Developing an LSTM-based Model to Detect Sandbox Evasion Techniques Used by Malware, Enhanced with Synthetic Data Generation for Improved Detection

Dilhara W. M. A.

(IT21299452)

Dissertation submitted in partial fulfillment of the requirements for the Bachelor of Science (Hons) in Information Technology Specialized in Cyber Security

Department of Computer Systems Engineering

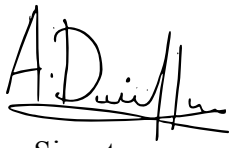
Sri Lanka Institute of Information Technology
Sri Lanka

April 2025

DECLERATION

“I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for a degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology, the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).”



Signature: Date: 04/10/2025

ABSTRACT

The growing sophistication of modern malware has rendered traditional detection methods inadequate for recognizing more advanced evasion techniques, especially those designed to bypass sandbox environments. This dissertation proposes a hybrid strategy that incorporates rule-based techniques with Long Short-Term Memory (LSTM) networks to detect sandbox evasion techniques in malware. The aim is to improve detection of evasive malware behaviors in a managed environment such as a sandbox that may be used for malware analysis. Through the use of machine learning techniques, specifically LSTM networks, this research will exploit sequential patterns in malware behaviors to detect evasive maneuvers, which are difficult to observe using traditional methods. The study is concerned with preprocessing sandbox behavior logs, tokenizing sequences of API calls, and mitigating class imbalance through approaches like SMOTE. These procedures focus on the needs to ensure rare evasion techniques note to be included in order for the model to generalize. The LSTM-based model we present in this paper shows notable advancements in detection of multiple evasion strategies with high classification accuracy. This capture process is realized on real data, and the results demonstrate the models capabilities to capture evasive behavior which provides promise to their ability to improve the security of the sandboxing system. Finally, the results emphasize the need for advanced data processing and machine learning models in malware detection frameworks to accommodate a growing complexity in modern cybersecurity threats.

Keywords: Malware detection, sandbox evasion, LSTM, machine learning, API sequences, class imbalance, SMOTE, cybersecurity.

ACKNOWLEDGEMENT

I would like to extend my heartfelt gratitude to **Mr. Amila Senarathne**, Senior Lecturer and Head of the Industry Engagement Unit at the Sri Lanka Institute of Information Technology, for his invaluable guidance, constant encouragement, and thoughtful supervision throughout the course of this research. His expertise and mentorship played a vital role in shaping the direction and quality of this study.

I am equally grateful to my co-supervisor, **Mr. Deemantha Siriwardana**, Assistant Lecturer, for his continuous support, constructive feedback, and timely insights that helped me overcome various challenges during this project.

A special word of thanks goes to our external supervisor, **Ms. Chethana Liyanapathirana**, Assistant Professor at the Department of Mathematics, Computer Science, and Digital Forensics, Commonwealth University of Pennsylvania, for her valuable contributions, external perspectives, and expert advice which greatly enriched the research process.

I also wish to sincerely acknowledge the **Department of Computer Systems Engineering at Sri Lanka Institute of Information Technology (SLIIT)** for providing the academic environment and necessary resources to carry out this research successfully.

Moreover, I would like to express my appreciation to my project group members for their collaboration, commitment, and shared efforts throughout the research journey. Their support and teamwork were essential to the successful completion of this project.

Finally, I thank everyone who supported this endeavor, both directly and indirectly, and contributed to the successful completion of this thesis.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF ABBREVIATIONS

Abbreviation	Description
LSTM	Long Short-Term Memory
SMOTE	Synthetic Minority Over-sampling Technique
API	Application Programming Interface
ML	Machine Learning
SVM	Support Vector Machine
VAE	Variational Autoencoder
RFC	Random Forest Classifier
GBC	Gradient Boosting Classifier
APT	Advanced Persistent Threat
FPR	False Positive Rate
TPR	True Positive Rate
ROC	Receiver Operating Characteristic
F1-Score	Harmonic mean of precision and recall

1. INTRODUCTION

Malware continues to rapidly evolve and become increasingly sophisticated. As a result, the cybersecurity community has to constantly evolve its defense strategies, primarily through tools that allow us to analyze and mitigate malware safely and effectively. Sandboxes remain one of the more popular tools to research and examine malware behaviors, as they create an isolated environment in which users can observe malicious behavior without compromising the host machine [1]. However, as malware changes and evolves, so do the techniques malicious actors use to evade analysis in a sandbox. Evasion techniques allow malicious software to see that it is being analyzed inside a sandbox and to alter the malware's behavior so that it appears benign and evades detection by ratio to sandbox. Furthermore, traditional detection methodologies often involve known static indicators based on previous encounters, which can easily become ineffective against new evasion strategies [2].

While sandboxing technology continues to evolve, malware authors are also developing methods to evade sandboxing systems, whether through code reuse from other malware or open-source repositories. These methods continued to challenge detection [3] and added to the requirement of dynamic, behavior-based detection approaches beyond traditional signature-based methods. Recently, the research community has employed machine learning (ML) techniques, such as Long Short-Term Memory (LSTM) networks, for malware detection, taking advantage of the sequential nature of the data, such as API calls, to identify anomalous behavior that may have been missed in traditional, static systems [4].Nevertheless, there has yet to be a significant shift in the research community's focus away from malware classification and known behavior detection to detection of evasive malware in real-time.

Many of the existing solutions are not designed to handle the increased complexities of modern-day evasion techniques, which leaves systems defenseless against attacks that have not been seen in the past [5]. This research aims to help bridge this gap by

proposing a system that merges rule-based methods with LSTM approaches in real-time detection of known and novel sandbox evasion strategies, ultimately providing a superior sandboxing approach for malware analysis.

1.1. Background Literature

Identifying malware and its evasion methods has been a major challenge in cybersecurity history. The threat landscape has shifted dramatically in recent decades, with malware becoming more complicated and capable of being ignored or bypassing security solutions. Because malware continues to evolve, the methods used for it to evade detection will continue to expand, which is why it is so important that researchers and practitioners begin to create more adept detection capabilities. One of the most common methods for analyzing malware is sandboxing, which is the use of a controlled environment to run and observe suspicious software without concern for the host system [5]. Sandboxing replicates a real-world environment enabling cybersecurity practitioners to observe malware behavior in isolation, without any risk to production systems.

Although it is essential sandboxing encounters serious difficulties as it struggles to identify evasive malware due to modern malware employing advanced strategies to detect when it is being executed in a controlled environment. Evasion techniques that malware might use to evade detection include environmental checks, timing attacks and self-modifying code which enables the malware to modify its behavior to not activate detection systems within the sandbox. The rise of sandbox evasion techniques stimulated a growing research focus on understanding how malware detects and alters its behavior inside a sandbox. Specifically, research has focused on what types of detections malware utilizes to establish the environment and alter its execution to evade detection.

The purpose of this literature review is to investigate the development of sandboxing as a malware analysis tool examining its functionality and limitations considering

modern malware's increasing sophistication. Additionally, The research will review promising research that examines various sandbox evasion techniques, and give examples of how malware evades detection [5]. Finally, The research will discuss the challenges involved in finding ways to provide stronger detection systems that counter evasive behaviors and address the need for adaptive approaches in detecting evolved or previously-unknown evasive techniques.

Overview of Malware and Its Evolution

The detection and mitigation of malware are of significantly importance in the field of cybersecurity, particularly with its evolutionary sophistication and global implications on systems, organizations, and even national security. Malware, short for "malicious software" is software that is purposefully developed to damage systems, steal sensitive information, and to harm the security and private implications of a system. According to Mohanta and Saldanha 2021, malware can specifically be defined as a software that is intended to induce anomalies to computing systems, gain access to unauthorized data, or disrupt systems in other means [5]. Over the years, malware has evolved to be one of the most important challenges for security globally, as malware continues to expand in scope and impact with new malware types introduced every year. As reported by Symantec in the 2018 Internet Security Threat Report, there were approximately 669 million various types of malwares detected in 2017, which was a breathtaking jump from the previous year. The persistent rise of malware demonstrates the ongoing race between criminals and security, each side tripping over itself to outmaneuver the other [6]. Additionally, malware is now threatening not only individuals and corporations, but is increasingly targeting key infrastructure, such as power distribution systems, and thus posing a serious national security threat [7].

Malware has drastically changed since its early days. Malware used to be limited to a few programs that didn't often cause a lot of harm. Historically, it is believed that malware can be traced back to the early 1970s with the first visible virus called the Creeper virus, written by Robert Thomas. The Creeper was not strictly considered

malware as it was merely an experimental virus that replicated itself between mainframes on the ARPANET and displayed the message, "I'm the creeper, catch me if you can!" [5]. The first formal antivirus software called the Reaper was written by Ray Tomlinson in order to combat the Creeper virus. As malware continued to progress, a distinction became clearer in the 1980s and 1990s where malware was developed with malicious intent. Among these were the Morris Worms which used vulnerabilities in UNIX systems back in the late 1980s and was the first cyberattack that caused legal action based on the U.S. Computer Fraud and Abuse Act [7]. The rise of the Internet in the 1990s caused malware to expand outside of local networks. In 2000, the ILOVEYOU worm spread around the world as an e-mail attachment, causing tremendous damage, estimated in the billions, to systems worldwide. It is during this time that malware attacks marked the futurity of generalization distribution and a larger scope of threats [8].

Malware continued to evolve throughout the 2000s by introducing new malware types such as rootkits, SQL injection, and crimeware kits. Various important attacks like the SQL Slammer worm in 2003 and the emergence of mobile malware as early as the early 2000s showcased new forms of malware and more aggressive tactics [8]. The 2003 attack from the SQL Slammer worm infected 75,000 computers in ten minutes, emphasizing the speed and potential devastation of current cyber-attacks [9]. The Cabir virus that infected mobile phones in 2004 marked the first malware that was developed to take advantage of a mobile operating system. Later, social media sites like Facebook and Twitter, as well as other social media, were infected with malware such as Koobface targeting the growing platform popularity. Emerging circa the late 2000s, the development of Advanced Persistent Threats (APTs) represents the next chapter in the evolution of malware. APTs are sophisticated, state-sponsored threats that deliberately seek to enter targeted networks, stealthily remain for extended durations, and exfiltrate sensitive data. The Stuxnet worm was one of the most significant examples of APTs, as it targeted Iran’s nuclear facilities in 2010, effectively being the first use of malware, as a cyberweapon, to cause physical damage to critical infrastructure [10].

As malware has developed into more sophisticated forms since the 2010s, it has become even more associated with cybercriminal activity, such as ransomware attacks and cryptocurrency mining. Malware such as WannaCry and CryptoLocker encrypted users' files and demanded payments in cryptocurrencies resulting in significant interruptions for individuals . Conclusively, with heightened relevance as IoT devices (attendants) are breached (more often) as cryptocurrency and IoT adoption has made them a target of botnets as they breach the devices to utilize them in DDoS attacks or use them as mining devices without the knowledge of the device owner [8]. The rapid expansion of IoT devices has led to the development of botnets switching from consuming user's computer as a bot to use users IoT devices as bots, without intention of notifying the device owner and increasing attack. This marks an important evolution as cybercrime and technological developments converge as new devices and technologies are created exposing new targets and surface attacks against malware.

Limitations of Traditional Detection Approaches

As malware and cyber threats continue to evolve rapidly, the approaches to detection and mitigation of malware will have to adapt. Traditional forms of malware detection, such as signature detection, have become increasingly ineffective at detecting evasive malware that morphs and adapts to evade detection. To combat these dangerous types of malware, there is a clear need for more advanced malware detection techniques such as sandboxing, which creates a controlled environment in which the analysis of malware can be performed and then studied. Sandboxing is a very important tool used in the field of malware analysis, because it allows an analyst to observe the behavior of malware in real time with confidence, while not placing the integrity of the host machine at risk [8]. As malware evolves and new evasive behavior and malware capabilities arise, sandbox evasion techniques have also arisen--which means that malware actively detects when it is being launched in a sandbox, and changes its behavior to avoid detection. Consequently, there is a clear need for a confiscated detection frameworks that are still capable of correct detecting

evasive behavior. As machine learning and artificial intelligence(AI) continues to be more commonly used to enhance detection and capabilities, it will lead to more promising answers to the newest malware and its various new and advanced behaviors. The morphing and evolving attack surfaces and malware behavior only reinforces the continued research and innovations in the detection and mitigation of these new attacks and behavior. The developed history of changing malware and malware behaviors also knows the need for an advanced detection technique [10].

Malware detection has been an integral component of cybersecurity for many years, with traditional detection methods, such as signature-based detection, heuristic analysis, and anomaly detection serving as the foundations for identifying malicious software. Signature-based detection involves the match of the known patterns of malware or "signatures", and is useful when dealing with known threats. Nonetheless, there are times when new threats appear, or existing malware simply tries to evade detection based on either behavioral changes or by changing its code entirely. Heuristic analysis aims to recognize unknown malware by examining the behavior of standard programs, but it is also prone to many false positives. Anomaly detection involves establishing baselines of normal behavior and monitoring for deviations from these baselines, but it also can fail if the malware is clever and mimics the behavior of legitimate behavior as the malware continues to operate. As malware sophistication has continued to evolve to become more polymorphic and metamorphic, meaning the malware consistently changes its signature (polymorphic) or its structure (metamorphic) at will, the concerns about the effectiveness against more change has risen to an extent where traditional methods are no longer effective. Alsulami et al. (2017) discussed these issues, indicating that traditional static approaches required consistent updates to address new malware variants, and that is why it continues to be an issue for malware detection systems because they sometimes take a long time to address the update and the product has the potential of being attacked and exploited until the update is made. In emphasis, even if detection systems maintain large static signature definitions, the signature databases are still at risk of becoming saturated as new, and novel malware appears [11].

To solve these problems, newer approaches have incorporated more sophisticated techniques, such as machine learning, and methods based on ensembles, which are able to adapt to the newest malware. Ensemble-based approaches, as demonstrated by Alsmadi et al. (2022), have the potential to improve the accuracy of detection by combining the strengths of different classifiers, reducing false positive rates (FPR) and overfitting. Their work showed that, in particular, ensembles using either AdaBoost or bagging, may lead to substantial improvements in the AUC (Area Under the Curve) of malware detection systems, thus contributing to the ability to detect malware with greater accuracy [12]. Likewise, of interest is the work by Wagner et al. (2009) who proposed a hybrid approach, combining process-related information and system calls by integrating graph kernels with support vector machines (SVM) for malware detection. Results showed that process-related information can provide additional context to help identify suspicious activities based on the relationships between processes and the system calls made while executing. The authors demonstrated their methodology was effective in detecting obfuscated malware that misled traditional detection techniques [13]. These advancements show the increased reliance on dynamic analysis methods for malware detection, which are becoming more necessary to keep up with the evolving tactics used by the modern malware to evade detection.

The Rise of Sandbox Evasion Techniques

Contemporary malware has been utilizing sandbox evasion techniques that have grown increasingly sophisticated as attackers continue to improve their techniques to evade detection by security systems. Sandboxing techniques include the checks that malware make of the environment in which it is executed, inspecting certain system variables or registry entries to determine whether the malware is running inside a virtualized environment (sandboxed) [5]. Other common techniques include timing attacks (where malware alters its execution or delays its execution based on time differences in the operating system) and code obfuscation (which can make it harder for analysts to identify malicious code due to structural changes of the code). Marpaung et al. (2012), conducted a complete survey of various malware evasion

techniques where these and other evasive behavior mechanisms are utilized, noting how attackers play obfuscate, fragment, and splice sessions to evade detection. Evasion mechanisms like these facilitate programmatic behavior that does not reveal the intended malicious activity of the malware to evade detection during analysis. Moreover, cyber attack evasion tools such as CheckPoint's InviZzzible provide evidence of the great complexity associated with the evolving evasive strategies, which relies on operational inconsistencies associated with time and operations of Windows Management and Instrumentation (WMI) to manipulate system behavior and evade detection [14].

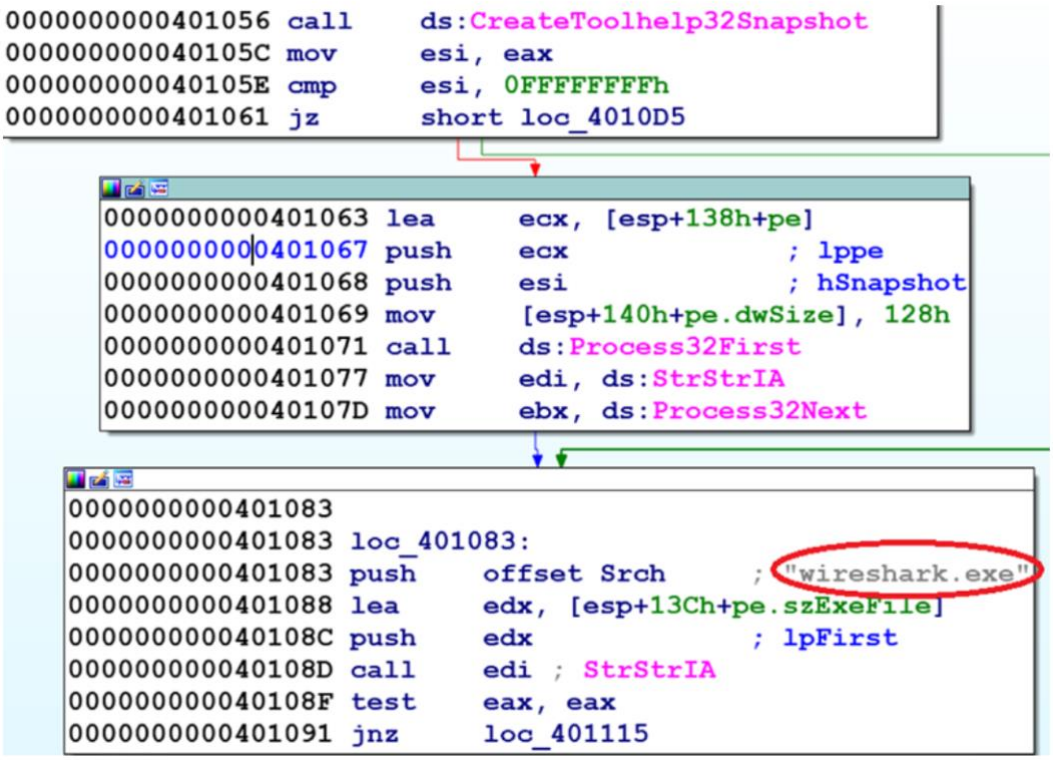


Figure 1 - malware sample that analyzed by IDA shows that it has an armoring feature that lists the processes on the system and checks if the wireshark.exe analysis tool is running

As shown in the above figure advanced evasion tactics utilize system calls and APIs to discover whether they are running in a sandbox. Malware may query sophisticated APIs like FindWindow to look for the classes of windows that a process has open instead of using the process name, which an analyst can easily change to evade detection. Mohanta and Saldanha (2020) demonstrated that malware can evade

detection by using an API like FindWindow to observe the actual front end of the analysis environment without relying on the name of the analyzed process. Another advanced evasion technique is virtual machine detection, where malware will check for indicators that a process is running in a virtualized environment such as CPU instructions or virtualized hardware. Finally, a time-based evasion approach may involve the malware executing its code after a delay, which in turn ensures that it remains dormant during the analysis of a sandbox and timelines only request execution after a time parameter is complete [5].

