Forensic Memory Classification using Deep Recurrent Neural Networks

Aishwarya Afzulpurkar
aa3852@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Master's Project is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Forensic Memory Classification using
Deep Recurrent Neural Networks

by

Aishwarya Afzulpurkar

A Capstone Submitted in Partial Fulfilment of the Requirements for
the Degree of Master of Science in Professional Studies: Data
Analytics

Department of Graduate Programs & Research

Rochester Institute of Technology
RIT Dubai
May 2020
Student Name: Aishwarya Afzulpurkar

Graduate Capstone Title: Forensic Memory Classification using Deep Recurrent Neural Networks

Graduate Capstone Committee:

Name: Dr. Sanjay Modak                          Date:
Chair of committee

Name: Dr. Ioannis Karamitsos                    Date:
Member of committee
Acknowledgments

I would like to thank my Capstone Mentor Ioannis Karamitsos for his constant encouragement and guidance throughout the duration of the M.S. degree as well as the Capstone project. I’d also like to thank the staff and faculty of RIT Dubai for their support.
Abstract

The goal of this project is to advance the application of machine learning frameworks and tools in the process of malware detection. Specifically, a deep neural network architecture is proposed to classify application modules as benign or malicious, using the lower level memory block patterns that make up these modules. The modules correspond to blocks of functionality within files used in kernel and OS level processes as well as user level applications. The learned model is proposed to reside in an isolated core with strict communication restrictions to achieve incorruptibility as well as efficiency, therefore providing a probabilistic memory-level view of the system that is consistent with the user-level view. The lower level memory blocks are constructed using basic block sequences of varying sizes that are fed as input into Long-Short Term Memory models. Four configurations of the LSTM model are explored, by adding bidirectionality as well as Attention. Assembly level data from 50 PE files are extracted and basic blocks are constructed using the IDA Disassembler toolkit. The results show that longer basic block sequences result in richer LSTM hidden layer representations. The hidden states are fed as features into Max pooling layers or Attention layers, depending on the configuration being tested, and the final classification is performed using Logistic Regression with a single hidden layer. The bidirectional LSTM with Attention proved to be the best model, used on basic block sequences of size 29. The differences between the model's ROC curves indicate a strong reliance on lower level, instructional features, as opposed to metadata or String features, that speak to the success of using entire assembly instructions as data, as opposed to just opcodes or higher level features.

Keywords:
Memory forensics, machine learning, LSTM, Attention
Table of Contents

ACKNOWLEDGMENTS .................................................................................................................. III
ABSTRACT ................................................................................................................................... IV
LIST OF FIGURES ...................................................................................................................... VI
LIST OF TABLES ........................................................................................................................ VI

CHAPTER 1 ................................................................................................................................ 1
1.1 BACKGROUND INFORMATION ............................................................................................ 1
1.2 PROJECT DEFINITION AND GOALS ..................................................................................... 2
1.3 STATEMENT OF THE PROBLEM .......................................................................................... 2

CHAPTER 2 – LITERATURE REVIEW ....................................................................................... 3
2.1 INTRODUCTION ..................................................................................................................... 3
2.2 MACHINE LEARNING HOST-BASED INTRUSION DETECTION .............................................. 6
2.3 USING HIGHER LEVEL SEMANTICS ..................................................................................... 7
2.4 USING MACHINE/ASSEMBLY CODE .................................................................................... 8
2.5 A COMBINATION OF HIGHER AND LOWER LEVEL DATA ................................................ 9
2.6 MOTIVATION FOR THIS PROJECT ....................................................................................... 11

CHAPTER 3 - METHODOLOGY AND ANALYSIS ................................................................... 13
3.1 CHOOSING THE SCOPE AND BREADTH OF DATA ............................................................. 13
3.2 COLLECTING THE DATA .................................................................................................... 13
3.3 DATA PREPROCESSING ...................................................................................................... 15
3.4 DATA MODELLING ............................................................................................................. 16

CHAPTER 4 - RESULTS AND ANALYSIS ............................................................................. 19
4.1 DATA STATISTICS ............................................................................................................... 19
4.2 MODEL RUNS ...................................................................................................................... 20
4.3 MODEL RUNS ...................................................................................................................... 22

CHAPTER 5 - CONCLUSION .................................................................................................... 25
5.1 CONCLUSION ...................................................................................................................... 25

REFERENCES ............................................................................................................................ 26
List of Figures

Figure 1: Hierarchy from lower level memory blocks up to user space applications.

Figure 2: IDA's graphical representation of basic blocks in chrome.exe. Within each block, a single call and a single jump exist, which are the two indirect blocks, which point to other direct blocks.

Figure 3: Diagram of the hierarchy between basic blocks, short term instruction sequences, intermediate term sequences and long term ones. Long term sequences represent app-wide end-to-end user interaction or per module patterns.

Figure 4: 3-D Input Matrix to the LSTM – non-overlapping sequences of basic blocks are fed into the model as individual data points, while basic blocks of a pre-defined size (5, 9, 19 or 29) belong to the recurrent, loopback sequence.

Figure 5: The LSTM layers, temporal max pooling layer, logistic regression layer, culminating in the output classification.

