell script in order to do so. This script and the previously mentioned in theory, should allows those who are unfamiliar with LTTng or Trace Compass to retrieve critical path and system call data from traces.

C. Data Representation

The resulting output contains a file for critical path and another for system call. They are simple text files with each new line containing a string representing the critical path and system calls sequences for a trace. For the critical path, each possible status is matched with a character. Each segment of the critical path is represented by the process’ name with said character appended to it (eg. wget-I, wget-j, wget-g). Due to the fact that there are more than 300 Linux system calls, they are instead represented simply with the name of the thread and the name of the system call. However, the sheer length alone makes it unfeasible to analyze it all without converting the data to alternate formats. In addition, neural networks usually expect numerical data, thus the data must be condensed. While there exists multiple predefine methods such as Word2vec[[10]](#endnote-11) and GloVe[[11]](#endnote-12), we elected to follow the work of Dymshits et al[[12]](#endnote-13). Their representation consisted of a vector with approximately 300 values, each corresponding to a possible system call and the number of times they occur. Our system call data is represented by an array of 315 values. Each value represents the number of times a specific system call occurs. We do the same for the critical path values with an array of length 10 for each of the possible statues. However, we create two vectors for the critical path, one for the number of occurrences of a status, and another for the total duration of time that the critical path is in a status for. Instead of converting previous output, we gather this data straight from the trace in an EASE script.Though an external program could easily do the same with a list of the system call and critical path sequence. The output for these representations are three comma-separated values (csv) files. Each cell represents a section in the critical path or system call sequence, each column a different status or system call and each row a different trace.

D. Dataset

We required a large sample of data in order to test our procedure. Unfortunately, our aforementioned method to extract data is not suitable for the number of traces to be analyzed. Due to limitations of EASE, it is impossible to close traces that have been opened in Trace Compass via scripting. This causes issues due to limitation in both memory and with Trace Compass itself when attempting to collect data for hundreds or thousands of traces. To ensure accuracy, we wanted to use thousands of data entries in our procedure. To do so we used a pre-existing dataset from [[13]](#endnote-14). Their data consisted of data for 36 different processes, each with four files. Each file contains different data for over a thousand requests for a process. We combined all the files to create three .csv files for the following: system call count, critical path count and critical path time. Instead of each trace representing a data entry, we made each process request a data entry providing us with 45,000 entries.

IV. NATURAL LANGUAGE PROCESSING

We first attempted to use natural language processing (NLP) to evaluate our data. Liddy et al[[14]](#endnote-15) defines NLP as: “Natural Language Processing is a theoretically motivated range of computational techniques for analyzing and representing naturally occurring texts at one or more levels of linguistic analysis for the purpose of achieving human-like language processing for a range of tasks or applications.” Having the data in string form allows the use of NLP functions for analysis and is easier for humans to read.

A common method of finding the similarity between strings is the Levenshtein Distance Algorithm, which provides the number of edits required to make the first string equal to the second[[15]](#endnote-16). Unfortunately, the size difference between strings has a large effect on the result, allowing two sequences of system calls of different lengths to be considered not similar even if they are both standard sequences. Nevertheless, it still has some use cases as the FuzzyWuzzy Python Library[[16]](#endnote-17) is built upon Levenshtein Distance and can be used to compare a system call or critical path sequence to a database of sequences. FuzzyWuzzy can then output a list of the most similar sequences. If the database contains known sequences of intrusion attempts, then the input sequence could be flagged if it meets a certain similarity threshold. Another algorithm that can be used to check similarity is the Cosine Algorithm[[17]](#endnote-18), which converts text into vectors and calculates the cosine between the vectors. As this is based on the direction of the vectors not the magnitude, the sequence length has less of an impact on the similarity. Unfortunately, this also means a process that takes five times as long as normal will be considered 100% similar if all values have the same ratio.

V. LSTM

Using the Keras Python library, we created a model to predict the expected values for the critical path and system calls. We use LSTM (Long Short-Term Memory), a recurrent neural network to train the model off the dataset. We first spilt the data with a (70%, 20%, 10%) split for the training, validation, and test sets. It is important to scale the values before training a neural network to decrease the variance which improves the accuracy.  Normalization is a common way of doing this scaling. We subtract the mean and divide by the standard deviation of each value. The training dataset is then fed into the model, training it. The model consists of the LSTM layer, a dense layer and an activation layer. The dense layer being simply an typical densely connected neural network layer while the activation layer applies a linear activation funciton. The model is complied with rmsprop as the optimizer. It is trained with 50 epochs and a validation split of 0.5. Afterwards, the test data is compared to the model’s predictions.

|  |  |  |  |
| --- | --- | --- | --- |
|  | System Call Count | Critical Path Time | Critical Path Count |
| Accuracy | 0.914626 | 0.914626 | 0.988706 |
| Precision | 0.076838 | 0.076838 | 0.711096 |
| Recall | 0.082773 | 0.082773 | 0.776180 |
| F1 score | 0.068038 | 0.068038 | 0.733936 |

VI. RESULTS

Table 1

Running in the Google Colab environment, we ran the three datasets into the model with good results for all three. The training validation returned high accuracy. With critical path time and system call count having a consistent 99%. However critical path count’s validation accuracy fluctuated between 83%-97%. The models then generate a prediction and the results are compare to the test data. The model for the critical path count dataset was the most accurate with 98% however had slightly lower precision than the other two. We also used the F1 score from the SKLearn package as an additional metric where critical path count also scored the highest. See Table 1.. Overall, the results show that all three types of data produce accurate results, with critical path count being the most accurate.