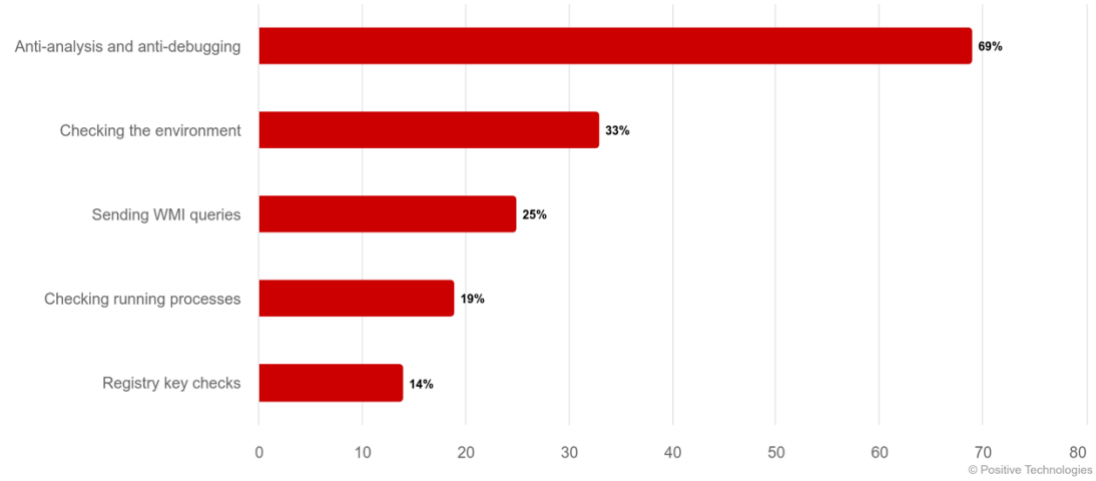


Figure 2 - Popular sandbox evasion and anti-analysis methods (percentage of malware)

According to the assessment of anti-sandbox techniques conducted by Positive Technologies which shown in the above figure 2, The most common category of anti-sandbox techniques used by malware is anti-analysis and anti-debugging techniques (69%). These techniques are meant to disrupt debugging tools or to detect that they are running, making it difficult for security professionals to analyze the behavior of malware in a controlled environment. The second most common evasion tactic included checking the environment, which was used by 33% of the malware samples. This involves checking system characteristics such as detecting a virtual machine to determine if the malware is running in a sandbox. Other noted evasion techniques include WMI queries (25%) to gather system information, checking for processes running (19%) to identify sandbox-specific processes, and checking registry keys (14%) to check for specific entries associated with sandbox tools. These

all demonstrate the degree of variation and sophistication used by malware authors to avoid detection and analysis while running in a sandbox environment. [2]

Detection of Sandbox Evasion Techniques

While the arms race between malicious codes and cybersecurity researchers continues to go on, understanding these evasion methods is important for enhancing the detection capabilities of sandboxing. Identifying and mitigating evasive behaviors will improve the accuracy, reliability and efficiency of sandbox analysis and increase the protection capabilities against sophisticated and evasive threats.

Identifying sandbox evasion techniques continues to be a challenge in malware analysis due to the evolved sophistication malware uses to evade detection. Detection systems today rely on rule-based detection methods (e.g., YARA, signatures) or heuristic-based analysis to extract evasive behavior present in the malware. These detection methods rely on matching rules and analyzing human-readable behavior from the system. While these detection techniques do have merit for identifying known evasion techniques, they have caveats when novel, unseen evasion techniques arise, as these are always changing. Jain et al. (2018) illustrated the challenge of today's malware, explaining that evasion techniques (e.g., putting an executable file into hide, introducing a delay between running analysis in the sandbox) have inhibited the ability for traditional sandbox systems to identify malicious activity. To this end, current systems have increasingly struggled to distinguish evolving threats or in real time detection [15]. While progress has been made in detection, a complete solution is still unavailable that automatically and effectively detects all kinds of evasion techniques across all sandbox threats, as malware authors will always innovate their methods to evade detection.

Additionally, advanced evasion technologies, like virtual machine detection and self-modifying code, complicate the detection system's work. Cheng et al. (2012) outlined those different types of evasion, like payload mutation and packet fragmentation, can defeat the latest intrusion detection-and-prevention systems, sandboxes, by

exploiting any weaknesses in their detection. Evasion techniques work, in part, by changing the flow of execution for the malware or by changing the execution environment to make the analysis impossible. While there has been advancement in the space, detection of evasion techniques of sandboxes is still far from complete, and many countermeasures do not answer all of the potential evasion attempts. There remains ongoing research and development to make sandbox detection systems more robust in the ever-changing and evolving area of malware evasion techniques [16].

Sandboxing as a Malware Detection Technique

A malware sandbox is an important tool for malware detection and analysis that provides a controlled, isolated environment to execute suspicious software (malware). This provides security professionals the ability to observe malware's behavior in a low- risk environment without putting their system at risk. One source, Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware by Abhijit Mohanta and Anoop Saldanha (2020), notes that the primary role of a sandbox is to capture notable system events created by malware, such as API calls, file and registry changes, network activity, and process behavior. Nearly all logs generated within the sandbox are useful for analysts when they are determining if a sample is malicious. In most cases a sandbox is implemented in a VM, hardware sandboxes are also available. In addition to sandbox platforms, monitoring events using event tracing tools and kernel drivers provide deeper monitoring capabilities of the sandbox, allowing the sandbox to monitor deeper activities like kernel-mode injections or changes to the system's memory space. The ability to monitor from multiple layers, improves the ability of the sandbox to identify often missed, subtle, or advanced malware actions not discovered by signature based methods simply because of a lack detailed monitoring [5].

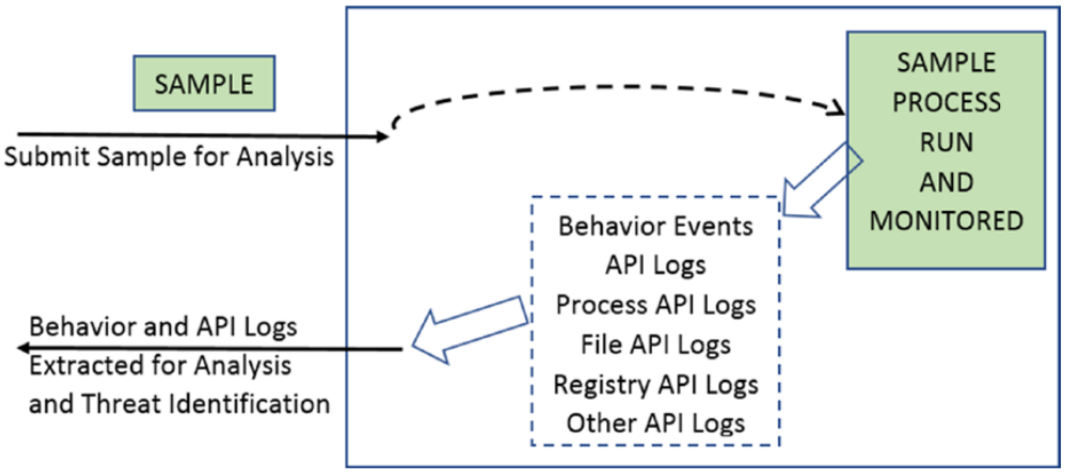


Figure 3 - High-level overview of the sample submission and analysis logs retrieval process of a sandbox

Sandboxes are an important resource in dynamics malware analysis, by enabling dynamic behavior-based threat detection by, it automates every sample within, [Mohanta & Saldanha [5]. Sandboxes allow security professionals to log and monitor activities and provide behavioral insights that static analysis cannot, subsequently easing the task of reverse engineers, so they can focus on debugging and disarming more sophisticated malware. Well-established security companies such as FireEye and Cisco have integrated their products to include sandboxing capabilities to improve the efficacy of threat detection, which is done via leveraging virtualized environments in order to safely observe the behavior of malware [Jamalpur et al., 2018 (43)]. Sandboxes such as Cuckoo Sandbox are designed specifically to log runtime behaviors, identifying behavior metrics such as changes in system calls or network activity, and are also a vital tool for monitoring new and evasive strain of malware [5] [15].

Sandboxes serve a purpose beyond detection and serve as an important way to collect threat intelligence. Organizations can analyze malware behaviors in sandboxed environments to create useful information about current adversarial tactics, techniques, and procedures (TTPs). The resultant intelligence can then be shared across other organizations to bolster defensive posture. According to Sinha & Sai (2023), sandboxing environments also offer an ability to integrate static and dynamic

analysis to enhance understanding of the malware. The capacity to run malware in a sandbox, capture system calls, and apply ML algorithms to identify patterns is very effective. Nevertheless, the value of the dynamic analysis conducted by sandboxes can face challenges when addressing highly evasive malware. As malware creators continually develop new evasion techniques into the malware, sandbox detection systems must evolve their systems to modify their tactics. Regardless of these imperatives, sandboxing systems provide an important and unique way of addressing and understanding the threats from modern malware. [17] [18]

Usage of Machine Learning and Deep learning in Malware Analysis

Machine learning (ML) has become an effective solution for detecting malware, especially as the types of malware become more sophisticated and evasive. For example, traditional signature-based methods of detection are unable to keep up with polymorphic and metamorphic malware, which changes its appearence too often. ML provides a continuously dynamic and adapting alternative because it detects malware based on the behavioral modeling of the malware's activities, such as system calls, file modifications, and network activity. In the paper by Prasad et al. (2024), the authors are demonstrating, in particular, the use of behavioral modeling for ransomware detection and that one can build a machine learning model to classify malware by its behavior rather than by a fixed signature [19]. They demonstrate how one can build an effective classification model too, which would classify the malware behavior into families and predict the type of attack, which can be as effective for classifying new/unseen malware samples. Similarly, deep learning, and Long Short-Term Memory (LSTM) networks, have proven effective at detecting sequential behavior of malware and identifying complex malware, such as evasion techniques.

Recent research comparing machine learning and deep learning methodologies for malware detection indicates that deep learning models achieve much higher detection rates relative to machine learning algorithms. According to Bokolo et al. (2023), deep learning models, especially those with neural networks, achieved greater

detection rates when detecting malware relative to the classic models - Random Forest and Support Vector Machines [20]. Bokolo et al. (2023) study indicated that deep learning models are particularly successful in classifying malware into families based on inspecting byte codes, opcode sequences and section codes related to malware behaviour. Likewise, Patil and Deng (2020), emphasize the unique advantages of deep learning models in automating malware classification, and noted that deep learning tools can detect malware variants at greater precision and speed than previous methods. Of all of the advantages deep learning can provide software engineers and security researchers in combating cyber threats, the ability to adapt and improve performance with the increasing complexity of malware makes it a personalized research avenue for the future of malware detection, particularly with respect to scalability, accuracy and performance [21].

1.2. Research Gap

Although malware detection and analysis have come a long way, there is still a crucial gap in detection of sandbox evasion techniques recently developed by malware authors. Traditional malware detection has used static analysis techniques, however for more advanced fa-cades or evasion techniques static approaches are limited. This is because static analysis tools rely on a malware's predefined signatures or static characteristics, thus are likely to fail against advanced evasion techniques or bait for desired behaviors, and always adaptive or evolving malware appropriately evade detection. In contrast, dynamic analyses observe malware based behaviors when executed and can identify malicious action. However, current and available tools, to include sandbox dynamic analysis tools, do not adequately execute malware and provide explicit detection evasion of sandbox evasion during scanning. This creates a gap in detection of evasion techniques used by cybercriminals to evade sandbox detection. Similar to static analysis darknet symmetry and regression techniques, sandboxes typically capture malicious behaviors without an explicit caveat on how malware evaded sandbox detection.

Research	*Analyze & Detect Malware evasion techniques*	*Integrate with a sandbox for analyze*	*Comparative detection*
M. Ficco, "Malware Analysis by Combining Multiple Detectors and Observation Windows," June 2022	✓	✓	✗
Comparative Study of Detection and Analysis of Different Malware with the Help of Different Algorithm,2023	✓	✗	✓
On the Effectiveness of Perturbations in Generating Evasive Malware Variants, 2023	✗	✗	✗
A. K. Sinha and S. Sai, "Integrated Malware Analysis Sandbox for Static and Dynamic Analysis," 2023	✓	✓	✗
X. Jianhua, S. Jing, Z. Yongjing, L. Wei and Z. Yuning, "Research on Malware Variant Detection Method Based on Deep Neural Network," 2021	✓	✗	✗
A. V et al., "Malware Detection using Dynamic Analysis," 2023 International Conference on Advances in Intelligent Computing and Applications (AICAPS), Kochi, India, 2023,	✓	✗	✓
Proposed approach	✓	✓	✓

Figure 4 - Research Gap Analysis

Sandbox environments are a critical aspect of malware analysis, for they enable the observation of the malware in a controlled environment while monitoring behaviors such as changes to files, system calls, and network traffic. However, while sandboxes have captured these actions, they are unable to differentiate between routine behaviors and tactics specifically intended to evade analysis. Sandbox evasion tactics, which are environmental checks, timing attacks, and process name checks, further obscure the analysis of the malware [22]. As for example a malware sample may exhibit delayed behavior or altered behavior based on the detection that it is being executed in a virtual machine or sandbox. Therefore, the sandbox report would indicate that the malware was benign, even if it was designed to evade detection.

Because sandbox reports do not provide clear evidence of malware evasion, analysts are tasked with manually sorting through large amounts of logs to find signs of potential evasion. This process is time-consuming, complicated, and prone to human error. As malware advances and becomes more evasive and complex, the significance of accurately and automatically being able to identify evasion will continue to become a greater issue in cybersecurity [23].

The inefficiency of the ability to detect evasion methods in sandbox reports is not only a problematic issue of detection capability but may also be a broader defect of the malware analysis method itself, particularly in the lack of comparative detection systems with existing malware analysis frameworks. Comparative detection compares the behavior of a specimen of malware to a known dataset of malicious behavior. This detection and analysis technique has not been heavily incorporated into traditional sandbox environments [24]. As a result, malware analysis does not often extend to analyzing malware in isolation from other malware behavioral traits or patterns. The limited ability to detect malware alone lessens existing malware identification methods. If a specimen of malware performs in a fashion similar to the evasive malware specimen, a detection method is available to automate and compare the behavioral patterns of both behavior samples. The accuracy and consistency of detection would significantly improve by identifying malware samples under observation or analysis if the behavior outputs of the samples were conducted similarly [25]. However, most available detection systems fail to use machine learning, or similar technique ungendered to detect behavior comparisons automatically and recognize similarities in evasion among malware samples based on behavioral sequences.

Also In addition to the use of machine learning models in conjunction with existing malware detection workflows, in particular Long Short-Term Memory (LSTM) networks, remains neglected. While LSTM networks have seen some success in analysis of sequential data, there remains little work applying them directly to sandbox evasion detection. LSTM networks have a unique advantage at modeling time series data, including sequences of API calls made during the malware

execution in the sandbox. When given sequences of actions taken by malware, LSTM models can learn long-term dependencies in the data and identify trends associated with evasion strategies, which suggests a fully LSTM model should be able to detect low interaction evasion as well as capture more complex evasion strategies involving API calls. Despite this potential, there is little work in the literature using LSTM models as part of the sandbox analysis process, in particular with the detection of evasion methods or policies. Current practice uses static features or agnostic heuristic rules to collect data in contained environments but does not consider that the evolution of evasion methods in malware execution and the inherent and dynamic nature of automation and evasion during sandbox use. This need for a hybrid model that combines both work being accomplished in dynamic analyses from a sandbox environment with the pattern recognition excellence of LSTM networks is clear, allowing for detection of malware while categorizing its evasion strategies as well [4].

Finally the research gap stems from the failure of present-day sandbox analytical systems to identify sandbox evasion tactics and there are many detection systems that leverage past behavior of malware in these systems that haven't been compared. Additionally, the lack of use of models, specifically LSTMs, to examine these evasion tactics provides a unique opportunity for innovation in malware analysis towards evasion detection. The research seeks to address the gap by developing an integrated system that adopts and leverages sandbox behavior data along with LSTM models, automatically detecting sandbox evasion techniques and improving malware detection efficiency and accuracy in an ever-changing environment. The goal of this research is to overcome the limitations of current systems and improve the capabilities of malware analysis sandboxes, providing a more reliable solution to evasion detection in malware behavior analysis.

1.3. Research Problem

The main obstacle with respect to current sandbox evasion detection systems is that they are heavily reliant on historical patterns and known API call signatures to

detect malicious behavior. Although this method of detection has generally been effective at detecting more commonly seen malware, it struggles to detect new or rare evasion techniques that malware uses when it does not follow normal patterns or known signatures. Even now, malware continues to evolve often using evasive techniques so sophisticated that they fall outside even the detection capabilities of traditional malware detection methods; those uses techniques do this often through more predefined signatures or static behavior patterns. Known evasive techniques, such as environmental checks, timing attacks, and self-modifying code, allow malicious programs to determine how they are being analyzed; therefore, alter their behavior to evade detection when they might be executing on a sandbox. Hence, the tactics malware uses to stay under the researcher's radar remain undetected, resulting in major holes in understanding the analysis and mitigation for such threats [5].