Figure 6: ROC Curve for Pure LSTM model with temporal max pooling and Logistic regression for classification

Figure 7: ROC Curve for LSTM with Attention and Logistic regression for classification

Figure 8: ROC Curve for Bidirectional LSTM with temporal max pooling and Logistic regression for classification

Figure 9: ROC Curve for Bidirectional LSTM model with Attention and Logistic regression for classification

List of Tables

Table 1: Basic block and instruction size statistics. The number of sequences denotes the total number of vertical entries in the LSTM input matrix or the number of rows. Each row contains instruction sequences of varying sizes, corresponding to the fixed number of basic blocks the instructions are a part of. The table shows the statistics of the row lengths, in terms of instruction size.
Chapter 1

1.1 Background Information

The recent boom in IoT, Big Data and Social Networking technologies has unintentionally led to an increasing rise in global cyber threats. Isolated attacks that previously exploited common vulnerabilities across systems have now given way to ‘scan-based’ attacks that identify and exploit system specific vulnerabilities across networks. These networks of compromised nodes, commonly referred to as botnets, representing personal computers, cellphones, even fax machines [26], morph, divide and link malware components in a way that allows unhindered, undetected propagation at incredibly fast speeds. Not only is the process of identifying and stopping the malware difficult, but certain species such as spyware make it difficult to even suspect that there has been an attack. Zero-day attacks that target firmware or hardware level components of a system essentially make it impossible to remove the malware, without a complete disassembly of the motherboard [8]. The scope of malware and its detection methods is prohibitively large so this paper will be focusing on one important aspect: volatile memory analysis on personal computers.

Memory forensics refers to the extraction and analysis of reliable volatile and non-volatile memory ‘dumps’ in order to infer the state of a machine at a given time interval. It is preferred over the injection of higher-level APIs as the latter is prone to interference by malware whereas the latest memory acquisition approaches have successfully been able to extract uncorrupt views of the system [16]. Analyzing memory blocks to reveal higher level information has so far been in the realm of a handful of security experts as it requires extensive knowledge and expertise in the area and is difficult to automate in a data-agnostic fashion. Recent advances in machine learning and deep learning have generated a plethora of new probabilistic approaches for malware detection, without the need for extensive expert analysis [5, 7, 11, 21]. Deep learning has achieved tremendous results in several areas such as natural language processing, that were previously thought to need supplementary semantic or causal models [12, 13]. This paper will explore the application of deep learning to forensic memory analysis, such that the expertise required to analyze a system’s memory will, to an extent, be acquired by the system itself.
1.2 Project definition and goals

The main goal of this project is to aid in the development of a memory-level view of the system that is *consistent* with the user-level view, residing in an isolated core with a hardcoded user-specific key required for memory-level RW access. The placement of and access to this view was determined by understanding the need for fast communication between system memory and the isolated analysis process as well as the obvious need for incorruptibility. This project shall focus on the generation of such a view, using a deep neural network architecture. Concretely, the goals are as follows:

- To find a probabilistic mapping between spatio-temporal read/write patterns in volatile memory and higher level activities.
- To utilize a hierarchical recurrent neural network framework to achieve layer-by-layer mappings between

![Diagram](image)

Figure 1: Hierarchy from lower level memory blocks up to user space applications.

1.3 Statement of the problem

Is there a way to achieve a 100% consistent and incorruptible view of the system that efficiently exposes and logs every running process on a personal computer?
Chapter 2 – Literature Review

2.1 Introduction

The importance for intrusion detection mechanisms was first implemented and penned to paper by Anderson [23] and Denning [24] in the 1980s. Since then, malware as well as its detection methods have evolved substantially and can today be classified along several dimensions.

Host-based detection methods perform analysis on end-points, or nodes, of a network that usually store malicious propagation code, while network-based detection prunes out malware from incoming and outgoing network data packets. Dynamic analysis refers to the probing and analysis of events during runtime, whereas static analysis attempts to detect malicious patterns without the need for code execution. Static analysis is usually much faster but less reliable than dynamic analysis and was the preferred method for commercial anti-virus products until the recent rise in computational parallelism and deep learning methods. Today, a combination of dynamic and static analysis is used in an effort to bridge the gap between the two using deterministic as well as heuristic processes [6,7].

Honeypots are a class of tools designed to attract or create ‘easy’ environments for a potential malware to intrude; if a malware succumbs to a honeypot, additional obfuscated code is peripherally placed in order to break into the malware itself and nullify its effects. Most such methods require prior knowledge of the nature and exploit strategy of targeted malware, which makes this approach difficult to scale. Sandboxes are perimeters around functionality that is meant to be monitored by an external agent residing outside of the perimeter. The most common type of sandbox is a virtual machine, allowing the host machine to monitor all virtual system activities. Unfortunately, there are types of malware that can penetrate through sandboxes, using the same system calls that transfer information about the virtual system to the host system.

Signature-based methods can easily detect malware that corresponds to a known byte-level pattern. Recent work has explored bit level patterns that are more efficient at detecting kernel structures [6, 10]. However, malware is able to perform similar functions using various different signatures so signature-based methods have a fundamental flaw of not being able to detect malware that has not been identified and added to the ‘signature database’ previously. Tools
such as Fuzzing improve the quality of signatures, randomly generating large sets of inputs that
are fed into a system, monitoring the outputs and making sure leaks don’t occur. However, due
to computational limitations, input sets are not exhaustive and small changes in a system’s code
may require vastly different types of fuzzer inputs.