VII. SUMARY AND FUTURE WORK

While both critical path and system call data can be used to detect intrusions and anomalies, critical path count resulted in more accurate models and predictions. Those results could be further supplemented by NLP techniques. It may be possible to append the critical path data to the system call data, similar to how Sarah Wunderlich et al appended the kernel module to the system calls. Our work was only tested with expected values and additional testing with abnormal runtimes should be considered. Models trained with anomalies in their datasets would likely result in varying results.

VIII. REFEREENCES

1. PERFORMANCE ANALYSIS OF HETEROGENEOUS AND DISTRIBUTED SYSTEMSUSING KERNEL TRACING [↑](#endnote-ref-2)
2. Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. The evolution of system-call monitoring.InComputer Security Applications Conference, 2008. ACSAC 2008. Annual, pp. 418–430. IEEE,2008. [↑](#endnote-ref-3)
3. LSTM-BASEDSYSTEM-CALLLANGUAGEMODELINGANDROBUSTENSEMBLEMETHOD FORDESIGNINGHOST-BASEDINTRUSIONDETECTIONSYSTEMS [↑](#endnote-ref-4)
4. Du Y. et al. (2020) HART: Hardware-Assisted Kernel Module Tracing on Arm. In: Chen L., Li N., Liang K., Schneider S. (eds) Computer Security – ESORICS 2020. ESORICS 2020. Lecture Notes in Computer Science, vol 12308. Springer, Cham. https://doi-org.proxy.library.brocku.ca/10.1007/978-3-030-58951-6\_16 [↑](#endnote-ref-5)
5. https://arxiv.org/pdf/1707.03821.pdf [↑](#endnote-ref-6)
6. System performance anomaly detection usingtracing data analysis

   https://brocku-my.sharepoint.com/personal/nezzatijivan\_brocku\_ca/\_layouts/15/onedrive.aspx?id=%2Fpersonal%2Fnezzatijivan%5Fbrocku%5Fca%2FDocuments%2FMicrosoft%20Teams%20Chat%20Files%2FE2083%20%281%29%2Epdf&parent=%2Fpersonal%2Fnezzatijivan%5Fbrocku%5Fca%2FDocuments%2FMicrosoft%20Teams%20Chat%20Files&or=teams&originalPath=aHR0cHM6Ly9icm9ja3UtbXkuc2hhcmVwb2ludC5jb20vOmI6L2cvcGVyc29uYWwvbmV6emF0aWppdmFuX2Jyb2NrdV9jYS9FYTgxN0VfS2R0Rk9qenhQTm1saldYQUJCS2VLdTV6dmpGeEh5cHdZVXBxa0hnP3J0aW1lPXdhZF9mNGloMkVn [↑](#endnote-ref-7)
7. https://arxiv.org/pdf/1904.07118.pdf [↑](#endnote-ref-8)
8. https://brocku-my.sharepoint.com/personal/nezzatijivan\_brocku\_ca/Documents/Microsoft%20Teams%20Chat%20Files/research\_paper%20(6).pdf [↑](#endnote-ref-9)
9. https://vaastavanand.com/assets/pdf/tracey\_report.pdf https://www.cs.ubc.ca/~carenini/TEACHING/CPSC503-20/SOME-FINAL-PROJECTS/anandvaastav\_34615\_8015936\_VaastavAnand\_JoeWonsil\_report.pdf [↑](#endnote-ref-10)
10. https://arxiv.org/pdf/1301.3781.pdf [↑](#endnote-ref-11)
11. https://github.com/stanfordnlp/GloVe [↑](#endnote-ref-12)
12. https://arxiv.org/pdf/1707.03821.pdf [↑](#endnote-ref-13)
13. Quentin Fournier - Polytechnique Montréal https://github.com/qfournier/request\_analysis/tree/master/data [↑](#endnote-ref-14)
14. https://surface.syr.edu/cgi/viewcontent.cgi?referer=https://scholar.google.com/&httpsredir=1&article=1019&context=cnlp [↑](#endnote-ref-15)
15. Frederic P. Miller, Agnes F. Vandome, and John McBrewster. 2009. Levenshtein Distance: Information theory, Computer science, String (computer science), String metric, Damerau?Levenshtein distance, Spell checker, Hamming distance. Alpha Press. [↑](#endnote-ref-16)
16. https://pypi.org/project/fuzzywuzzy/ [↑](#endnote-ref-17)
17. SIDOROV, Grigori; GELBUKH, Alexander; GOMEZ-ADORNO, Helena and PINTO, David. Soft Similarity and Soft Cosine Measure: Similarity of Features in Vector Space Model. *Comp. and Sist.*[on-line]. 2014, vol.18, n.3 [cited 2020-12-17], pp.491-504. Available at: <http://www.scielo.org.mx/scielo.php?script=sci\_arttext&pid=S1405-55462014000300007&lng=es&nrm=iso>. ISSN 1405-5546.  <https://doi.org/10.13053/CyS-18-3-2043> . [↑](#endnote-ref-18)