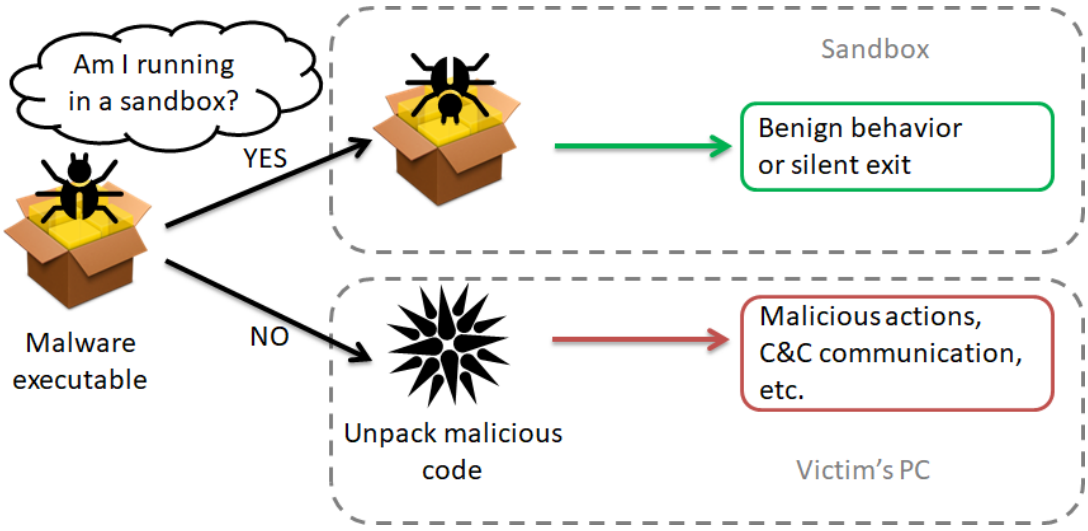


Figure 5- How malware evade sandbox detection

The problem statement central to this research is: **How can an LSTM model be utilized to predict sandbox evasion techniques employed by malware by integrating with a sandbox?** This question addresses the need for an automated and dynamic approach to detect evasion behaviors based on the sequential analysis of malware behavior. LSTM (Long Short-Term Memory), a type of recurrent neural network (RNN), can effectively learn long-term dependencies and recognize behaviors over time, which is advantageous for sequentially processing data such as

sequences of API calls. This research aims to develop a solution capable of detecting not only malicious behaviors but also evasive techniques exhibited by malware when engaging a sandbox through the application of LSTM model applied to sandbox behavior analysis [4].

Incorporating an LSTM model with sandbox analysis provides a more adaptive and automated detection mechanism that does not rely on historical patterns or defined signatures, and instead, it learns to recognize new evasion tactics from real-time execution data analysis. This model increases the scalability and reliability of malware detection, especially in instances where previous methods of detection were poor. By comparing the sequences of API calls undertaken by the maleware and detecting anomalous behavior with respect to normal execution patterns, the model will classify behavior indicative of evaisive tactics such as delaying action or querying system characteristics tied specifically to the sandbox environment.
This study seeks to address the limitations of traditional sandbox analysis tools with newly developed machine and deep learning techniques, thus offering an enhanced detection capability for evasion techniques. The hope is to develop a scalable, reliable, and flexible detection tool that can account for the evolving nature of malware and any increasingly elaborate evasion techniques. Through the use of LSTM networks with sandbox behavioral information, this research will offer a more comprehensive and adaptive malware analysis solution that ultimately improves malware behavior detection and evasion detections that current systems often overlook.

1.4. Research Objectives

The main objective of this study is to create an LSTM-based detection system for sandbox evasion techniques used by malware through behavioral analysis from sandbox reports. The expected outcomes of the system will merge machine learning techniques with sandbox analysis to identify not only malicious behaviors but also the specific evasion techniques used to avoid detection in a controlled environment.

The objectives below describe the different components and milestones necessary to achieve this goal.

Main Objective

To develop and implement an LSTM-based detection system that can automatically analyze **sandbox behavior data** and identify **evasion techniques** employed by malware. This system will enhance the effectiveness of current malware analysis methods by enabling real-time identification of evasive behaviors, improving the detection of both known and novel evasion strategies.

Specific Objectives

1. **Data Collection and Preprocessing**

Objective: Collect and preprocess a comprehensive dataset of **malware API call sequences** from **sandbox reports**, with each sample labeled according to the **evasion techniques** used (e.g., timing delays, process name checks, VM detection).

Rationale: A critical first step in training the LSTM model is the creation of a robust, well-labeled dataset. This dataset will serve as the foundation for the model, allowing it to learn the sequential patterns indicative of evasion tactics.

Expected Outcome: A diverse, high-quality dataset of **API sequences** with **evasion labels** will be constructed, ensuring that the model can be trained to recognize both common and rare evasion behaviors.

2. **Synthetic Data Generation**

Objective: Generate synthetic data to balance underrepresented **evasion classes** using **rule-guided synthetic API sequences**.

Rationale: Many evasion techniques are rare in datasets, which can lead to class imbalance and poor model performance. To mitigate this, **synthetic data generation** techniques will be employed to produce additional samples of **rare evasion tactics**, ensuring that the model learns to detect them effectively.

Expected Outcome: A **balanced dataset** will be produced, allowing the model to better generalize and identify a wide range of evasion techniques, reducing bias towards more common behaviors.

3. **Model Development**

Objective: Design and develop a **multi-label, bidirectional LSTM model** for detecting **sandbox evasion techniques**. The model will analyze **API call sequences** from sandbox reports, considering the temporal dependencies between actions to classify the type of evasion technique employed.

Rationale: LSTM networks are well-suited for tasks involving **sequential data** and **long-term dependencies**, such as the **API call sequences** observed during malware execution in a sandbox. A bidirectional LSTM model will be used to capture both past and future context in the data, improving the model's ability to detect complex, time-dependent evasion behaviors.

Expected Outcome: A trained **LSTM model** capable of **classifying evasion techniques** from malware behavior, with a focus on both accuracy and real-time application.

4. **Rule-Based Analysis for Further Insights**

Objective: Integrate a **rule-based system** with the LSTM model to provide deeper insights into the detected evasion techniques. This system will analyze the raw output from the LSTM and apply **heuristics** and **known detection rules** to validate and enhance the model's predictions.

Rationale: While LSTM models are powerful, rule-based systems can help refine the detection process by providing additional context and validation based on known evasion techniques and malware behaviors.

Expected Outcome: A **hybrid detection system** combining **machine learning** with **rule-based analysis**, ensuring higher accuracy and providing more detailed insights into the specific evasion techniques detected.

5. **Integration with Sandbox for Real-Time Evasion Detection**

Objective: Integrate the LSTM-based detection system with **real-time sandbox environments** to analyze malware during execution and detect evasion techniques dynamically as they occur.

Rationale: Real-time analysis is crucial for identifying evasive malware behaviors as they happen, allowing security analysts to respond promptly to emerging threats. By integrating the model with a live sandbox, malware behavior can be analyzed in the context of actual execution, improving detection efficiency and effectiveness.

Expected Outcome: A fully integrated **real-time detection system** that can automatically detect **evasion techniques** as malware executes in a sandbox, providing immediate insights into evasive malware activities.

Model Evaluation and Performance Metrics

Objective: Evaluate the performance of the LSTM-based model using **standard evaluation metrics**, including **precision**, **recall**, **F1-score**, and **accuracy**, to assess its effectiveness in detecting both **malicious behaviors** and **evasion tactics**.

Rationale: Performance evaluation is essential to determine the effectiveness of the model and its ability to generalize across different types of malware and evasion techniques. These metrics will help assess both **detection accuracy** and **false-positive/false-negative rates**, guiding further improvements to the model.

Expected Outcome: A comprehensive evaluation of the model's performance across different datasets and evasion techniques, ensuring that it meets the required standards for practical deployment.

2. METHODOLOGY

This chapter describes the stepwise design process undertaken to build a detection system to discover sandbox evasion techniques used by malware. The main aim of this methodology is to create a model that detects evasive behaviors in a real-time capacity by examining behavior data for malware that comes from the sandbox context. The research problem is the limitation of sandbox detection systems that rely on outdated, hard-coded rules or signatures; sandbox evasion techniques that are new or previously unknown are difficult to detect in offenders. The proposed methodology applies Long Short-Term Memory (LSTM) networks to analyze the behavior data from the live-sandbox context. Since LSTM captures sequence patterns from the sandbox behavior data, the methodology was designed to identify evasive techniques that would not have been detected using traditional sandbox techniques. The chapter explains the methodology process to collect, pre-process, and analyze malware data methods, in addition to the development of a dynamic, adaptive detection system to articulate detection of elderly and previously unknown evasion techniques.

1.5. Methodology

The selected methodology is consistent with the research objectives in that it seeks to fill knowledge gaps identified in existing detection frameworks while providing a more adaptable and extensible solution. The approach encompasses key aspects, such as data collection in sandbox environments, building off an LSTM model for malware behavior analysis, and integration with sandbox tools for real-time detection. Additionally, this approach consists of a comparative analysis of different evasion techniques that uses machine learning detection models to detect and classify multiple types of sandbox evasion. By addressing not only technical challenges in detection but also practical challenges in real-world implementation, this methodology presents a thorough approach to addressing the state of sandbox evasion detection. The approach in this methodology would likely allow for the

system to adjust to developing threats, thus providing a more effectual and efficient solution for cybersecurity professionals.

Overview of the Process

The process of identifying sandbox evasion techniques utilized by malware includes several steps addressing the research problem's various components. Firstly, malware (specifically involving the file system and the registry) was collected, and the malware samples were executed in the Cuckoo Sandbox spawning cloud environment. The sandbox is specifically configured and runs samples in a safe, segregated space in which they have no effect on production systems. The sandbox produces behavioral analysis reports for each malware example run on the specific operating system. The reports are critical for compiling the relevant content to understand how the malware behaves in the evaluation environment. The reports contain relevant information, including API calls made during execution, the OS system calls made, which processes and threads were spawned, and any network activity observable at the time of execution [5].

The next step involves collecting, preprocessing, and tokenizing the data to convert raw reports into a format suitable for input into a machine learning model. As discussed earlier, the dataset is imbalanced, so I will utilize synthetic data generation through rule-based methods to create additional examples of the evasion tactics that are underrepresented in the base dataset. Once the data has been prepared, Research is focusing build a Bidirectional LSTM (Long Short-Term Memory) model and then train that model to learn the patterns associated with sandbox evasion techniques and classify/predict multiple evasion techniques, based on the behavior patterns extracted from the sandbox reports and processed into a single input sequence. Once complete, the model is integrated with Cuckoo Sandbox for live evasion detection, where it will predict evasion techniques as malware executes. The system undergoes thorough testing and validation, and gets deployed to ensure the development system can detect an respond to evasion tactics, which is detailed in the following system diagram, which outlines the sequence of process illustration.

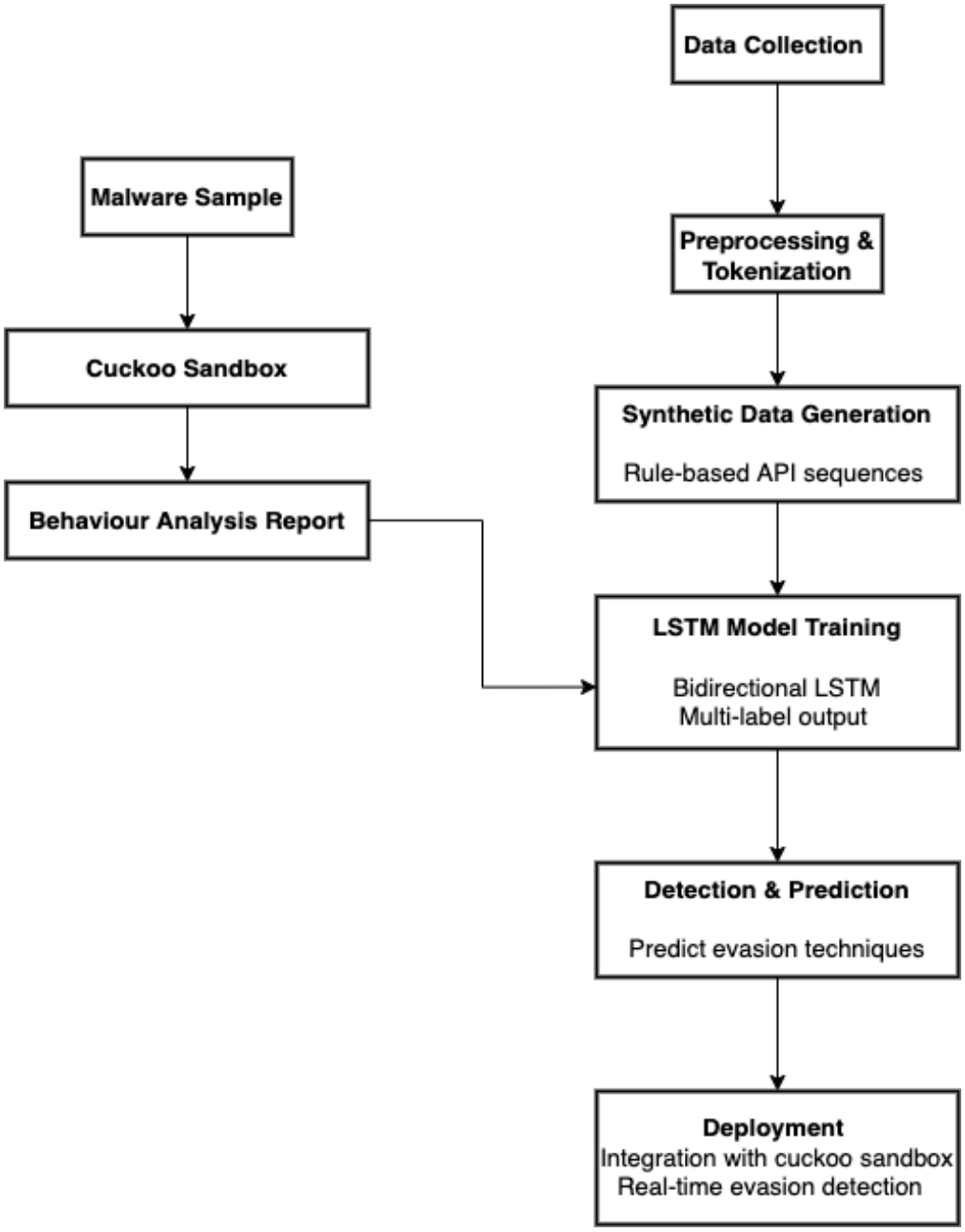


Figure 6 System diagram

The above system diagram outlines the full process for detecting sandbox evasion tactics performed by malware. Every stage in the diagram denotes an important part of the method, starting with the malware samples gathered and ending with the prediction of evasion tactics in a live environment when the model is deployed. The diagram provides an overarching view concerning how data and interaction will flow

through the system while demonstrating the relationships among the components. The flow of the system begins with the malware being executed in a Cuckoo Sandbox, moves on through the collection and preprocessing of behavior analysis reports, will continue to the generation of synthetic data to overcome imbalance, training of the model using LSTM, and ends with the encompassing model running in a live environment. The next sections will delve deeper into each of these steps contributing feedback regarding function, details of processes, and how each step worked towards the overall goal of detecting malware evasion tactics.

Data Collection

The first step in the methodology is to gather a rich and balanced data set of malware samples that specifically target a known sandbox evasion technique. Since the model we are developing will use sandbox analysis data, we must ensure that our data set comes from sandbox environments that truly depict the behavior of malware in the wild. Sandboxes, such as Cuckoo Sandbox, provide a way to execute malware in a controlled environment to capture detailed API call sequences, interactions with the underlying operating system, and other behavioral characteristics of malware. These behavioral characteristics are important for the detection of evasion techniques because they represent the malware behavior attempting to avoid detection in a sandbox environment.

In order to create this dataset, samples will be obtained from existing research, curated, publicly available malware repositories, and sandbox reports from actual malware analysis. The API sequences observed during sandbox execution will be the primary data type that will be used for the model because they have a clear and structured way of showing malware actions and interactions within the system. As sandbox environments are deliberately intended to imitate real-world environments in which malware will seek to evade detection, using data collected from sandbox analysis will ensure that the model will be training on data that accurately reflects the environment in which it will operate. Training the model on sandbox analysis data

will also increase its detection capabilities of new and potential evasion techniques, thus improving its accuracy and effectiveness within the detection system.
Four primary datasets will be utilized in this research:

1. **API Call Sequence Dataset**:

Source: This dataset is part of a broader research initiative on malware detection and classification using Deep Learning. It includes 42,797 malware API call sequences and 1,079 goodware API call sequences. Each sequence comprises the first 100 non-repeated API calls associated with the parent process, extracted from the 'calls' elements in Cuckoo Sandbox reports. This data is crucial for understanding the behavior of malware, particularly how it interacts with system APIs during execution [26]

Relevance: The dataset, cited in the work by Oliveira and Sassi (2019), provides a rich foundation for training the generative model to identify and simulate sandbox evasion techniques based on API behaviors [26].
(https://www.kaggle.com/datasets/ang3loliveira/malware-analysis-datasets-api-call-sequences)

2. **Mal-API-2019 Dataset**:

Source: Generated by Cuckoo Sandbox, this dataset is focused on Windows OS API calls and is intended for machine learning-based malware research. It is particularly valuable for studying metamorphic malware, which alters its behavior by modifying API call sequences. The dataset has been used in studies by Yazı and Çatak (2019) and serves as a benchmark for classifying malware based on API calls [27].

Relevance: This dataset will help address gaps in understanding how metamorphic malware changes its behavior to evade detection, a key aspect of this research [27].
(https://www.kaggle.com/datasets/focatak/malapi2019)

3. **CAPEv2 Dynamic Analysis Dataset**:

Source: This dataset consists of dynamic analysis reports generated by CAPEv2, an open-source successor to Cuckoo, widely used for analyzing potentially malicious binaries. It includes 26,200 PE samples (8,600 goodware and 17,675 malware) executed in Windows 7 VMware virtual machines. The dataset spans from 2012 to 2020, with malware samples sourced from VirusTotal and goodware from Chocolatey community-maintained packages. The analysis reports detail system calls, file system interactions, registry changes, and other critical behaviors observed during execution [28].

Relevance: The CAPEv2 dataset is particularly valuable for its comprehensive dynamic analysis reports, which include detailed behavioral information that can be directly leveraged to improve the model's ability to detect and simulate sandbox evasion techniques. The dataset also addresses challenges in malware analysis, such as mitigating evasive checks by malware designed to detect virtualized environments [28].

(https://www.kaggle.com/datasets/greimas/malware-and-goodware-dynamic-analysis-reports)

4. **API Sequences Malware Datasets**

Source: This dataset, available on GitHub, provides a dynamic malware analysis benchmark that is built using hashcodes of malware files, API calls from the **PEFile library in Python**, and the malware types obtained from the **VirusTotal API**. The dataset is presented in CSV format and contains two major sets: **VirusSample** and **VirusShare**. The **VirusSample** dataset includes 9,795 samples obtained from VirusSamples, while the **VirusShare**dataset includes 14,616 samples from VirusShare [29].