Malware has been shown to potentially live on almost every part of a host, including the kernel,
OS, BIOS, PCI-expansion ROMs and their connected cards, USBs, firmware, power
management areas, system management area, etc. [2, 8, 15, 17, 20]. Limiting the functionality of
a system results in fewer side-effects and fewer ways in which a system could be hacked.
However, functionality cannot be restricted for it to perform to its full extent, and side-effects of
systems are often necessary. For example, out-of-order operations have led to the massive
growth in parallelism, but their executional implementation gives rise to a high number of
exploitable side-channels [25]. This makes it necessary to enumerate all possible ways in
which a malware can traverse its path, given that known and unknown vulnerabilities exist in all
parts of a system.

Volatile memory forensics deals with the correctness of system information in a more reliable
way than non-volatile memory forensics does. Hackers avoid leaving footprints on non-volatile
memory as it can be read without it executing its code. Volatile memory is where process
execution takes place and while non-volatile memory is larger in space, volatile memory
changes fast and is harder to keep track of. Nevertheless, certain volatile memory acquisition
processes have provably provided correct and complete views of the system, that are not
intelligible to a user in its raw form but can be analyzed without the need for external or dynamic
validation [4, 16, 19]. Commonly used frameworks such as Volatility, however, only provide
static dumps of the RAM, and require a separate acquisition of page tables and other kernel
level structures for contextual relevance.

Monitoring program execution on volatile memory, i.e. CPU registers, stacks, RAM, requires
tracing the program’s control flow graph. The machine or assembly code associated with
programs is linked in the form of direct and indirect calls within a program that span the entire
execution sequence. Fine grained control flow tracing is strict and makes sure that execution
follows the exact same sequence per assembly operation. Coarse grained control flow tracing
ensures the overall sequence per basic block (sequence of operations appearing in between
two call ops) is maintained. For this reason, coarse grained CFG tracing is more efficient but
less effective than fine grained CFG tracing [29]. It also supports confidentiality of programs and
prevents illegally reverse engineering intellectual property.

CFG tracing, also called Control Flow Integrity (CFI) is one way to perform dynamic analysis and fine-grained CFI can be combined with static analysis for increased efficiency, while maintaining effectiveness and not compromising confidentiality. Extended Fault Isolation (XFI) uses fine grained CFI as well as shadow stacks for protecting return addresses. [30] Program shepherding relies purely on dynamic binary rewriting, which is performed in the TCB. It is generally assumed that an efficient dynamic analysis framework is enough to spot deviations in functionality but the sophistication of attacks today render this assumption false. Dynamic program analysis cannot detect data being read outside of its own data region. One such program used for program shepherding is DynamoRIO, which provides an efficient framework for analyzing basic blocks and maintaining execution history up to a limit [31, 32].

Register liveness analysis is a compiler optimization that predicts the probability that a register will be used in the near future. While based on heuristic calculations, liveness analysis may affect the correctness of the running program, but not its security. This type of analysis is not only used for security enforcement but also compiler optimizations. While the parallel between security and optimizations tools runs deep, at times loops within programs formed by optimizations lead to distorted flows of execution such that similar API call sequences result in very different looking optimized assembly code. This can throw off many machine learning models that aim to predict "next-in-line" execution commands.

Taint analysis also traces program execution at the assembly instruction level but suffers from inefficiency - specifically has a performance overhead of up to 30 times. TaintQEMU is one such tool that performs whole system fine grained taint analysis. Because of its overhead, it is used offline and its insights are applied to rule based dynamic and static analyzers. [33]

Network Intrusion Detection Systems (NIDS) have been more popular than Host-based detection systems as they deal with a smaller state space and do not directly interact with the CPU. The additional buffer through which network packets pass before entering "execution mode" offers extra layers of security and platform for analysis. However, the volume and velocity of packet transfers, encrypted content and encrypted endpoint information presents challenges that cannot be overcome with current analysis techniques. Deep packet inspection technology, or the analysis of packet content, as opposed to structural and header information, is now a popular field that aims to combine host-based analysis with packet analysis to determine enforcement policies. [34, 35] Solely relying on network packet activity is not enough
as at times, dynamic analysis is required to understand the true functionality of packets, in which case running the packets through the CPU within a sandbox (using host-based techniques) becomes necessary.

Even though NIDS is not a 100% effective in catching malware on its own (and no system of analysis is currently, NIDS has proven to be a good testbed for machine learning applications in security. This is because it is easier to manage and generate network data, fundamentally spans a smaller state space, and has greater vulnerability because packets can only be monitored on a relatively sophisticated endpoint such as a computing device. It is difficult to monitor intermediate hops between base stations, switches, hubs and routers. Some NIDS based ML techniques have been effective on hosts as well: for example, network packet flow analysis uses very similar methodologies to control flow integrity technique. The focus of this paper is on host based analysis and below is a review on current ML techniques for host-based intrusion detection.

2.2 Machine Learning Host-based Intrusion Detection

The success of machine learning algorithms applied to the problem of malware detection depends heavily on the quality of extracted features. Traditional machine learning approaches are simplistic in nature and cannot extract contextual information from raw data, so current research is moving towards employing deep learning pipelines to be able to propagate different representations of features, each layer of the pipeline ingesting a more contextually complex set of features.