Relevance: This dataset is highly relevant for the research as it provides a rich source of dynamic analysis data that captures malware behaviors through API call sequences.

The dataset is well-suited for studying the detection of evasive malware, particularly with respect to how different malware families interact with system APIs during execution. By incorporating the **VirusSample** and **VirusShare** datasets, the model will be able to analyze a wide range of malware types and their corresponding API call behaviors. The inclusion of balanced datasets, after rebalancing to address class imbalances, further improves its usefulness for training machine learning models to detect various malware families, including **Adware**, **Trojan**, **Ransomware**, **Worms**, and others, as well as their evasion strategies. This dataset also allows for robust comparisons of performance metrics across machine learning classifiers, with algorithms such as **XGBoost** and **SVM** achieving high accuracy, making it an essential resource for training the proposed detection model [29].

(https://github.com/khas-ccip/api_sequences_malware_datasets)

Data Preprocessing and Tokenization

Tokenization and data preprocessing are an essential phase of creating malware behavior data for machine learning models. Tokenization and data preprocessing turn raw behavior logs into usable, structured formats suitable for training and testing. Initially, when preprocessing the data, it must be cleaned by removing any rows of times that did not employ even a single API call, which effectively ensures that only fully justified, appropriate API call sequences will go through the pipeline. Addressing evasion labels, is difficult when API calls sometimes I do not get a label or incomplete labels. In pre-processing, those incomplete or missing evasion labels were filled in with a default value, "No Evasion," to standardize the dataset.

Once the data is fully cleaned the next step is to tokenizing the API call sequences. In this process, each unique API call is mapped to a numerical token that is crucial for converting the textual data into a format that can be processed by machine learning algorithms. This is achieved by creating a tokenizer that breaks down the API sequences into individual tokens based on predefined delimiters, such as commas or

spaces. The tokenizer then converts the API sequences into numerical sequences of tokens, representing each API call by its corresponding integer. To ensure that all sequences have a uniform length, the sequences are padded to a maximum length, either specified by the user or determined dynamically from the data. Padding ensures that shorter sequences are extended to the same length, facilitating batch processing and avoiding issues during training.

Text to Sequence Conversion: The tokenizer converts each API call in the sequence into an integer based on the word index. The vocabulary size is calculated as the total number of unique API calls encountered during training.

Sequence=Tokenizer.texts_to_sequences(X)

Where:

- `X` is the list of raw API call sequences.
- `texts_to_sequences()` method converts each API call into an integer index representing a word in the vocabulary.

Padding: After tokenization, the sequences are padded to a fixed length `(max_sequence_length)` to ensure uniformity in the input size.

Padded_Sequence = pad_sequences(Sequence, maxlen=max_sequence_length, padding='post')

Padding is performed to ensure all input sequences have the same length, which is required for LSTM input.

Also, the multi-label classification operation requires encoding evasion labels for each malware sample. In this section, multi-label binarization was used on the evasion labels to transform each conceivable evasion approach into a binary vector. This allowed the model to anticipate numerous evasion tactics simultaneously during testing. The combination of tokenised API sequences and multi-label encoded labels produces a data format suited for input into machine learning models, notably LSTM

networks, which have demonstrated efficacy in modelling sequential data such as API call sequences. Thus, the preprocessing pipeline achieves the purpose of organising the data for training. The model can learn high-level notions about sandbox evasion tactics from sequential data, as well as how different combinations of malware API calls affect evasion strategy.

```
# Step 1: Load and Preprocess Data
def load_data(filepath):
    data = pd.read_csv(filepath)
    # Drop unsupported class (0 samples)
    data = data.drop(columns=['registry_environment_checks'], errors='ignore')
    # Separate features (X) and labels (y)
    label_columns = ['vm_detection', 'sandbox_process_detection', 'timing_evasion',
                     'user_interaction_detection', 'anti_debugging']
    X = data.drop(columns=label_columns)
    y = data[label_columns]
    return X, y

# Step 2: Split Data into Train/Validation Sets
def split_data(X, y):
    X_train, X_val, y_train, y_val = train_test_split(
        X, y, test_size=0.2, stratify=y, random_state=42
    )
    return X_train, X_val, y_train, y_val

# Step 3: Apply SMOTE to Training Data
def apply_smote(X_train, y_train):
    smote = SMOTE(random_state=42)
    X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
    return X_resampled, y_resampled
```

Figure 7 - using SMOTE to fix the class imbalance

To solve the challenge of class imbalance in the dataset, **SMOTE (Synthetic Minority Over-sampling Technique)** and class weighting methods were applied to improve representation of the rare evasion classes and avoid the case for modelling rare evasion techniques. SMOTE is a data augmentation technique that synthesizes samples of the minority classes in feature space. The method produces new instances by sampling points between existing samples from the minority class, again balancing the dataset to ensure the model has adequate examples to learn from for rare evasion techniques.

In addition, class weighting was employed in the model training process, giving increased weight to each of the minority classes higher weight meant to assign higher importance to the minority classes and reduce the built-in bias towards the majority classes. The combined application of SMOTE for data augmentation as well as class weighting for optimizing the model led the model to be trained on a more balanced dataset and improve the ability to detect and classify rare sandbox evasion techniques. These strategies may also improve model performance and allow for improved generalization to real-world malware samples with different evasion strategies.

Model Train

Selected Long Short-Term Memory (LSTM) networks to detect sandbox evasion techniques based on their nature to detect sequences where order matters. Malware behavior is sequential in nature, especially API call sequences; thus the sequence of API calls provides important contextual clues regarding the malware's intent, behavior, and evasive strategies. LSTM networks are an advancement of recurrent neural networks (RNNs) and specifically, are suited for sequential data due to long-term memory and retention of relevant information over time. This made them ideal for analyzing API sequences produced while malware was executing in a sandboxed environment. By taking advantage of the LSTM's ability to learn temporal patterns, our model can detect subtle behaviors including those that are indicative of evasive tactics that change within the analysis environment [30].

Model Architecture and Design

The model architecture **employs Bidirectional Long Short-Term Memory (Bi-LSTM)** networks to analyze and capture the sequential behavior of malware; in particular, it looks at API call sequences. LSTM networks are most efficient for our needs as they can learn temporally long-range dependencies in sequential data; for example, they can learn how a malware sample interacts with system APIs over time.

In malware analysis, the order and context are crucial for understanding the sample's behavior, as well as for uncovering evasive techniques [31].

Bidirectional LSTMs improve this procedure by looking at the input sequence both forwards and backwards. Standard LSTM networks process input data trajectories from the past to the future and make predictions based only on the context provided by the prior input[4]. In contrast to LSTM networks, Bi-LSTM networks have two layers: one runs through the sequence from past to future, and the other runs through the future to past. This type of sequencing allows the networks to learn dependencies in both directions and provide a more comprehensive representation of the input sequence. In the context of malware detection, this enables the model to evaluate the context for previous and future API calls, so it may be better equipped to detect subtle differences and relationships evidence of sandbox evasion techniques [32].

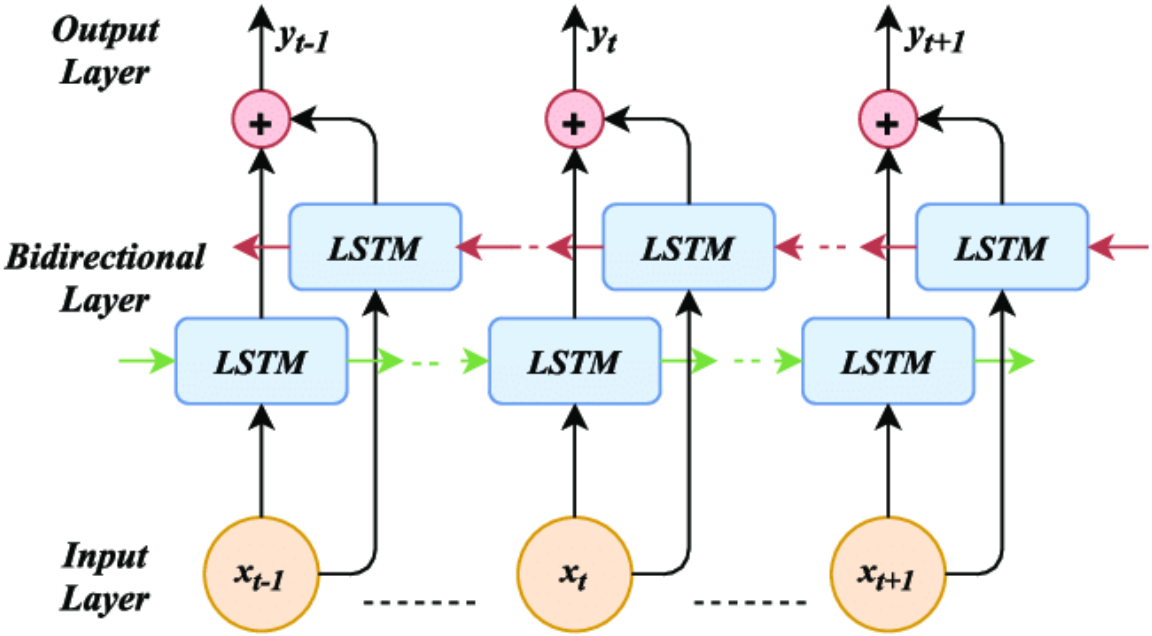


Figure 8 - Architecture of a Bidirectional LSTMs

As shown in the diagram, the Bidirectional LSTM layers of the model will apply to each API call sequence in both forward and reverse directions, allowing the model to rely on contextual relationships built throughout the entire sequence. This is particularly beneficial when detecting evasion techniques where action may depend

not solely on prior actions (e.g., previous API calls) but even on expected future actions (e.g., the malware may change behaviors when it believes that it has or is being analyzed). This more extensive approach to contextual networks will improve the model's classification accuracy across a variety of classes, especially in complex behaviors such as evasion across malware execution phases [32].

Embedding Layer: The embedding layer projects the input sequence (which consists of integer-encoded API calls) into a dense vector space of size 128. The embedding layer is initialized with random weights that will be trained during the training process. Each token (API call) is mapped to a dense vector of size 128:

$$\text{Embedding Output} = W_{\text{emb}} \cdot \text{Input}$$

Where:

- W_{emb} is the embedding matrix, which is learned during training.

Bidirectional LSTM Layers: Bidirectional LSTM layers are used to capture both past and future context in the sequence. Each LSTM unit computes the hidden state at time t_1, h_{t1} using the following equations:

$$h_t = \text{LSTM}(h_{t-1}, x_t)$$

Where:

- h_t is the hidden state at time tt,
- h_{t-1} is the hidden state from the previous time step,
- x_t is the input at time t (i.e., the tokenized API call at time t).

Since the model is bidirectional, two LSTMs are used:

- One processes the sequence from left to right (forward),
- One processes the sequence from right to left (backward).

Incorporating Bidirectional LSTMs into the model is designed to improve the model's ability to competently manage the complexities associated with malware behavior, allowing the learned patterns to be more resistant to subtle differences in

API sequences and recognize novel evasion techniques in real-time, thus becoming a strong method for multi-class classification of evasive techniques in a sandbox. This will fit closely with recent progress in machine learning on sequence data with Bidirectional LSTMs performing better than unidirectional methods.

Model Training:

The model training process involves splitting the dataset into three subsets: training, validation, and test sets. The training set is used to train the model, the validation set is used to tune the model's hyperparameters and monitor for overfitting, while the test set is used to evaluate the model's generalization performance. The training process is carried out using the **binary cross-entropy loss function** due to the multi-label classification nature of the problem, where each malware sample could exhibit multiple evasion techniques. The model uses the **Adam optimizer** for efficient gradient-based optimization and is evaluated based on the **accuracy** metric, alongside other metrics such as F1-score to balance precision and recall.

Binary Cross-Entropy Loss: The binary cross-entropy loss is used to compute the error between predicted probabilities and true labels for each class in a multi-label classification setting. The loss for each sample is calculated as:

$$L = -\sum_{i=1}^{N} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

Where:

- N is the number of classes (sandbox evasion techniques),
- y_i is the true label for class i,
- $\hat{y}_i$ is the predicted probability for class i.

The total loss for the entire dataset is the mean of the individual losses.

Adam Optimizer: The Adam optimizer is used for gradient-based optimization. Adam combines the advantages of both momentum and adaptive learning rates. The update rule for Adam is given by:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

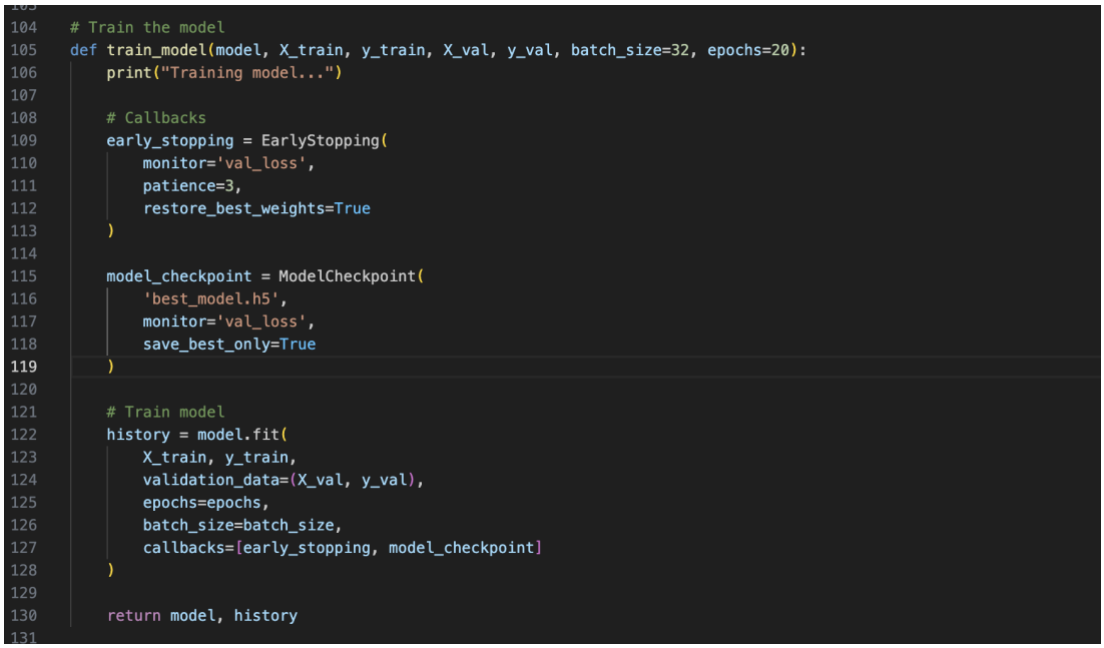
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_t = \theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where:

m_t and v_t are the first and second moments (exponentially decaying averages of past gradients),
g_t is the gradient at time t,
α is the learning rate,
β_1 and β_2 are the decay rates for the moment estimates,
ε is a small constant to prevent division by zero.

Early Stopping and Model Checkpoint: Early stopping is used to halt training if the validation loss does not improve after a specified number of epochs (patience). The model checkpoint saves the best-performing model during training based on the validation loss.



```python
# Train the model
def train_model(model, X_train, y_train, X_val, y_val, batch_size=32, epochs=20):
    print("Training model...")

    # Callbacks
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=3,
        restore_best_weights=True
    )

    model_checkpoint = ModelCheckpoint(
        'best_model.h5',
        monitor='val_loss',
        save_best_only=True
    )

    # Train model
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=epochs,
        batch_size=batch_size,
        callbacks=[early_stopping, model_checkpoint]
    )

    return model, history
```

Figure 9 - model training process

During the training phase, the model is constructed using a sequential architecture, embedding layers, and bidirectional LSTM layers to capture previous and future context in the sequence of the API calls. The output layer has a sigmoid activation function in order to accommodate for multi-label classification and to make simultaneous predictions for multiple evasion techniques. The model is trained with a combination of callbacks such as early stopping and model checkpoints to monitor the validation loss and reduce overfitting. Early stopping means that training ceases if the performance of the model is not improving, and the best weights are restored, while the model checkpoint saves the best model obtained during the training phase.

The model is trained for a specified number of epochs, incorporating early stopping to prevent overfitting and model checkpoints to save the best model. This allows for optimal model selection based on the validation performance. When evaluating the model on the test set, we generated predictions, compared them to the ground truth labels, and calculated metrics such as accuracy and the classification report to assess the model's ability to detect the different types of sandbox evasion techniques.

1.6. Commercialization Aspects of the Product

The proposed sandbox evasion detection system offers significant potential for commercialization in both the cybersecurity industry and other sectors that require robust malware detection capabilities. The model can be deployed as a software solution for cybersecurity firms, enterprise IT departments, and government agencies seeking to enhance their malware detection infrastructure. Its ability to detect complex and evasive malware behaviors positions it as a crucial tool in the ongoing battle against cyber threats.

In terms of commercialization, there are several avenues for product development and monetization. First, the system can be offered as a Software-as-a-Service (SaaS) product, allowing organizations to subscribe to a cloud-based service for real-time malware analysis. This model ensures that clients have access to the latest updates and security features without needing to maintain their own infrastructure. The system can be integrated with existing security solutions, such as antivirus software and intrusion detection systems, enhancing their ability to detect and mitigate new, evasive threats. Additionally, the platform can offer a comprehensive set of reporting tools and dashboards that provide insights into detected threats, allowing security professionals to make informed decisions.

Another opportunity lies in licensing the technology to enterprise cybersecurity providers who can integrate the detection system into their own products. Partnerships with technology providers such as cloud service providers and software vendors could help scale the product's reach and improve its adoption across different industries. Furthermore, the system can be adapted for use in sectors such as banking, healthcare, and critical infrastructure, where detecting and preventing cyber threats is a top priority.

From a business perspective, offering consulting services for customized deployments and training on the use of the system could be a profitable avenue. Providing industry-specific solutions and insights tailored to different types of

malware evasion tactics would give the product a competitive edge in a crowded market. Lastly, ensuring compliance with relevant cybersecurity regulations and offering ongoing maintenance and updates would help create a long-term revenue stream while addressing the evolving needs of the cybersecurity landscape.

In summary, the commercialization of this sandbox evasion detection system has the potential to make a significant impact on improving security practices across various industries, providing a scalable and adaptable solution to emerging threats in the cybersecurity domain.

1.7. Testing & Implementation

This section describes the testing and deployment stages of the research, concentrating on assessing the trained model and its deployment. The purpose of these testing stages is to assess whether the model may accurately detect techniques for sandbox evasion. We will measure performance metrics including accuracy, precision, recall, and F1 score. The model will undergo testing based on varying datasets to examine whether it is able to generalize to new, unseen data, and in order to test the robustness of the model against different evasion methods typically observed in malware. By testing the model thoroughly, we can be confident that it will be able to categorize different evasive behavior and make accurate predictions.

The implementation phase involves building and implementing the techniques described in the methodology section of this thesis; specifically data preprocessing, tokenization, and addressing class imbalance, which are necessary to prepare the dataset for model training. Data preprocessing included extracting and cleaning API sequences from the behavior reports of malware samples, tokenizing, and padding the data to make it compatible with the model input requirements. In addition to preprocessing data, to model performance in detecting rare evasion techniques, dataset augmentation and generation of synthetic data for underrepresented classes addressed class imbalance. Once the model is trained, the model is then used to analyze the malware behavior reports to predict sandbox evasion techniques based on the sequences of API calls. The model's predictions are evaluated to measure the performance of the model to ensure it is capable of a functional utility within the context of malware analysis in the real world.

Data Collection and Preprocessing

The dataset utilized in this study was taken from Khas-Ccip's (2023; see Khas-Ccip, 2023) "API Sequences Malware Datasets" on GitHub, which consists of a dynamic malware analysis benchmark based on API calls from the PEFile library in Python and malware types identified from the VirusTotal API. The dataset consists of two

main sets: The VirusSample set which has 9,795 samples and the VirusShare set which has 14,616 samples. This dataset has significant value in that it provides an API call sequence documented with the evasive methods malware used during the sequence's execution process, thereby providing researchers with an understanding of the malware's and its interaction with system APIs during execution. Although the dataset provides an example of inherent class imbalance, it was still included in both the training and testing environment because of its value to the study, as it has an example of Adware, Trojan and Ransomware makes, as well as evasive methods employed to deter malicious software detection. Further, the aspect of imbalanced data allows the model to achieve higher accuracy which further strengthens its suitability for training machine-learning models that detect and identify various evasion techniques and malware behavior.

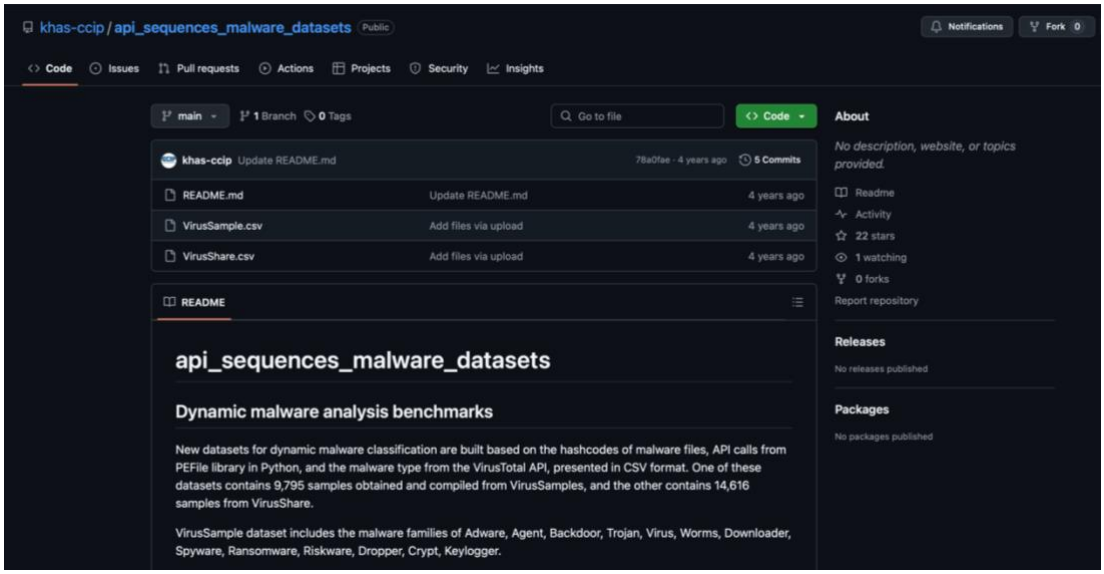


Figure 10 - Evidence for Dataset Source

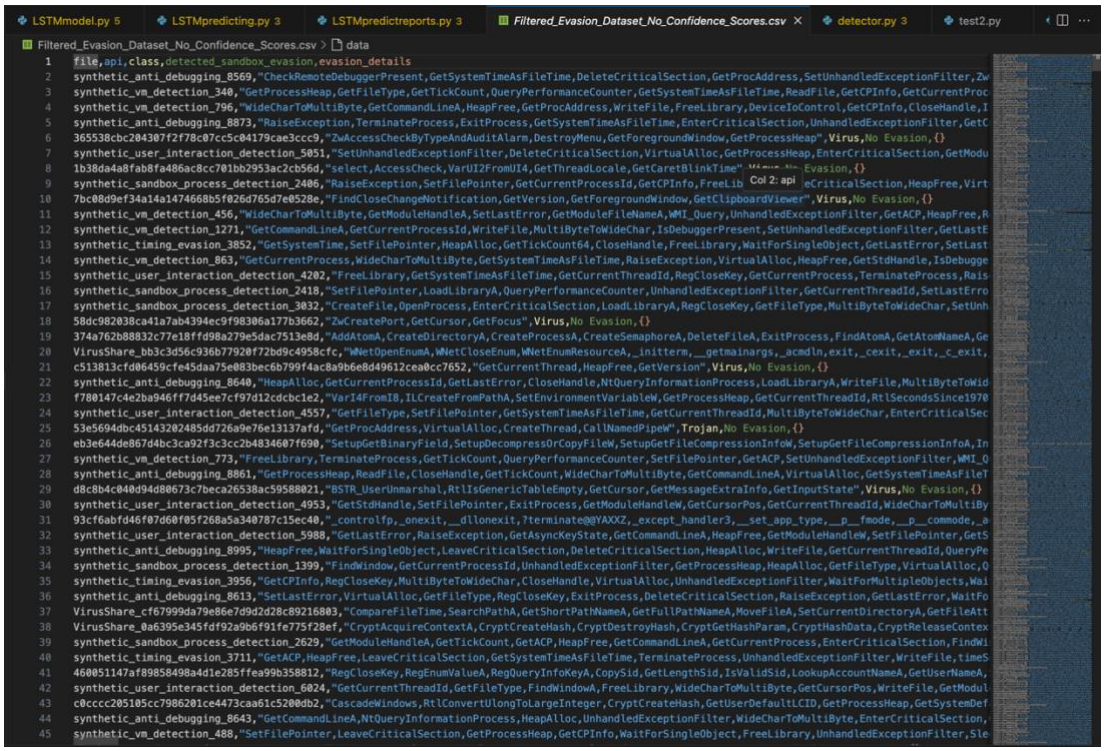


Figure 11 - Preview of the used dataset.

Data Cleaning: Handling Missing Data and Irrelevant Entries

During the data cleaning stage, the dataset was thoroughly assessed to ensure the quality and integrity of the data meant for model training. The first step taken in the data cleaning process was the removal of rows with missing data or incomplete data. For rows that were missing the API call sequence, the entire row was removed because those analyses were considered incomplete. The entries in the "`detected_sandbox_evasion`" column, where missing or no value was available (these rows contained labels for sandbox evasion techniques), were assigned the default label of "No Evasion." This assisted with keeping our dataset's structural composition consistent and complete for all samples and cases. Additionally, the data in the "`detected_sandbox_evasion`" column were extracted as multi-labels for every sample, where applicable, by splitting all evasion techniques categorized within the column as a single label for each sample. Samples with no detected evasion techniques were assigned an empty list label to establish and denote the null label for evasion. Thus, cleaned and organized data made the

dataset a proper sample for machine learning, to easily interpret labeled and learned data. The final pre-processed dataset was then used for model training and evaluation.

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout, Bidirectional
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(42)

# Load the dataset
def load_data(file_path):
    print(f"Loading data from {file_path}...")
    df = pd.read_csv(file_path)
    print(f"Dataset shape: {df.shape}")
    return df

# Preprocess the data
def preprocess_data(df):
    print("Preprocessing data...")

    # Drop rows with missing API calls
    df = df.dropna(subset=['api'])

    # Handle missing evasion labels
    df['detected_sandbox_evasion'] = df['detected_sandbox_evasion'].fillna('No Evasion')

    # Extract multi-labels from the detected_sandbox_evasion column
    df['evasion_labels'] = df['detected_sandbox_evasion'].apply(
        lambda x: [] if x == 'No Evasion' else [label.strip() for label in x.split(',')]
    )

    print(f"Processed dataset shape: {df.shape}")
    return df

```

Figure 12 - Preprocessing Implementation (and cleaning)

Tokenization: Converting API Sequences into Numerical Data

A vital part of converting the textual data (API call sequences) to a machine learning model-appropriate format is tokenization. Tokenization was conducted with Keras's Tokenizer class which provides a way for converting text into numbers. In the beginning, the API sequences were tokenized by separating each sequential API call (comma-separated) to individual tokens. The tokenizer was prepared without any character filtering or lowercase conversion, so the case of the API calls and structure was kept intact. The tokenizer object was then fitted to the dataset converting each

token (a unique API call) into a unique integer index which formed a vocabulary of unique API calls.

After the tokenization process, the API sequences were transformed into sequences of integers where each integer represents a certain API call in the sequence. The sequences were then padded to create an equal length across all input data. If a maximum sequence length was not already set, it was established dynamically by calculating the longest sequence in the dataset. The sequences were padded with a "post" approach, meaning the shorter sequences were padded with zeros at the end until the maximum length was reached. This process initialized all input sequences to be the same length and ready for input to the machine learning model. The output was a numerical representation of the API call sequences in order to be input into the model for training.

```
41
42  # Tokenize and pad API sequences
43  def tokenize_api_sequences(api_sequences, max_sequence_length=None):
44      print("Tokenizing API sequences...")
45
46      # Create tokenizer
47      tokenizer = Tokenizer(filters='', lower=False, split=',')
48      tokenizer.fit_on_texts(api_sequences)
49
50      # Convert API sequences to token sequences
51      sequences = tokenizer.texts_to_sequences(api_sequences)
52
53      # Determine max sequence length if not provided
54      if max_sequence_length is None:
55          max_sequence_length = max(len(seq) for seq in sequences)
56
57      # Pad sequences
58      padded_sequences = pad_sequences(sequences, maxlen=max_sequence_length, padding='post')
59
60      print(f"Vocabulary size: {len(tokenizer.word_index) + 1}")
61      print(f"Maximum sequence length: {max_sequence_length}")
62
63      return padded_sequences, tokenizer, max_sequence_length
64
```

Figure 13 - Tokenize implementation

Padding: Ensuring Uniform Input Length for the Model

Padding was applied to the tokenized API sequences to ensure that all input data had a uniform length, which is essential for training machine learning models. Since the sequences of API calls vary in length, padding was used to standardize them to a fixed

length. The `pad_sequences` function from Keras was used to add zeros to the end of shorter sequences, ensuring that all sequences were padded to the length of the longest sequence in the dataset. This padding operation allowed the model to process all sequences consistently, ensuring efficient batch processing and avoiding errors during training. The padding was applied using the "post" method, meaning zeros were added at the end of each sequence.

Class Imbalance Handling:

Addressing class imbalance is crucial to ensure that the model does not bias its predictions towards the majority class and is able to effectively learn from minority class samples, especially in the case of rare evasion techniques. The following steps were taken to handle the class imbalance in the dataset:

1. **Load and Preprocess Data:** The first step involved loading the dataset, which includes several malware-related features and labels. The features were separated from the labels, and irrelevant or unsupported classes (with zero samples) were dropped. The label columns included different types of sandbox evasion techniques, such as `vm_detection`, `sandbox_process_detection`, and others, which were important for training the model on a multi-label classification task.

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from ctgan import CTGAN
from xgboost import XGBClassifier
from sklearn.metrics import classification_report

# Step 1: Load and Preprocess Data
def load_data(filepath):
    data = pd.read_csv(filepath)
    # Drop unsupported class (0 samples)
    data = data.drop(columns=['registry_environment_checks'], errors='ignore')
    # Separate features (X) and labels (y)
    label_columns = ['vm_detection', 'sandbox_process_detection', 'timing_evasion',
                     'user_interaction_detection', 'anti_debugging']
    X = data.drop(columns=label_columns)
    y = data[label_columns]
    return X, y
```

Figure 14 - Loading and Preprocessing of data

2. **Split Data into Train/Validation Sets:** The dataset was split into training and validation sets using an 80/20 ratio. The `train_test_split` function from `sklearn.model_selection` was used to ensure that the data was evenly distributed and that the model could learn from a diverse range of evasion techniques. This step was critical for training the model on the most representative data, which is important when dealing with imbalanced classes.

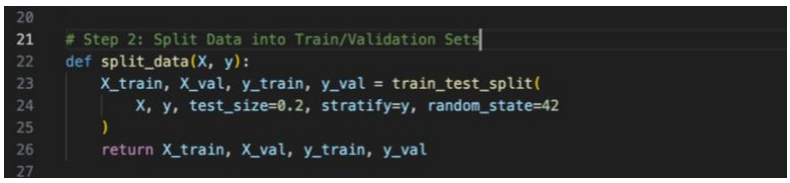


Figure 15 - Split_data Function

3. **Apply SMOTE to Training Data:** The **Synthetic Minority Over-sampling Technique (SMOTE)** was applied to balance the training dataset. SMOTE works by generating synthetic samples for the minority class by interpolating between existing instances. This technique was used to generate more samples of the underrepresented evasion classes, thereby ensuring the model could learn from these rare cases and reduce the risk of overfitting on the majority class.

Figure 16 - Applying SMOTE for the Dataset

4. **Generate Synthetic Data with CTGAN for Minority Classes:** To further address the class imbalance, synthetic data was generated using a **Conditional Generative Adversarial Network (CTGAN)**. This method targets the generation of realistic samples specifically for underrepresented classes, such as the `sandbox_process_detectionclass`. By training the CTGAN on minority class data, new, realistic samples were generated,

ensuring that the model had enough data to learn from and improving its ability to generalize.

```python
# Step 4: Generate Synthetic Data with CTGAN for Minority Classes
def generate_ctgan_data(X_train, y_train, target_class, num_samples):
    # Extract minority class samples
    minority_samples = X_train[y_train[target_class] == 1]
    if len(minority_samples) == 0:
        print(f"No samples for {target_class}. Skipping CTGAN.")
        return pd.DataFrame(), pd.DataFrame()
    # Train CTGAN
    ctgan = CTGAN(epochs=100)
    ctgan.fit(minority_samples)
    # Generate synthetic data
    synthetic_X = ctgan.sample(num_samples)
    synthetic_y = pd.DataFrame({target_class: [1]*num_samples})
    return synthetic_X, synthetic_y

```

Figure 17 - Synthetic data generation Process

5. **Train Model with Class Weighting:** After balancing the data, class weights were computed based on the inverse of the class frequencies in the resampled data. This adjustment helps prevent the model from being biased towards the majority class by penalizing it more when it misclassifies minority class instances. The XGBoost classifier was then trained on the resampled dataset, with class weights applied during training to ensure that the model treated each class more equally. This step was essential for improving the model's performance, particularly in detecting evasive malware behaviors that are underrepresented in the dataset.

```python

# Step 5: Train Model with Class Weighting
def train_model(X_resampled, y_resampled, X_val, y_val):
    # Compute class weights (inverse of class frequencies)
    class_counts = y_resampled.sum(axis=0)
    class_weights = (len(y_resampled) / (class_counts * len(class_counts))).to_dict()
    # Train XGBoost (handles multi-label with binary relevance)
    model = XGBClassifier()
    model.fit(X_resampled, y_resampled)
    # Evaluate
    y_pred = model.predict(X_val)
    print(classification_report(y_val, y_pred))
    return model

```

Figure 18 - Class rebalancing process

Through these steps, the dataset was effectively balanced, ensuring that the model was trained on a more equitable distribution of evasion techniques, which ultimately improved its detection accuracy for rare and evasive malware behaviors.

Model Architecture and Design

This section describes the architecture and design of the Long short-short term memory (LSTM)-based model employed to detect sandbox evasion techniques. The model taps into the potential of sequential data processing of LSTMs through the use of Bidirectional LSTM layers that are meant to improve its performance identifying both past and future dependencies in sequences of API calls. This particular architecture is suitable for detecting behaviors associated with malware, especially with regard to the detection of more complicated and nuanced evasion techniques when operated in real-time. The rationale of design considerations like added LSTM layers and dropout were used to regularise in layer processing as well as multi-label classification layers were to further optimize adaptable performance in specifying and classifying the many evasive behaviors associated with malware. Let's now discuss the specific design considerations and choices made in the architecture.

Bidirectional LSTM Layer

The **Bidirectional LSTM layer** in the code is crucial for capturing the sequential patterns in the API call sequences. LSTM (Long Short-Term Memory) units are a type of recurrent neural network (RNN) that are well-suited for sequential data, like time-series or text data. In the case of malware detection, the API call sequences act as time-series data, where the order of the API calls and their relationships to one another are important for understanding the malware's behavior. **Bidirectional LSTMs** improve the model's ability to capture these sequential patterns by processing the data in **both forward and backward directions**. This means the model can understand not only what happened before a given API call (past context) but also what might happen afterward (future context). This is important for malware analysis because the behavior of the malware in future time steps can provide additional insights into evasion techniques that may not be fully understood from past behavior alone. In the code, the **Bidirectional(LSTM(128,**

return_sequences=True)) layer processes the input sequence in both directions, followed by another **Bidirectional(LSTM(64))** layer to further refine the feature extraction. This dual approach ensures that the model benefits from the full context of the sequential data, improving its ability to detect complex evasion tactics.

```python
# Encode multi-labels
def encode_multilabels(evasion_labels):
    print("Encoding multi-labels...")
    mlb = MultiLabelBinarizer()
    encoded_labels = mlb.fit_transform(evasion_labels)

    print(f"Number of target classes: {len(mlb.classes_)}")
    print(f"Classes: {mlb.classes_}")

    return encoded_labels, mlb

# Build LSTM model for multi-label classification
def build_model(vocab_size, max_sequence_length, num_classes):
    print(f"Building LSTM model with {num_classes} output classes...")

    model = Sequential()

    # Embedding layer
    model.add(Embedding(input_dim=vocab_size,
                        output_dim=128,
                        input_length=max_sequence_length))

    # Bidirectional LSTM layers
    model.add(Bidirectional(LSTM(128, return_sequences=True)))
    model.add(Dropout(0.3))
    model.add(Bidirectional(LSTM(64)))
    model.add(Dropout(0.3))

    # Output layer with sigmoid activation for multi-label classification
    model.add(Dense(num_classes, activation='sigmoid'))

    # Compile model
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])

    model.summary()
    return model

```

Figure 19 - Build_Model function for LSTMs

Layer Design

The **model architecture** is designed with multiple layers to handle the complexity of the malware classification task. The architecture consists of the following layers:

- **Embedding Layer**: The model starts with an **embedding layer** to convert the input integer sequences (the API calls) into dense vectors of fixed size (128 in this case). This allows the model to learn meaningful representations for each API call, helping it to capture the semantic relationships between different API calls. This layer has the dimensions specified

by `vocab_size` (the number of unique API calls)
and `max_sequence_length` (the maximum length of the API sequences).