Malware detection can be performed by either analyzing and abstracting the functionality of benign programs or doing so for malware. Having generated a hidden representation of benign programs, deviations from this representation, or anomalies, are then labelled as malware. Solely analyzing malicious programs allows one to gain an understanding of the general concepts used by malware, and can generate a stronger categorization for different types of malware. This allows analysts to create a bounding box around the general concepts exploited by malware. This is insightful because malware behavior can mimic behavior of benign apps to a large extent, hence deviations from "normal behavior" can be difficult to detect. Understanding the inner workings of specific categories of malware yields rich features that can be fed into a "benign program analysis" framework, in order to generate a combined pipeline of different
inputs, different models and more accurate outputs. Using such combinations, throughout literature, has proven to always be more effective than focusing on a specific set of functionalities that span the system, but fail to branch out effectively.

Features used in machine learning based malware detection can divide into two subcategories: a) using lower level machine or assembly code and b) using higher level API calls, system logs, and events generated by applications.

2.3 Using higher level semantics

API calls cover a variety of functional spaces: registry, network sockets, memory management, etc. Combinations of inputs and corresponding outputs of the same API calls but in different parts of a sequence of execution are fed into several kinds of machine learning models to extract insights or predictive capacity.

Temporal API call information is harder to capture and is susceptible to more attacks – these features are usually modelled using Markov Chains with a prior understanding of general movement of calls. What makes temporal information difficult to process and utilize is the fact that most systems allow asynchronous processing of variable length API and system calls. This makes it necessary to pre-process temporal API call data using statistical analysis and information theoretic techniques before being able to analyze it and gain insights. [36]

Usually a combination of local and global calls is used for features. Local calls provide a large number of features within context. However, garbage collection and timing based attacks can successfully evade local call analysis, by injecting code at the system call level. Global calls make more robust feature sets as they take into account entire call streams across scopes, instead of individual calls that just locally branch out.

Behavior API call graphs have become a popular trend in the recent past. Heterogeneous information networks classify API calls using scope; for example two calls in the same file would have a ‘meta-path’ connecting them. Different meta-paths are used for different scopes, including function scope, package scope, code block, and invoke method. While theoretically sound, HINs fail to capture patterns outside of those that are already visible using standard, non-HIN techniques. [37, 38] This is because API calls are only as specific as the symbol table allows them to be, and any changes made dynamically in the symbol table are never reflected in
the higher level calls during program execution.

A good amount of machine learning effort has been put in with Android API calls, both in the context of dynamic and static analyses. DroidDolphin used Support Vector Machines to build a model using thirteen API call features and deployed the checking engine dynamically. CopperDroid used system calls, which are a level below application-based calls, and process communications (intra as well as inter) as features that were sandboxed and dynamically monitored by a virtual machine inspection framework. DroidMat used app permissions and intention messages as additional input along with API calls as features for a k-NN classifier. DroidMiner used an associative classifier using pure API calls and there are many others that have acquired similar but different results using the same features on different models such as Decision Trees and Regression based algorithms. [39]

The general problem with using higher level semantics is that it captures higher level, abstract relationships that can remain intact even on a compromised system. While this allows one to retain general contextual information over longer periods of time, it does not keep track of lower level changes that have a structure of their own. Malware that operates at the firmware level will almost always be able evade systems that use API calls as features, hence the need for processing assembly code.

2.4 Using machine/assembly code

Lower level code used to be off-limits simply because of its nearly unreadable semantics – the recent surge in lower level malware has pushed specialists to develop tools that help with the readability and analysis of assembly code.

Invincea [40] uses raw data without unpacking or filtering binaries to quickly train and deploy a low resource model. It uses PE import features, metadata information on PE files, and bin values of a two dimensional byte entropy histogram that models a file’s distribution of bytes. The data is fed into multiple DNN models, using ReLU as activation and high dropout rates. The insight found was that string values corresponding to metadata and aligning with higher level semantics override lower level details.

Byte n-grams is a commonly used feature types in static analysis. It condenses information within files such every n-gram is a unique combination of n consecutive bytes. This feature type
is easy to construct as it requires almost no knowledge of context. It was [41] observed that n-grams rely on and better generalize ASCII string information, or pre-defined data, rather than execution information contained in program code. N-grams can be constructed for program-level, language specific bytes, opcode, or machine language bytes, or DLL and PE import information bytes. Similar accuracies and results were observed across multiple values of n and types of n-grams used along with regularized Elastic-Net models [41]. This reflects the fact that n-grams extract little abstract, contextual information beyond string values, compared to models run on hand-crafted features.

The paper on KiloGrams [42] has reported positive results by increasing the size of n a thousand fold. The difference in accuracy and feature retention becomes significant at n=64. The theory behind this was that at large n sizes, the top k most frequent features will start getting redundant, allowing for a greater number of features to be included - the concept of approaching a redundancy limit on features by increasing the value of n before adding more features and repeating the process is referred to as hashing stride.

This shows the lack of effectiveness of machine learning for narrowly scoped tasks - training machine learning algorithms on chunks on data that appear within the same spatial and temporal space results in less generalizability and worse results. One more example of this phenomenon is described in [44]. This paper attempts to detect function boundaries using raw binary file bytes as data. In general it was assumed that analyzing return and call pointers as the only "features" would give one a fairly good idea about function boundaries, but the paper shows that the accuracy of such manual analysis is about 70% while a machine learning based approach, which uses features that a manual analyst may deem irrelevant, increases it to 98% accuracy.