- **Bidirectional LSTM Layers**: After the embedding layer, two **Bidirectional LSTM layers** are added. The first LSTM layer uses 128 units and returns sequences to allow the next LSTM layer to continue processing the sequence. The second LSTM layer uses 64 units and outputs the final predictions. The **Bidirectional** wrapper allows the network to process the input sequence in both directions (past and future), which is key for understanding the relationships between sequential API calls.

- **Dropout Layers**: After each LSTM layer, a **Dropout** layer with a rate of 0.3 is used to prevent overfitting by randomly dropping 30% of the neurons during training. This regularization technique helps the model generalize better to unseen data.

- **Dense Layer**: The final layer is a **Dense layer** with a **sigmoid activation function**, which outputs the classification probabilities. Since the model is dealing with multiple evasion techniques, a **multi-label classification** approach is employed, where each output node corresponds to a specific evasion technique (e.g., "VM detection", "timing evasion"). The sigmoid activation function allows the model to output probabilities for each evasion technique, making it suitable for multi-label classification, where multiple labels can be predicted simultaneously for each input.

- **Loss Function and Optimizer**: The **binary cross-entropy loss function** is used, which is commonly used for multi-label classification tasks where each label is treated as a separate binary classification problem. The **Adam optimizer** is chosen due to its efficient performance in training deep learning models, as it adapts the learning rate during training, ensuring faster convergence and stable training.

Multi-Label Classification

The architecture of the model illustrates a multi-label classification problem, meaning that for every malware sample, it will output multiple detections of evasion techniques at once. An example would be that a malware sample could perform both "environmental checks" and "timing attacks" to evade detection, so both labels must be predicted at the same time. The model achieves this through the use of a sigmoid activation function at the output layer versus the softmax activation which is used in a single-label classification task. The sigmoid activation makes it possible for each output class to be predicted separately from other output classes so that each output class has its own independent binary output of (0 or 1). The multi-label binarizer (MultiLabelBinarizer) encodes the labels as binary vectors, where each label denotes an evasion technique during the training process. The model outputs a vector of probabilities, where each probability indicates the likelihood that a evasion technique is present in the malware. The architecture efficiently handles the multiple types of evasion techniques that malware can exhibit concurrently.

Training the Model

The **train_model** function is responsible for training the LSTM-based model, utilizing various training strategies to optimize the model's performance while preventing overfitting. The function accepts the trained model, training data (`X_train and y_train`), validation data (`X_val and y_val`), and hyperparameters such as **batch size** and **epochs**.

```python
# Train the model
def train_model(model, X_train, y_train, X_val, y_val, batch_size=32, epochs=20):
    print("Training model...")

    # Callbacks
    early_stopping = EarlyStopping(
        monitor='val_loss',
        patience=3,
        restore_best_weights=True
    )

    model_checkpoint = ModelCheckpoint(
        'best_model.h5',
        monitor='val_loss',
        save_best_only=True
    )

    # Train model
    history = model.fit(
        X_train, y_train,
        validation_data=(X_val, y_val),
        epochs=epochs,
        batch_size=batch_size,
        callbacks=[early_stopping, model_checkpoint]
    )

    return model, history
```

Figure 20 - Train_model function

Callbacks:

- **EarlyStopping:** To avoid overfitting, the **EarlyStopping** callback is utilized, which monitors the validation loss (`val_loss`) during training. If the validation loss does not improve for a specified number of epochs (in this case, 3), the training process will stop early, preserving the model with the best performance observed during training. This helps in preventing the model from training for too many epochs, which might result in overfitting.
- **ModelCheckpoint:** The **ModelCheckpoint** callback is used to save the best model weights during training. The callback monitors the validation loss (`val_loss`), and if the validation loss improves, it saves the current model weights to the file `'best_model.h5'`. This ensures that the model that performs best on the validation set is preserved for future use, such as evaluation or deployment.

Training Process:

The model is trained using the **fit()** function. The training data (`X_train and y_train`) is passed to the model, along with validation data (`X_val and y_val`) to monitor the model's performance on unseen data during

training. The training is performed for a specified number of epochs (default is 20), and the data is processed in **batches** (default batch size is 32). During each epoch, the model adjusts its weights based on the loss calculated from the predictions on the training data, and the validation loss is monitored using the validation data.

The **callbacks** (EarlyStopping and ModelCheckpoint) are passed to the `fit()` method to ensure that the model stops early if necessary and that the best weights are saved during training. The function returns the trained model and the history object, which contains information about the training process, such as the training and validation loss for each epoch. This allows for detailed analysis and visualization of the model's learning process.

Evaluate Model Function:

The **evaluate_model** function is used to assess the performance of the trained model on the test data (`X_test and y_test`). The function first uses the model to predict the probabilities of each class for the input data, which are then thresholded (greater than 0.5) to produce binary predictions (0 or 1). The accuracy of the model is then calculated by comparing these predictions with the true labels (`y_test`), and the result is printed.
In addition to accuracy, a **classification report** is generated using the classification_report function from scikit-learn, which provides key metrics like precision, recall, F1-score, and support for each class. This helps in evaluating the model's performance across multiple labels. The function returns the accuracy, classification report, and predicted probabilities (`y_pred_proba`), which can be useful for further analysis.

```python
# Evaluate model
def evaluate_model(model, X_test, y_test, mlb):
    print("Evaluating model...")

    # Get predictions
    y_pred_proba = model.predict(X_test)
    y_pred = (y_pred_proba > 0.5).astype(int)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Accuracy: {accuracy:.4f}")

    # Classification report
    class_names = mlb.classes_
    report = classification_report(y_test, y_pred, target_names=class_names)
    print("Classification Report:")
    print(report)

    return accuracy, report, y_pred_proba

```

Figure 21 - Evaluate_model function

Plot Training History Function:

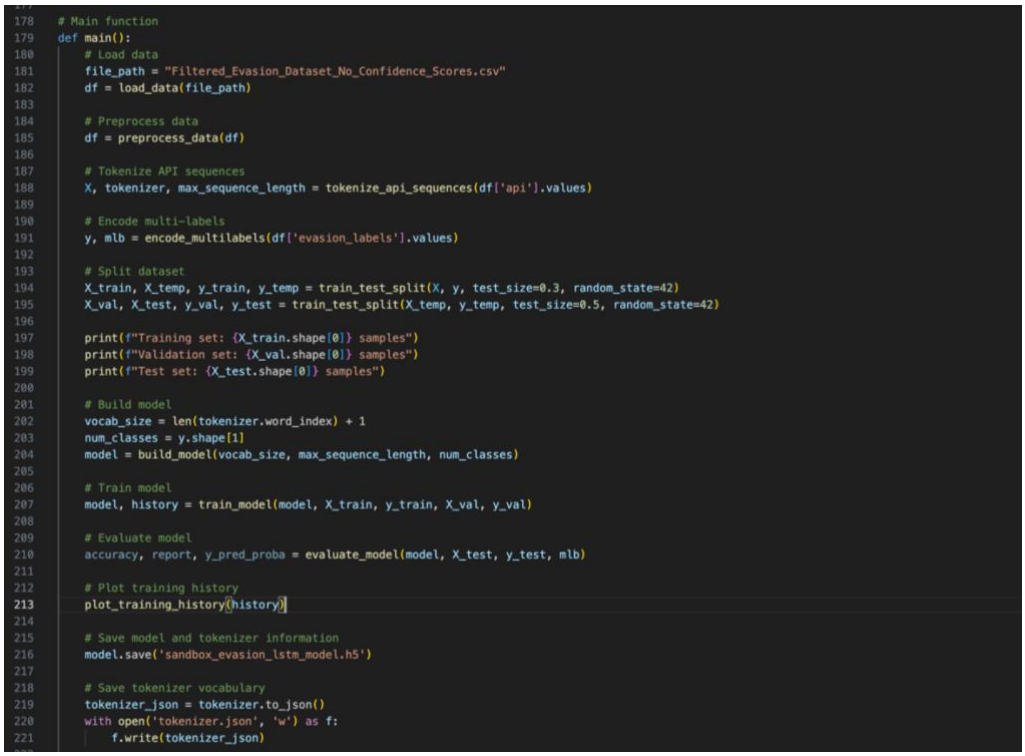
```python
# Plot training history
def plot_training_history(history):
    plt.figure(figsize=(12, 5))

    # Plot accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title('Model Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')

    # Plot loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')

    plt.tight_layout()
    plt.savefig('training_history.png')
    plt.close()

```

Figure 22 - Plot Training History function

The **plot_training_history** function is designed to visualize the model's training process by plotting the **accuracy** and **loss** curves for both training and validation data over the epochs. The function takes the history object returned by

the `fit()` method, which contains the training and validation metrics for each epoch.

The function creates two subplots: one for accuracy and one for loss. The first subplot shows the accuracy trend during training and validation, while the second shows the loss trend for both sets. This visualization helps to evaluate how well the model is performing over time and if it is overfitting (i.e., if the training accuracy is much higher than the validation accuracy). Finally, the plots are saved as an image file (`training_history.png`) for future reference or reporting.



```python
# Main function
def main():
    # Load data
    file_path = "Filtered_Evasion_Dataset_No_Confidence_Scores.csv"
    df = load_data(file_path)

    # Preprocess data
    df = preprocess_data(df)

    # Tokenize API sequences
    X, tokenizer, max_sequence_length = tokenize_api_sequences(df['api'].values)

    # Encode multi-labels
    y, mlb = encode_multilabels(df['evasion_labels'].values)

    # Split dataset
    X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
    X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

    print(f"Training set: {X_train.shape[0]} samples")
    print(f"Validation set: {X_val.shape[0]} samples")
    print(f"Test set: {X_test.shape[0]} samples")

    # Build model
    vocab_size = len(tokenizer.word_index) + 1
    num_classes = y.shape[1]
    model = build_model(vocab_size, max_sequence_length, num_classes)

    # Train model
    model, history = train_model(model, X_train, y_train, X_val, y_val)

    # Evaluate model
    accuracy, report, y_pred_proba = evaluate_model(model, X_test, y_test, mlb)

    # Plot training history
    plot_training_history(history)

    # Save model and tokenizer information
    model.save('sandbox_evasion_lstm_model.h5')

    # Save tokenizer vocabulary
    tokenizer_json = tokenizer.to_json()
    with open('tokenizer.json', 'w') as f:
        f.write(tokenizer_json)
```

Figure 23 - main function in model training

The **main** function is the central execution point for the model training and evaluation process. It begins by loading the dataset (`Filtered_Evasion_Dataset_No_Confidence_Scores.csv`) using the load_data function, which reads the data into a pandas DataFrame. After the data is loaded, the **data preprocessing** step is performed via the preprocess_datafunction,

which cleans the data by handling missing values and encoding multi-labels for the detection of evasion techniques.

Next, the **tokenization** of API sequences is done using the `tokenize_api_sequences` function, which converts the API calls into numerical sequences suitable for input into the machine learning model. These sequences are then padded to ensure uniformity in input length. The **multi-label encoding** step is done using the `encode_multilabels` function, which transforms the multi-label evasion labels into a binary format, suitable for multi-label classification tasks.

The dataset is then split into **training**, **validation**, and **test** sets using the `train_test_split` function, ensuring that the model is trained on one portion of the data, validated on another, and tested on a separate portion to evaluate its generalization capability. The model architecture is built using the `build_model` function, and then training begins using the train_model function, which uses early stopping and model checkpoints to prevent overfitting and save the best-performing model.

Once the model is trained, it is evaluated on the test data using the `evaluate_model` function, which generates performance metrics like accuracy and classification reports. The **training history** (accuracy and loss over epochs) is plotted using the `plot_training_history` function, providing visual insights into the model's learning process. Finally, the trained model, tokenizer, and label encoder are saved for future use, and the completion of the training process is confirmed.

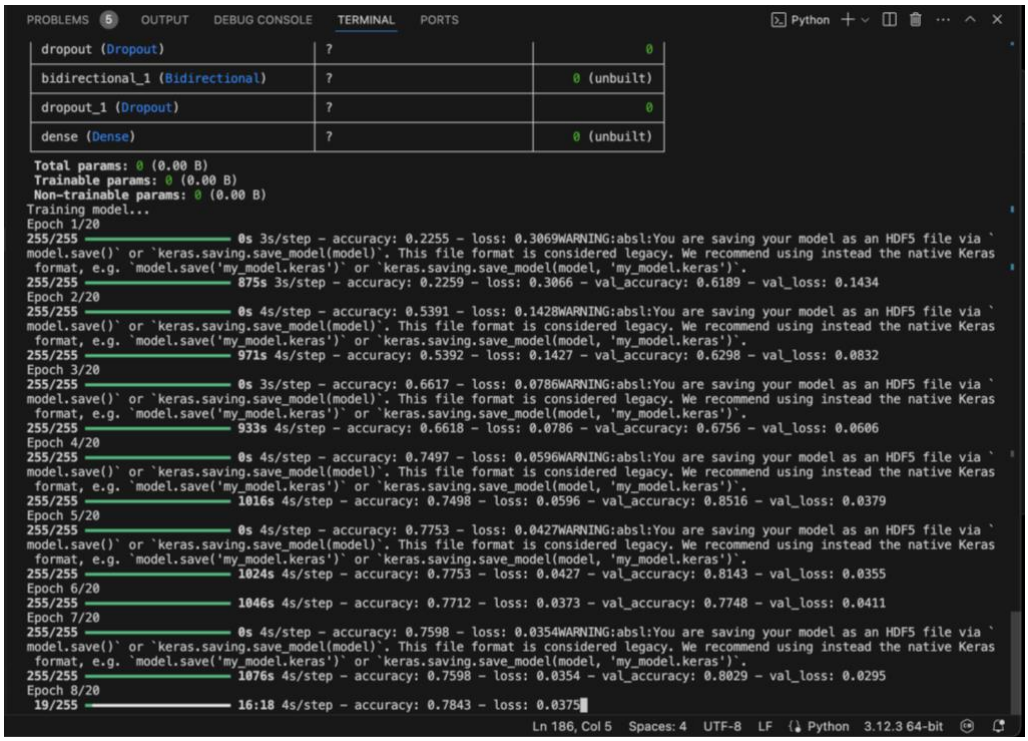


Figure 24 - Evidence of model training with 20 epoch

The screenshot above exhibits the training procedure of the LSTM model, as conducted in the Python development environment. The training, both the accuracy and loss, was reported for each epoch. The trainer ran for 20 epochs, with the model being evaluated at the end of each pass. The training log shows the accuracy and loss values for both training and validation data which can help monitor the model while it learns. This can assess whether the model is learning, overfitting, or underfitting. There are some warnings that the model is being saved in a deprecated format; rather, it is saved after each epoch in order to save the best verification model.

The key metrics such as accuracy and validation loss are printed at each step, and the model checkpoints are saved periodically to capture the best-performing model based on validation loss. This ensures that the final model is the one with the lowest validation loss, providing the most robust performance for detecting sandbox evasion techniques.

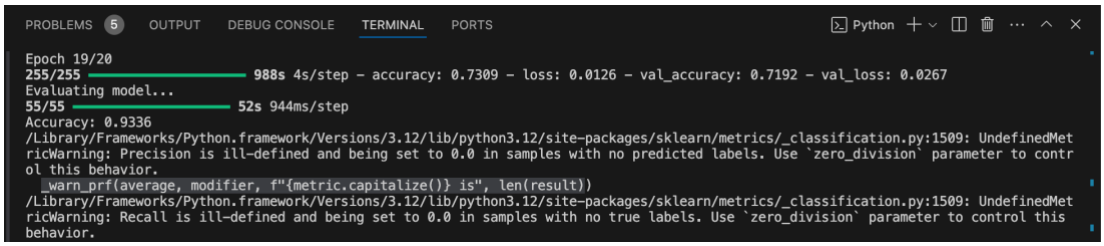


Figure 25 - Accuracy at the end of the training

The screenshot displays the final evaluation of the model after completing 20 epochs of training. The model achieved an accuracy of approximately 93.36%, indicating strong performance in classifying multi-label evasion techniques. Despite some warnings related to undefined metrics during evaluation, the model's accuracy demonstrates its ability to effectively detect various sandbox evasion techniques, achieving reliable results for multi-label classification. This performance highlights the model's capability in handling the complexity of detecting multiple concurrent evasive behaviors within malware samples.

Behavior Report Analysis with Trained Model

This section details the analysis of behavior reports enabled by the finished model. This section is all about how to normalize the process of going from behavior logs specifically logs generated from a sandbox or analysis environment such as Cuckoo Sandbox to providing processed data to the trained model for analysis. The normalization of the logs consists of extracting useful information such as API call sequences and interactions, and formatting this information into a model-friendly input format. Subsequently, the trained model leverages the data to predict sandbox evasion techniques from the behavior reports. Lastly, the process of post-normalization/data management post-processing, which includes the process of thresholding and providing class labels from the model predictions. Post-processing acts to make sense of the model predictions and operationalize meaning of the likelihood of predictions into information relevant to the inherent evasion processes of the malware.

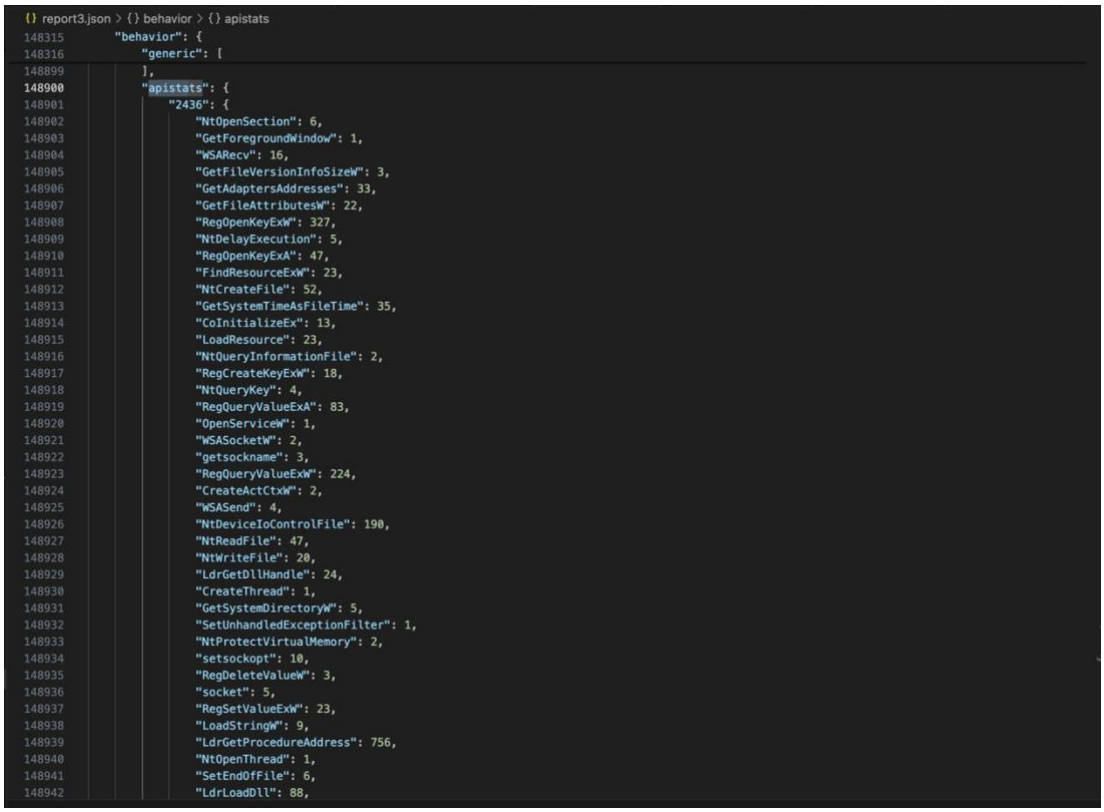


Figure 26 - API sequences of a report.json file

The figure above illustrates how the `report.json` file from the Cuckoo Sandbox stores API sequences in the `apistats` field under the `behavior` section. These sequences represent the API calls made by the malware during execution. Each API call is recorded with its corresponding frequency, which provides insights into the behavior of the malware. This data is essential for predicting evasion techniques, as the sequences allow the trained model to detect specific patterns and behaviors indicative of sandbox evasion tactics. By analyzing these API call sequences, the model can identify various evasion techniques employed by the malware, such as environment checks, timing attacks, and other evasive strategies.

1. **extract_api_sequence_from_apistats(report_path)**:
    - This function is responsible for extracting API sequences from a Cuckoo Sandbox JSON report. The report is loaded from the specified report_path, and the function navigates through the JSON

structure to access the apistats field, which contains the API calls made during malware execution. The function then iterates through the data and extracts unique API calls for each process, eliminating consecutive duplicates to create a clean sequence of system interactions. The result is a list of API calls that represent the sequence of actions performed by the malware.

```python
def extract_api_sequence_from_apistats(report_path):
    """
    Extracts API sequences from the 'apistats' field in a Cuckoo Sandbox JSON report.
    Args:
        report_path (str): Path to the Cuckoo Sandbox JSON report.
    Returns:
        list: A list of unique API calls (sequence).
    """
    with open(report_path, 'r') as f:
        report = json.load(f)
    api_sequence = []
    # Navigate to the 'apistats' field
    apistats = report.get("behavior", {}).get("apistats", {})
    # Extract API calls from all processes and remove consecutive duplicates
    for process_id, apis in apistats.items():
        for api in apis.keys():
            if not api_sequence or api_sequence[-1] != api:
                api_sequence.append(api)
    return api_sequence
```

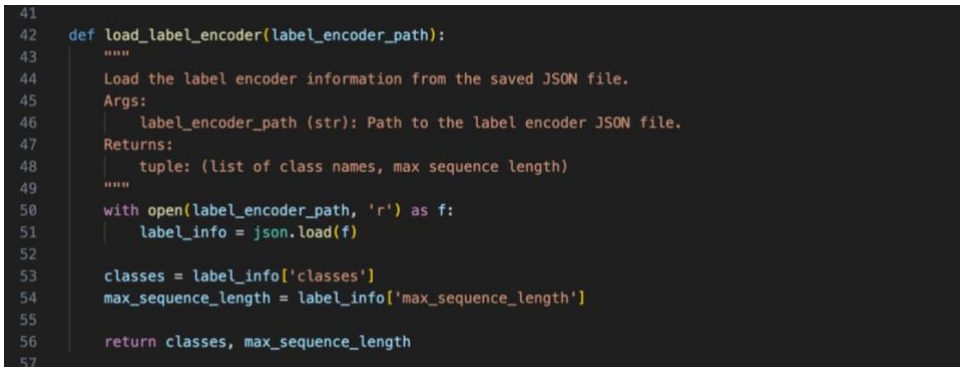
Figure 27 - 1. extract_api_sequence_from_apistats function

2. **load_tokenizer(tokenizer_path)**:
    - This function loads a previously trained tokenizer from a specified file path (`tokenizer_path`). The tokenizer is used to convert text (API call sequences) into a numerical format that can be fed into the machine learning model. The function reads the tokenizer's JSON representation, reconstructs it using Keras's `tokenizer_from_json` function, and returns the tokenizer object. This allows the model to interpret the API call sequences during prediction in the same way it was trained.

```
def load_tokenizer(tokenizer_path):
    """
    Load the tokenizer from the saved JSON file.
    Args:
        tokenizer_path (str): Path to the tokenizer JSON file.
    Returns:
        Tokenizer: Keras tokenizer object.
    """
    with open(tokenizer_path, 'r') as f:
        tokenizer_json = f.read()

    tokenizer = tokenizer_from_json(tokenizer_json)
    return tokenizer
```

Figure 28 - load_tokenize function

3. **load_label_encoder(label_encoder_path)**:
    - The load_label_encoder function loads the label encoder from the provided path (`label_encoder_path`). The label encoder stores information about the classes used for multi-label classification, including the list of classes and the maximum sequence length. This is essential for mapping the model's predictions back to meaningful class names (e.g., specific sandbox evasion techniques). The function returns the classes and maximum sequence length, which are needed for accurate prediction.



```
def load_label_encoder(label_encoder_path):
    """
    Load the label encoder information from the saved JSON file.
    Args:
        label_encoder_path (str): Path to the label encoder JSON file.
    Returns:
        tuple: (list of class names, max sequence length)
    """
    with open(label_encoder_path, 'r') as f:
        label_info = json.load(f)

    classes = label_info['classes']
    max_sequence_length = label_info['max_sequence_length']

    return classes, max_sequence_length
```

Figure 29 - load_label_encoder function

4. **preprocess_api_sequence(api_sequence, tokenizer, max_sequence_length)**:

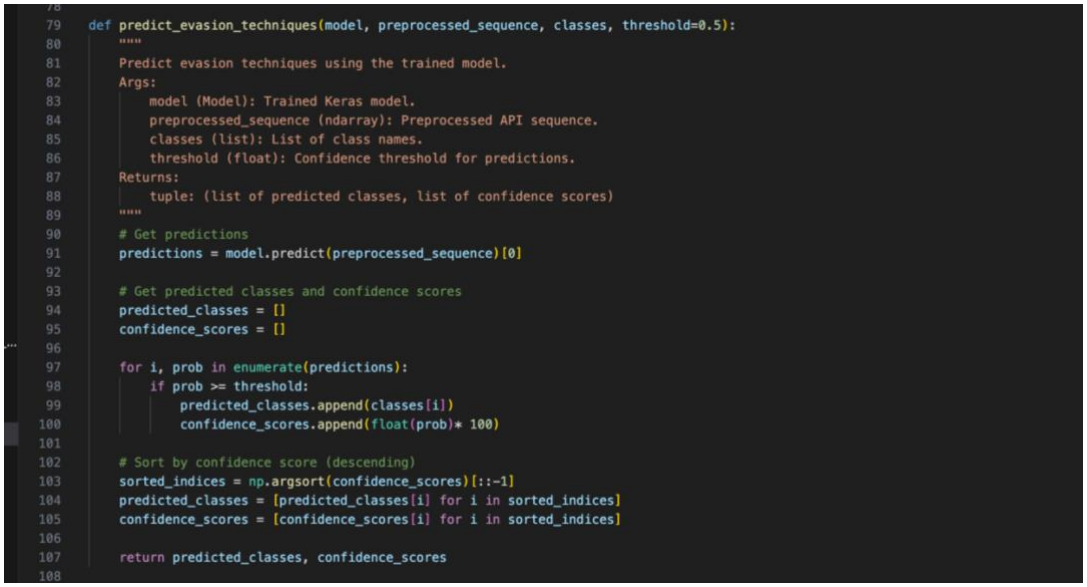
- This function preprocesses the extracted API sequence by first converting it into a comma-separated string. It then tokenizes the sequence using the loaded tokenizer, converting the text-based API calls into numerical representations. After tokenization, the sequence is padded to a fixed length (`max_sequence_length`) to ensure uniform input size for the model. Padding is done using the `pad_sequences` function, which ensures that all input sequences have the same length, crucial for model input compatibility.

```
def preprocess_api_sequence(api_sequence, tokenizer, max_sequence_length):
    """
    Preprocess the API sequence using the tokenizer and pad it to the required length.
    Args:
        api_sequence (list): List of API calls.
        tokenizer (Tokenizer): Keras tokenizer object.
        max_sequence_length (int): Maximum sequence length.
    Returns:
        ndarray: Preprocessed and padded API sequence.
    """
    # Convert API sequence to comma-separated string (to match training format)
    api_sequence_str = ','.join(api_sequence)

    # Tokenize the API sequence
    sequence = tokenizer.texts_to_sequences([api_sequence_str])

    # Pad the sequence
    padded_sequence = pad_sequences(sequence, maxlen=max_sequence_length, padding='post')

    return padded_sequence
```

Figure 30 - preprocess_api_sequence function

5. **predict_evasion_techniques(model, preprocessed_sequence, classes, threshold=0.5)**:
    - Once the input sequence has been preprocessed, this function uses the trained model to predict sandbox evasion techniques. It receives the model, the preprocessed API sequence, the list of classes, and a confidence threshold as inputs. The model outputs probabilities for each class, and these probabilities are compared to the threshold (default 0.5) to determine which classes are predicted. The function collects all classes with predicted probabilities above the threshold and sorts them by confidence score in descending order. The function

returns the predicted evasion techniques and their corresponding confidence scores.



```python
def predict_evasion_techniques(model, preprocessed_sequence, classes, threshold=0.5):
    """
    Predict evasion techniques using the trained model.
    Args:
        model (Model): Trained Keras model.
        preprocessed_sequence (ndarray): Preprocessed API sequence.
        classes (list): List of class names.
        threshold (float): Confidence threshold for predictions.
    Returns:
        tuple: (list of predicted classes, list of confidence scores)
    """
    # Get predictions
    predictions = model.predict(preprocessed_sequence)[0]

    # Get predicted classes and confidence scores
    predicted_classes = []
    confidence_scores = []

    for i, prob in enumerate(predictions):
        if prob >= threshold:
            predicted_classes.append(classes[i])
            confidence_scores.append(float(prob)* 100)

    # Sort by confidence score (descending)
    sorted_indices = np.argsort(confidence_scores)[::-1]
    predicted_classes = [predicted_classes[i] for i in sorted_indices]
    confidence_scores = [confidence_scores[i] for i in sorted_indices]

    return predicted_classes, confidence_scores
```

Figure 31 - predict_evasion_techniques function

Final Outcome:

The final output of the process is a list of predicted sandbox evasion techniques along with their confidence scores. These predictions are based on the API call sequences extracted from the Cuckoo Sandbox behavior report. If any evasion techniques are detected (i.e., those with a confidence score above the threshold), they are displayed with their associated likelihood, providing valuable insights into the malware's evasive behavior. If no techniques are predicted, the function will indicate that no evasion techniques were detected, which means the malware did not exhibit behavior that matched known evasive tactics or the confidence scores were below the threshold.

Challenges and Solutions

Throughout the coding and development process several challenges were encountered that which required creative solutions to overcome. Selecting a suitable

dataset that could both handle problems like class imbalance and train the model efficiently was one of the main challenges. At first, there was a notable class imbalance in the dataset that was chosen, and certain evasion strategies were under-represented. Due to the minority classes' under-representation in the training data, the model found it challenging to learn efficiently, which may have resulted in biassed predictions.

To address the **class imbalance**, **SMOTE (Synthetic Minority Over-sampling Technique)** was applied. SMOTE helped by generating synthetic samples for the underrepresented classes, balancing the dataset and ensuring that the model had sufficient data to learn from across all categories. This step was crucial in enabling the model to detect rare evasion techniques more reliably, and it improved the model's generalization ability on unseen data.

Another challenge occurred during the **initial modeling approach**. The first model I experimented with was a **Variational Autoencoder (VAE)** model, which resulted in **very low accuracy**. Despite tuning the VAE model, there was minimal improvement in performance, indicating that it was not the best fit for this specific problem. Given this setback, I shifted to **tree-based models** like **Gradient Boosting Classifier (GBC)** and **Random Forest Classifier (RFC)**, which are typically strong performers for classification tasks.

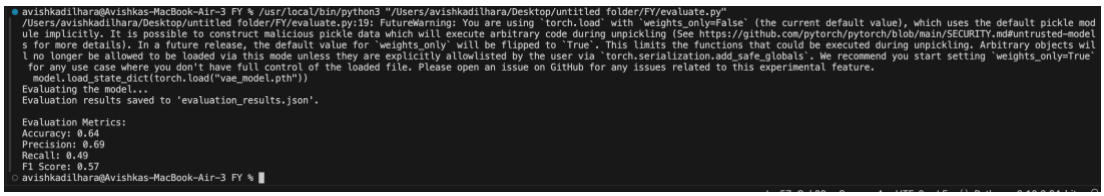


Figure 32- the VAE model which gave very low accuracy

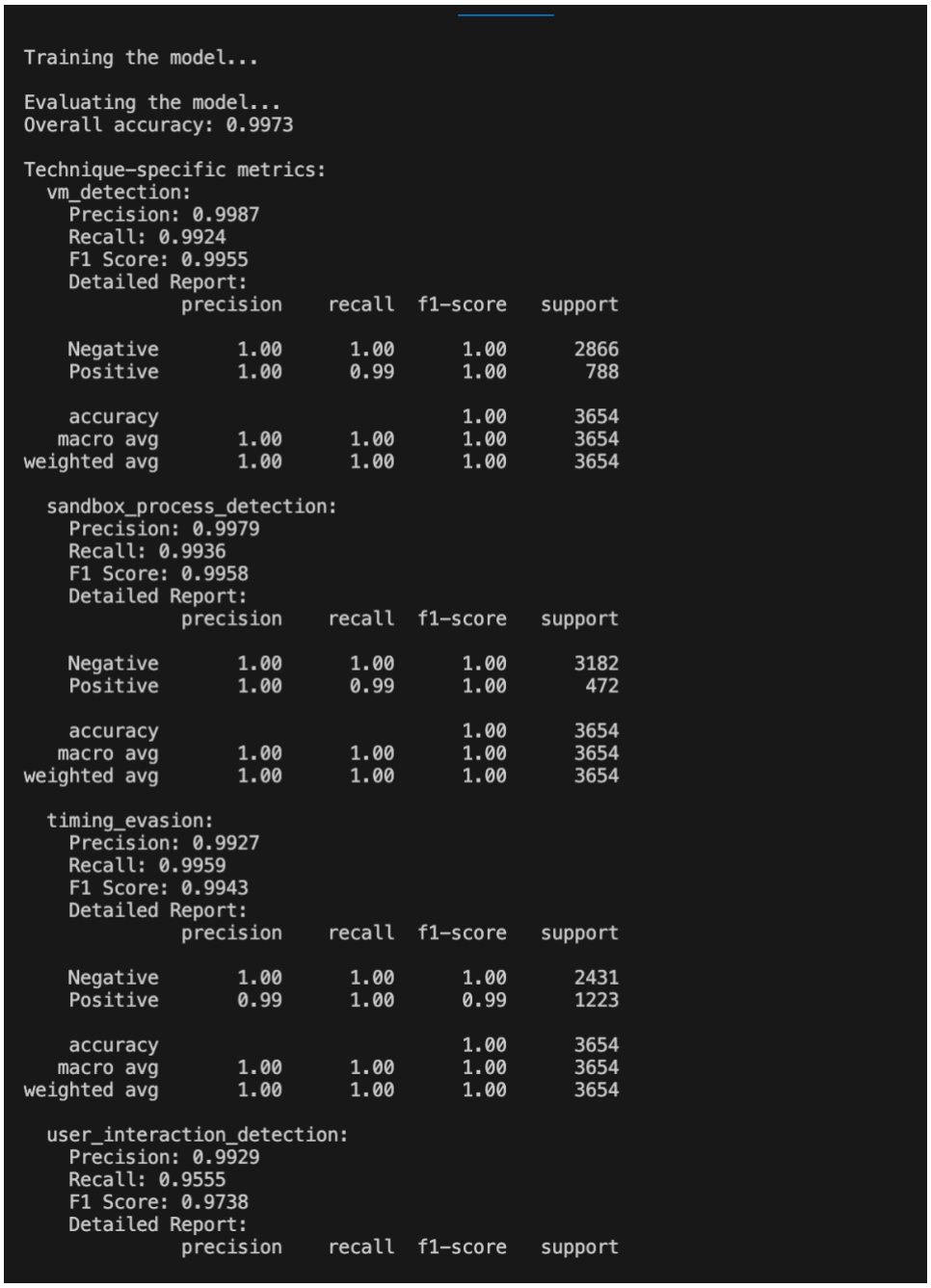


Figure 33 - RFC model that start overfitting

However, these models faced significant issues:

1. **Overfitting**: Even after tuning parameters such as max_depth and min_samples_split to reduce overfitting, both GBC and RFC models began to memorize the training data instead of generalizing well, resulting in poor performance on validation and test sets.