2.5 A combination of higher and lower level data

The danger in combining lower level and higher level data is that lower level data may be perceived by an abstraction engine to be too noisy and chaotic, and might be inclined to ignore that data. It is important to systematically ensure that higher level features add value to the lower level ones, instead of overriding and diminishing them.

Microsoft published a paper [28] on using sequential models on lower level, opcode sequences
along with higher level system logs. The two models tested were: RNNs which have proven their strength with handwriting and speech recognition, and ESNs, which are used extensively for chaotic systems. They show that RNNs failed to capture salient features and that in fact Echo state network (ESN) models utilized all features more effectively and resulted in a much better accuracy. The hidden states of the recurrent models were fed into a temporal max pooling layer, to detected reordered temporal patterns, and logistic regression was used afterwards for performing the final classification. Only half the hidden states were fed into the classifier (referred to as half-frame), to avoid overfitting and leaky units were used to increase long-term memory.

A follow up paper [43] trained several models and configurations including Character level CNNs, RNNs, LSTM/GRU models and Attention models. 75,000 Windows PE files which were equally split between malicious and benign files were trained, validated and tested. The LSTM model with temporal max pooling outperformed the Char-CNN, GRU, Attention-based ones as well as all previously tested models i.e. RNN/ESN.

Apart from HIN models, all literature above trained machine learning models on file-based data, i.e. performed classification on files rather than smaller or larger chunks, and allowed a human analyst to narrow down the problem using patterns in infected files. This strategy may not always work because: a) malware can be disseminated across entire suites of files affecting critical and wide-ranging functionality or b) changes to single files can be too subtle for even a rule-based checking engine to detect them. This is evidently in practice already as malware is increasingly exploiting techniques such as return-oriented programming (ROP), Just-in-time compilation (JIT) and concept drift.

DeepCheck [29] uses CFI checking using a deep learning model. CFI retains lower level branches so it is fundamentally stronger but can be inefficient: updates in programs can completely change CFGs which requires modifying runtime libraries or binary files themselves – often not recommended because of security loopholes in doing so and the possibility of unexpected program execution. DeepCheck requires no such modifications because it uses an external framework based on predictions generated by the machine learning engine. It uses lower level details, i.e. the Intel Processor Trace, logging branches taken and not taken, as well as application binary code. This paper yielded good accuracy scores and showed that a multi-layered deep NN architecture is a lot better than using a single, large NN for malware detection.

Kernel Object Detection using Deep RNNs [1] presents a topic most closely related to the one
presented in this proposal. A graph is constructed from linked and adjacent lower level memory blocks and fed into a graph neural network as features, from which kernel objects were identified. The high classification accuracy of the model was dependent on the fact that kernel objects are represented in memory in a probabilistically deterministic way. Although subject to high amounts of feature pre-processing, the model in the paper presents a good benchmark for further tests of similar nature. A natural second step would be to try out the model for objects in the user space.

2.6 Motivation for this project

Control Jiujutsu's paper [45] shows how argument corruptible indirect call sites (ACICS) can be used to subvert order of execution while remaining within the boundaries of allowed call sites. A single heap/stack vulnerability can be enough to cause a compromise that can evade even fine grained detection mechanisms, let alone machine learning based, probabilistic techniques.

The paper uses, as an example, a sample target function, which is a piped log spawn function that takes any number of logging functions with user specified output ports/filenames and starts redirecting logs to those ports/filenames. This architectural pattern is commonly seen in web servers such as Apache HTTPD/Nginx. A single heap/stack vulnerability to corrupt the list of 'accepted' log port redirections on multiple scopes of checks, was all it took to cause the attack.

Because stack/heap vulnerabilities are difficult to mitigate on their own [] as it would entail enumerating every execution scenario and checking for it during every execution. For example, an arbitrary program downloaded from the Internet may have an insufficient garbage collection scheme that causes unused values to stay in memory. The heap usually takes care of this by removing such values after a certain period of time, i.e. garbage collection. In the meantime, if the “target” application, or the target of the malware, references such values, the stack/heap of the target would get infected. This can lead to any number of consequences such as return to malicious address or hardcoded value modifications, i.e. port numbers, website URLs.

As a result, most research today focuses on detecting and correcting malicious activity before it causes serious damage, as opposed to completely preventing it from infiltrating the system. The specific malware use case this paper focuses on, or the line of corruption is as follows:

a) Exploit stack/heap vulnerability which is provably difficult to prevent.
b) Cause system to ignore the corruption by exploiting limited context of operation, essentially due to a single thread not being aware of adjacent thread activity, both spatially and temporally.

This project’s goal is to investigate the intrinsic accuracy of using a selected pool of data to predict malicious attacks. Often, information from multiple data sources have significant overlaps and a repetition of overlapping data points can sway machine learning model weights. It is therefore important to have a primary data source which contains a fixed type of information, which can then be supplemented by data containing other types of complimentary information.

For example, log files contain system events which are usually piped once every X number of indirect calls. Usually there is quite a lot of gap in execution between two log events which may result in log events ignoring carefully implemented hacks. However, unexpected changes in logged event timings can be useful supplemental information, as these changes may not be indicative of an attack but will strengthen external evidence of an attack. While the malware may succeed in leaving the CFG unchanged, a few extra loops in inter-procedural calls could be maliciously executed. Those extra loops can be a sign of legitimate activity as well: legitimate programs that are genuinely waiting on user input/resource to be free, will cause noticeable delays in certain log events. However, complex executions that cause no delay in log events and no extra inter-procedural calls may indicate malware that is trying hard to not affect the log events. Malware almost never trigger log event delays because these delays always trigger external checks such as resource pooling checks, GUI checks and network checks. Attack surfaces may not be able to affect all three scopes so instead attackers choose to limit log event delays.

Control Jiujitsu’s paper [45] also shows that pointer dereferencing checks are a problem because usually only pointers generated during structural instantiation are checked. While shadow stacks and magic numbers allow constant value and function pointer checks, interprocedural pointer analysis remains ignored due to the computational complexity of keeping track of it all, either in the form of extra storage or AST traversals.

These limitations affect rule-based analysis however, supplementing rule-based analysis with contextual analysis derived using probabilistic models will definitely strengthen system immunity. This project focuses on choosing the right ‘learning’ model and the right contextual data that is able to identify interprocedural hacks without explicitly tracking softly linked pointers.
Chapter 3 - Methodology and Analysis

3.1 Choosing the scope and breadth of data

Lower level machine code can be used in several forms:

a) Raw memory bytes – Retaining all lower level bytecode can lead to the vanishing or exploding gradient problem, where the model ignores too much relevant information or fails to converge, due to too much randomness in data.

b) Instruction Opcode Embeddings (after discarding instruction arguments) – While opcodes themselves form a fairly limited set, they do not contain enough raw data to detect malicious changes. Microsoft’s paper, described in the literature review, supplemented opcode data with higher level log file data, as well as API function embeddings.

c) Function Embeddings – the set of learnable embeddings is too expansive, requiring a new models to be trained for every new version of software and/or firmware.

d) Functionality Embeddings – functionality refers to neighboring functions that should appear together, with at least one indirect-direct-indirect call pattern. This has the potential to capture common system/user interaction with single threads/processes.

e) Neighboring functions with longer “I-D-I” patterns – The longer the pattern, the more condensed and heuristic its representation becomes in hidden layers, which increases evasion likelihood. The shorter the sequence, the lesser the likelihood of finding deterministic patterns.

This paper focuses on exploring d) and e) as they have not been explored by previous literature and may provide insight on the feasibility of gathering contextual information from neighboring function calls without manual interference.

3.2 Collecting the data

The raw data used for this project came from memory dumps of single processes running on a Windows OS - this includes loaded modules, including both statically and dynamically linked
libraries. Assembly code for 50 PE files on Windows was gathered using IDA Dissassembler [46], including code for application files such as Google Chrome, Notepad, Command Prompt, and kernel files such as NTOSkrnl and NSLOOKUP.

10 types of malware published online, including Trojans, Rootkits, zero-day attacks, that operate both on the host as well as network were downloaded. All malware samples affected the kernel and/or RPC call functionality. So out of the 50 PE files used, 38 of them were affected by running the malware samples. Both the benign version as well as malicious versions of the files were used. So a total of 88 sample PE files were used for data.

Figure 2: IDA’s graphical representation of basic blocks in chrome.exe. Within each block, a single call and a single jump exist, which are the two indirect blocks, which point to other direct blocks (the green arrow for the call path, the red for jumps). The opcode is the string on the left, i.e. xor, mov, and function boundaries are conservatively assumed to be at call locations that do not revert back to the block they stemmed from.

A script was written to read IDA’s Basic block Extraction utility and modify the structure of the data so it resembled a 3-D matrix. Keras TF APIs [47] were used to build the model, train, validate and test the data on a single laptop with Core i7. The reason for not porting the model
to the cloud was because of the small number of files analyzed, as the creation of LSTM-consumable input was not purely automated. Differences in IDA's interpretation of file headers and data variables required some amount of manual analysis and crafting of features, though not as much as required by a purely manual feature base.

### 3.3 Data preprocessing

The input type for an LSTM is a three dimensional matrix, the axes representing the sequence length, timesteps, and batch size. Four basic block sequence configurations (i.e. direct - indirect - direct is a configuration of size 3) were tested:

1) a chain of direct and indirect blocks of size 5  
2) a chain of direct and indirect blocks of size 9  
3) a chain of direct and indirect blocks of size 19  
4) a chain of direct and indirect blocks of size 29

The reason for choosing basic blocks instead of whole functions as datum was that there are almost always more basic blocks than functions. Indirect calls are prime sites for exploitation and considering whole functions with multiple indirect calls as one chunk risked leaving out important contextual information about the general nature of indirect calls, i.e. pattern of appearance. Conditionals, loops, jumps, calls are all indirect edges connecting basic blocks, which consist of a determined sequence of executions. This also made feature extraction easier as extrapolating function boundaries would have required another machine learning model to be run as a pre-processing step.

The purpose here is to assign an execution order to an incoming stream of execution events. Execution events can be defined in terms of basic blocks, which are transformed into block embeddings. Our goal is to identify an inherent short term order of execution and possibly a general, app-wide, broad longer term order, without a need for order in intermediate block sequence orderings. This is because intermediate orderings often contain exponential state spaces that are hard to manage, as opposed to short and long term orderings.
Figure 3: The rounded rectangles in the first layer represent single basic blocks. The green layer is made up of multiple blue layers, which are made up of multiple black layers, which are made up of multiple basic blocks. The first layer shows a short term sequence of basic blocks, which makes up the second layer, the intermediate sequences. Intermediate sequences may or may not have patterns, so our goal is to use them to find patterns in the third layer, which are app-wide end-to-end user interaction or per module patterns.

3.4 Data Modelling

The base model chosen for this paper is the Long Short Term Memory model (LSTM) as it is great at retaining longer periods of information and has been provably more accurate than other sequence based models previously tested in literature. The LSTM has also been used in NLP, for sentence parsing and document classification. The hierarchy of words to sentences and paragraphs to documents is similar to the hierarchy of basic blocks to short, intermediate and longer term sequences. The regular RNN has a limited capacity to remember word probabilities beyond the phrase level and the LSTM enhances its capacity by remembering bursts of short term sequences on a longer term. This makes the LSTM better able to predict next words in sentences and paragraphs, but not in entire documents.

4 different model configurations are tested in this project:

1) Pure LSTM with temporal max pooling and Logistic regression for classification
2) Addition of attention to the model above
3) Bidirectional LSTM without attention
4) Bidirectional LSTM with attention
In all four scenarios, the LSTM predicts the next-in-sequence basic block. Instructions within the basic block are features of the x-axis, sequential basic blocks populate the y-axis, each row ending at a maximum of N basic blocks (one of 5, 9, 19, and 29).

![Figure 4: 3-D Input Matrix to the LSTM – non-overlapping sequences of basic blocks are fed into the model as individual data points, while basic blocks of a pre-defined size (5, 9, 19 or 29) belong to the recurrent, loopback sequence.](image)

Once the LSTM has run, its hidden states are passed in as input to the temporal Max Pooling layer. This layer acts as an aggregator and makes sure that blocks belonging to the same row entries in the data matrix can be re-ordered (taking into account execution optimizations and lower level resource handling). The temporal Max Pooling layer outputs single feature vectors corresponding to the basic blocks within individual modules of individual files. The reason for dividing files into modules was because the malware used for this project only target a subsection of functionality within entire files, and I was lucky to get labels for the specific functionality being targeted by each malware specimen. Hence, the logistic regression classification is done module-wise, not file-wise. Once the max pooling layer outputs embeddings for the modules, a supervised logistic regression classification is performed to obtain the final classification.
Figure 5: The LSTM layers, temporal max pooling layer, logistic regression layer, culminating in the output classification. LSTM figure taken from https://www.researchgate.net/figure/Unidirectional-LSTM-based-DRNN-model-consisting-of-an-input-layer-several-hidden-layers_fig3_320886290.
Chapter 4- Results and Analysis

4.1 Data Statistics

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Number of sequences</th>
<th>Maximum instruction size</th>
<th>Minimum instruction size</th>
<th>Mean instruction size</th>
<th>Median instruction size</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>51,768</td>
<td>78</td>
<td>11</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>9</td>
<td>28,760</td>
<td>124</td>
<td>57</td>
<td>88</td>
<td>88</td>
</tr>
<tr>
<td>19</td>
<td>13,623</td>
<td>254</td>
<td>97</td>
<td>152</td>
<td>159</td>
</tr>
<tr>
<td>29</td>
<td>8,925</td>
<td>306</td>
<td>110</td>
<td>201</td>
<td>204</td>
</tr>
</tbody>
</table>

Table 1: The number of sequences denotes the total number of vertical entries in the LSTM input matrix or the number of rows. Each row contains instruction sequences of varying sizes, corresponding to the fixed number of basic blocks the instructions are a part of. The table shows the statistics of the row lengths, in terms of instruction size.

I chose a 60 : 20 : 20 ratio for the training, validation and testing sets. For block sizes 5 and 9, the block sequence ordering was retained the way it appeared within programs as the block size is small enough to generate enough randomness. For block sizes 19 and 29, the block sequence was randomized for half the data and retained the way it was for the other half. This configuration generated the best and most consistent results.

Intermediate results are omitted in this paper as it took well over 30 runs per model to achieve optimal accuracy. The process involved readjustment of weights, fine-tuning of the model and addition of optimizations and regularizations, as well as dropouts.

The final LSTM configuration used for all 16 combinations of data and models was: Epochs = 50, Num_steps = 30, Batch_size = 20, Hidden_size = 500, Dropout = 0.3. The softmax activation function was used for the LSTM final layer and LeakyReLU was used for intermediate layers, including the recurrent layers. The advantage of LeakyReLU over ReLU is that it maintains state in a stickier manner by allowing a small gradient when a LSTM unit is not active. Bias weight regularization was not helpful but input weight elastic regularization (L1L2 = 0.01)
showed significant improvement for the models. A lot more available configurations were left untested due to the limited scope of this project, and are left as future work.

The logistic regression model included a single hidden unit, which performed better than multiple hidden units and used the ReLU activation function.

4.2 Model Runs

1) Pure LSTM with temporal max pooling and Logistic regression for classification

2) LSTM with Attention and Logistic regression for classification
3) Bidirectional LSTM with temporal max pooling and Logistic regression for classification
4) Bidirectional LSTM with Attention and Logistic regression for classification

Figures 6-9: ROC Curves for each of the 4 types of models, showing results for all four block sizes, the largest block size having best results across all charts, and the smallest size having the worst results.

4.3 Model Runs

From the charts, we see that using basic blocks of size 29 yielded the best results, and as block size shortened, model performance deteriorated. This is true across all four model configurations, and while the relative accuracies changed slightly, the general trends remained the same. The best model configuration was the Bidirectional LSTM with Attention, and this model also took the longest time to run. The bidirectional LSTM concatenates output from running the time steps in both forward and backward order. The attention mechanism, with an in-built temporal max pooling layer, is a better aggregator than the pure temporal max pooling layer that we layered onto the model externally. If the model was picking metadata or String features for each of the PE modules, the attention based models and the temporal max pooling
based models would have yielded similar results. This would have meant that lower level data was again being reduced down to noise. However, the fact that attention improved model performance tells us that lower level features are in fact being used, which shows an inherent pattern exists even within lower level machine code. This may be due to the large variance in PE files chosen to be part of the dataset – any similarity between basic block sequences within largely differing files is picked up, and especially so for the longer basic block sequences.

An alternate configuration was tried by feeding a combination of LSTM outputs generated by data of block sizes 29 and 19 into a common temporal max pooling layer. This was performed using the Bidirectional LSTM with Attention. The result was almost identical to the result produced by running just the data with block size 29. Again, this was probably a result of greater randomness in smaller basic block sequences, which the model considered to be noise. So the features of data with basic block size 19 consistently vanished, and the bi-directionality possibly enforced this phenomenon. Simply put, the smaller the block size, the greater randomness there is in the data, and the worse the performance.

Adding attention to the model affected results more positively than bi-directionality did. It is possible that certain instruction sequences across basic blocks inherently reflected bi-directionality, by literally showing up in reverse order within the blocks. Specifically, the addition of attention increased the true positive rate faster than the false positive rate, and the last model's ROC curves flat-lined past the 0.3 false positive rate mark. The same flat-lining can be observed for the first, pure LSTM model. For the other two models, there is a steady increase in the false positive rate, as the true positives increase.

We can also observe that the changes in performance between models more obviously affected the model runs using larger block sizes. The difference in performance for data using basic block size 5 was almost negligible.

The results seen here are in agreement with observations made in previous literature. While it may take more model runs with a much larger number of epochs to get truly good results, a general trend in block size configurations and model configurations can be made out from the charts above. For example, the KiloGrams paper also concluded that generating hashes across larger instruction segments (in thousands, hence “kilo”) resulted in far better performance than using simple n-grams (instructions of single digit sizes). Also, it seems that the decision to use entire instruction sequences was better than limiting the data to only using opcodes. The fear was that register numbers may generate too much noise for the model to retain important
information, but it seemed to have actually improved performance.

The ROC curves generated for the test data are slightly worse than those generated for training data. Without LeakyReLU, dropout and regularization, the variance between the two was even greater. While these configuration changes lessened variance, they did decrease the overall result. It is expected that using a larger dataset of PE files can resolve this issue in the future.

The need to train for several epochs and also using a larger dataset would mean possibly training on a larger cluster of GPUs, as those available on the Cloud. The models for this paper were trained and tested locally, on a Core i7 laptop to test for resource consumption patterns. It took over 48 hours to run each model configuration without “hot start” configured. Training models on the Cloud would have resulted in superior performance, and access to more binaries may have resulted in better accuracies, comparable to previous literature.

This paper used LSTM to predict short term sequences, but recognizes the need to use deeper pipelines to be able to find patterns in larger data contexts such as files or folders. The effects of the decision to use module-wise classification as opposed to file-based classification are vague, and further tests could be done to come to a conclusion on optimal classification sequence lengths.
Chapter 5 - Conclusion

5.1 Conclusion

Overall, the findings of this paper are positive, and reflect a general trend that using larger sequences of basic blocks as input to sequential models results in a stronger hidden representation. The addition of attention and bi-directionality to the core LSTM model significantly enhanced accuracy and results, even when using a relatively small dataset. The results achieved are in sync with results shown by previous work on similar topics, and while accuracies for this project could have been better, I am confident that testing on larger samples would bridge that gap. Overall, I can conclude that using neighboring basic blocks as data to a sequential predictive model such as LSTM works well for generating relevant context for program executions. Complicated pointer arithmetic and vague rule-based checks can be traded for efficient, external, probabilistic checks that supplemented static and/or dynamic analysis methods.

In the future, I plan to utilize the same concept of neighboring basic blocks, but for longer sequences, i.e. using document classification techniques. I would use results from this project, such as hidden LSTM states and max pooling embeddings, as features for this next step. Hopefully, I will be able to test the same concepts on a larger sample of files, both benign and malicious.

The purpose of this project was to aid in the detection of underlying patterns in machine level code that may directly link to the functionality of that code. Specifically, it was to find a direct mapping between lower level machine/assembly code and the functional requirements of or expectations from an application, circumventing API calls and other higher level functionality. The fact that the attention-based model outperformed the model using the max-pooling layer tells us that there are in fact such patterns. Uncovering specific patterns is beyond the scope of this project but is definitely a topic of interest for future work.
References


43. Athiwaratkun, Ben, and Jack W. Stokes. “Malware Classification with LSTM and GRU Language Models and a Character-Level CNN.” 2017 IEEE International Conference on