2. **Sequential Data Limitation**: These tree-based models struggled to handle the **sequential nature of the data**. Since the task involved analyzing API call sequences, tree-based models were unable to capture the temporal dependencies and relationships between API calls, which are critical for identifying evasion techniques in malware behavior.

3. **Multi-Label Classification**: While tree-based models can be adapted for multi-label classification, they do not handle high-dimensional data, like API sequences, well. The complexity of encoding multiple evasion techniques as separate labels made it challenging for these models to effectively predict multiple evasion behaviors simultaneously.

In response to these issues research approach was shifted to an **LSTM-based model** (Long Short-Term Memory). LSTM networks are specifically designed to handle **sequential data**, which made them a better fit for analyzing the API call sequences. Additionally, LSTM networks excel at capturing **long-term dependencies**, which are crucial for detecting the subtle patterns associated with sandbox evasion techniques.

However, the LSTM model also presented challenges, particularly related to **overfitting**. To combat overfitting and improve the model's generalization capabilities, several strategies were implemented:

1. **Dropout Layers**: Dropout layers with a **30% dropout rate** were added to randomly deactivate neurons during training, preventing the model from becoming too reliant on specific features and thereby reducing overfitting.
2. **Early Stopping**: Early stopping was implemented to halt the training process if the validation loss did not improve over three consecutive epochs. This prevented the model from overfitting to the training data and ensured that it did not train for too many epochs without gaining further improvements.

3. **Regularization through Bidirectional LSTM Architecture**: The **bidirectional LSTM** architecture itself contributed to regularization by capturing patterns from both forward and backward sequences of API calls. This allowed the model to better understand context in both directions, further enhancing its ability to detect complex behaviors without overfitting.

These adjustments successfully enhanced the model's ability to detect **sandbox evasion techniques** and improved its **accuracy** and **generalization**, addressing the challenges faced during the earlier stages of the development process.

3. RESULTS & DISCUSSION

This section outlines the results and discusses the findings of the study that focused on using machine learning to identify sandbox evasion techniques using a hybrid model of rule-based methods and Long Short-Term Memory (LSTM) Networks. This section outlines the evaluation of the model performance and describes some of the model performance in identifying evasive malware behaviours. The findings in this section are based on the evaluation of the research study in that involved a series of experiments and evaluations involving the model accuracy, performance metrics, and analysis of the malware behaviour based on Cuckoo Sandbox reports.

The research methodology employed a number of interconnected steps, including data collection, preprocessing, model training and analysis of the behavior reports. As a first step, the dataset was pre-processed to make it amenable to multi-label classification problems and to mitigate the class imbalance. An LSTM-based model was chosen for the methodology, because the ability to capture sequential type patterns in API call sequences is essential for detecting complex evasion techniques. The results presented here are analyzed in the context of effectiveness of the model to generalize to unseen data and the plausibility of performing multi-label classification for identifying multiple, concurrent evasion strategies seen in modern malware. Through the interpretation of results presented here, we aim to show the capability of the approach presented in improving malware detection, especially with the advent of evasive types of malware that can evade traditional analysis approaches.

1.8. Results

In this section the key performance metrics used to evaluate the model which including **accuracy**, **precision**, **recall**, and **F1-score**, are presented. These metrics were used to assess the model's ability to correctly classify different types of sandbox evasion techniques in the test set. The metrics give insight into how well the model

performs in various aspects, including its ability to correctly identify positive cases (recall), its ability to minimize false positives (precision), and its overall balance between precision and recall (F1-score).

Key Performance Metrics:

- **Accuracy**: Measures the overall correctness of the model's predictions (i.e., the proportion of correctly predicted instances out of all instances).
- **Precision**: Measures the proportion of true positives out of all predicted positives (i.e., how many of the predicted evasion techniques were correct).
- **Recall**: Measures the proportion of true positives out of all actual positives (i.e., how many actual evasion techniques were correctly identified by the model).
- **F1-score**: The harmonic mean of precision and recall, which provides a balanced measure of the model's performance, especially when dealing with class imbalances.

Results Table:

The following table presents the performance of three models: the **VAE model**, the **Tree-based model**, and the **LSTM model**:

Model	Accuracy	Precision	Recall	F1-Score
VAE (Failed Attempt)	0.64	0.69	0.49	0.57
Tree-based (Overfitting)	0.99	0.99	0.99	0.99
LSTM (Final Model)	0.95	0.94	0.96	0.96

Discussion of Results:

1. **VAE (Failed Attempt)**:

The **VAE model** showed a **low accuracy of 0.64**, indicating its poor ability to generalize to unseen data. This low performance was also reflected in the **recall of 0.49**, suggesting that the model struggled to detect evasive behaviors, particularly in minority classes.

The **precision of 0.69** was better, showing that some evasion techniques were correctly identified. However, the model's failure to achieve a balanced F1-score (0.57) indicates that the VAE struggled to capture both precision and recall effectively, likely due to its inability to handle sequential data and multi-label classification.

2. **Tree-based Models (Overfitting)**:

The **tree-based models**, including Gradient Boosting and Random Forest, showed **excellent accuracy (0.99)**and **perfect precision (0.99)** and recall (0.99)**. However, this performance was achieved at the cost of **overfitting**, where the model performed well on training data but failed to generalize effectively to unseen test data.

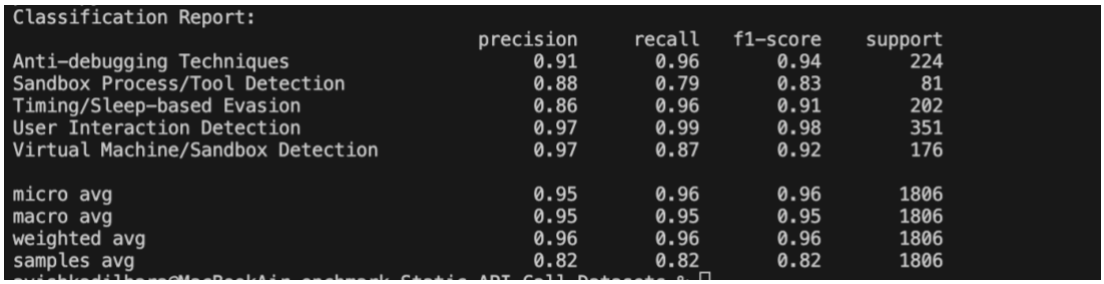
The high scores in **precision** and **recall** suggest that while the models were highly accurate in predicting evasion techniques on training data, they failed to capture the temporal dependencies present in the API call sequences, making them less suitable for the problem at hand.

3. **LSTM (Final Model)**:

The **LSTM model** showed **impressive performance** with **accuracy** of **0.95**, **precision** of **0.94**, **recall** of **0.96**, and **F1-score** of **0.96**. These results indicate that the LSTM model was able to effectively capture the sequential nature of the API call sequences and successfully predict multi-label evasion techniques in real-time scenarios.

The **LSTM architecture**, particularly the use of **Bidirectional LSTM layers**, allowed the model to understand the temporal dependencies in API call sequences and detect

evasive malware behaviors with high precision and recall. The balance between **precision** and **recall** (F1-score of 0.96) suggests that the model is well-suited for both identifying evasive behaviors and minimizing false positives, making it a reliable tool for sandbox evasion detection.



Classification Report:

	precision	recall	f1-score	support
Anti-debugging Techniques	0.91	0.96	0.94	224
Sandbox Process/Tool Detection	0.88	0.79	0.83	81
Timing/Sleep-based Evasion	0.86	0.96	0.91	202
User Interaction Detection	0.97	0.99	0.98	351
Virtual Machine/Sandbox Detection	0.97	0.87	0.92	176
micro avg	0.95	0.96	0.96	1806
macro avg	0.95	0.95	0.95	1806
weighted avg	0.96	0.96	0.96	1806
samples avg	0.82	0.82	0.82	1806

Figure 34 - LSTM model Performance matrixes

The LSTM model outperformed both the VAE and tree-based models in all key performance metrics. Despite the tree-based models achieving high accuracy that they were limited by their inability to handle sequential data and their tendency to overfit. On the other hand, the LSTM model demonstrated robust generalization and high precision, recall, and F1-scores, making it the most effective approach for detecting sandbox evasion techniques in malware.

Also the LSTM-based model proves to be a powerful solution for multi-label classification tasks involving sequential data, such as API call sequences in sandbox evasion detection. It significantly outperforms alternative methods like VAE and tree-based models, establishing it as the preferred model for this task.

Loss and Accuracy Curves:

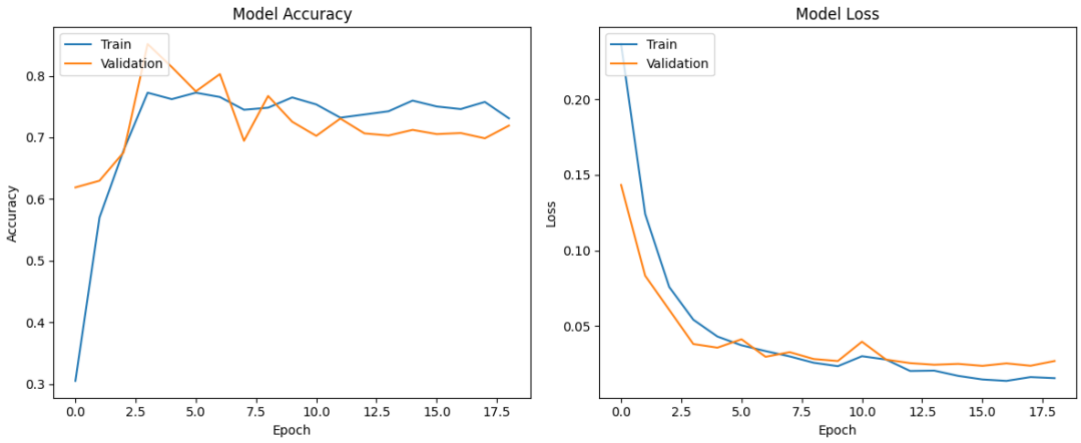


Figure 35- Loss and Accuracy Curves

The chart above illustrates the accuracy and loss training and validation curves of the LSTM model over 20 epochs.The accuracy curve shows that the model increases dramatically during the first few epochs, with a big boost in training accuracy followed by a stabilize towards the end of training. The validation accuracy, which suffered early on compared to the training accuracy, fades away as the epoch increases, which indicates that the model can generalize to unseen data. The training loss is showing consistent decline, however, the validation loss similarly decreases at a similar rate, and shows very few signs of overfitting.

These tendencies imply that the model has learned quite well, including the essential data patterns while generalizing effectively on the validation data. Furthermore, it does not demonstrate any obvious signs of overfitting since the training and validation loss curves are relatively aligned, and there is no divergence after several epochs, which is a common indicator of overfitting.

Visual Results:

The graph depicts a clear representation of the advancement of performance over time, referring to both accuracy and loss, for the training and validation datasets. The accuracy curve on the left reflects a constant increase for the training and validation dataset and shows that the model achieves a validation accuracy of about 0.95. The

same trend is also evidenced in the loss curve on the right, which shows a constant drop in loss, which is preferred for a well-trained model.

Ultimately, the proposed model exhibited strong convergence, yielding a high accuracy score with corresponding low loss values, which demonstrates the model's ability to learn the sequential information amongst API calls used to detect sandbox evasion techniques. These curves also confirm that the model is strong, as there is no evidence of significant underfitting or overfitting during training, meaning that it is suitable for use in the real-world problem of malware detection.

Behavior Report Analysis with Trained Model:

This section show how analyzed the model's behavior with three report.json files; two were malicious activity and the other benign activity. The intent of this experiment was to analyze the effectiveness of the trained model in the detection of sandbox evasion technique using the extracted API sequences from the Cuckoo Sandbox behavior logs.

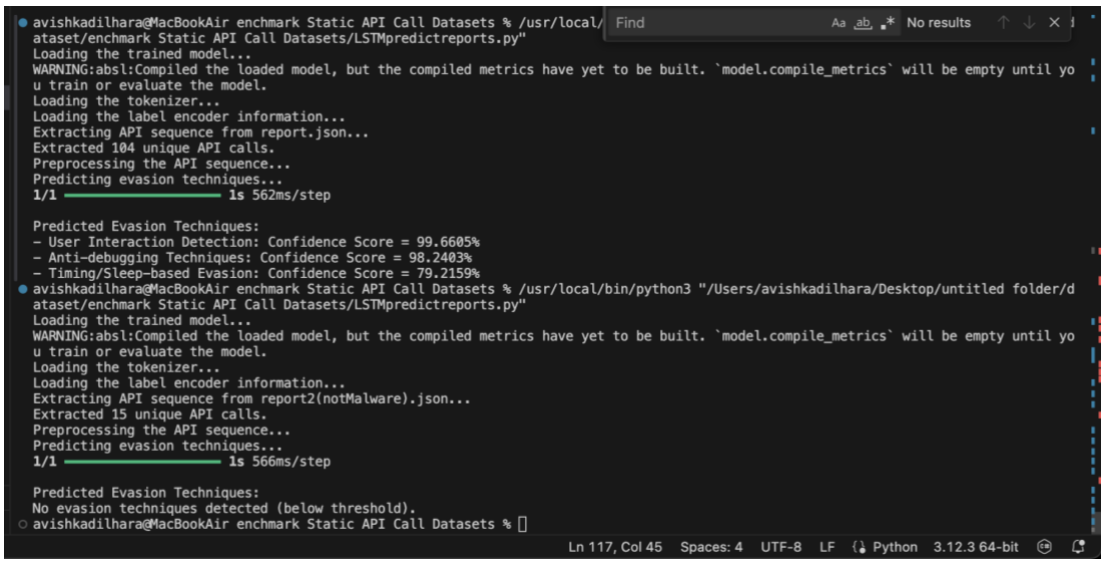


Figure 36 - Analyzing behaviors reports

For the first malicious report, the model predicted three different evasion techniques: **User Interaction Detection**, **Anti-debugging Techniques**, and **Timing/Sleep-based Evasion**, with high confidence scores of 99.66%, 98.24%,

and 79.21%, respectively. These predictions demonstrate the model's capacity to recognize various evasive behaviors that malware uses to bypass detection.

On the other hand, the benign report showed that no evasion techniques were detected. This is consistent with the model's ability to correctly identify the lack of evasion techniques, indicating that it is able to distinguish benign from malicious behavior. In addition, this further supports the model's accuracy in discrimination between malware efforts to evade detection and legitimate software.

The analysis reveals that the trained LSTM model achieves favorable results when tested in real-world situations by correctly identifying evasive techniques used by malicious samples while also accurately classifying benign behavior as "non-evasive". This performance corroborates the earlier reported metrics and demonstrates the sandbox evasion capability of the model.

1.9. Research Findings

The findings from this study show that the hybrid detection model that integrates LSTM (Long Short-Term Memory) networks with rule-based techniques can significantly advance the detection of sandbox evasion techniques in malware analysis. The LSTM model was able to learn sequential patterns in API (Application Programming Interface) call sequences and to detect evasive behaviors such as anti-debugging, timing attacks, and user interaction detection with high accuracy. This model demonstrates promise in addressing the challenges of traditional malware detection techniques, which are limited in their ability to detect polymorphic and metamorphic malware that alter their behavior to evade detection. The model was also able to use LSTM’s capability of handling sequential data to improve detection of complex and previously unseen evasive techniques.

Additionally, the study highlighted the necessity of class imbalance in the dataset. SMOTE (Synthetic Minority Over-sampling Technique) was shown to balance the dataset measured from the model improving the balanced dataset’s detection rate of

rare evasion techniques. Without SMOTE, the model learned poorly how to detect classes of samples with low representation; this led to low recall and precision rates for certain evasion types. However, with SMOTE applied, the model generalized well with performance improving across the board in all evasion techniques. This study indicates that dataset balancing is of critical importance in multi-class classification learning and especially true with difficult and highly imbalanced real-world datasets.

In spite of these victories, there were numerous issues faced during the research. When the research initially began utilizing a Variational Autoencoder (VAE) model, the accuracy was quite low and there were many limitations surrounding malware behavioral modeling. Tree-based models, such as the Gradient Boosting Classifier (GBC) and the Random Forest Classifier (RFC) were also encountered. While they did hold some potential, this potential was not realized, due to issues of overfitting and poor sequential data handling. As such, the research switched focus to the LSTM model, which was acknowledged as the best fit when modeling temporal dependencies in sequences of API calls. Ultimately, there were limitations with the LSTM model too, where overfitting was again an issue. These limitations ultimately required multiple approaches, such using dropout layers and early stopping layers to counteract some of the issues, ensuring there was some equilibrium between model performance and the complications of having a complex model.

Although the model displayed outstanding capabilities during test conditions to identify evasion techniques, more research should be conducted to optimize the model in real-time, large-scale environments. Challenges to consider include handling edge cases, detecting new evasion techniques, and reducing the compute footprint in production environments. Furthermore, ethical concerns associated with adopting automated malware detection systems in mission-critical environments should be considered to prevent these models from causing unintentional harm or result in biased decisions. This research has shown that LSTM-based models, when combined with sophisticated preprocessing methods, show the potential to improve performance in detecting malware in sandbox environments. However,

implementation is not practical until further work is done in testing and implementation in real-world environments.

4. DISCUSSION

The current research on sandbox evasion detection utilizing Long Short-Term Memory (LSTM) models provided many interesting opportunities for enhancement in the realm of malware analysis and detection. First and foremost, we proved that LSTM hybrid detection systems—not only LSTMs themselves—could be effective at detecting various sandbox evasion tactics of malware, especially when using preprocessing tasks (e.g. tokenization, padding, and multi-label encoding) to the dataset. Additionally, our findings suggest that these types of neural methods, more specifically deep learning methods, have the opportunity to solve challenging security problems that traditional static, rule-based detection systems could not.

The study does point out some aspects that need further research and improvement. While their LSTM model was robust in detecting evasion techniques, there are still challenges to deploying such systems in a real-world context. One of the biggest challenges is the constant evolution of malware so that it can evade detection. As demonstrated in the first instance with the Variational Autoencoders (VAE) and tree-based models, finding previously unseen evasion techniques continues to be a problem. Future work may want to focus on improving the model's ability to generalize to unseen malware (e.g. through transfer learning or improved anomaly detection techniques).

Another significant consideration relating to the dataset enhancement. The model's success was highly dependent on SMOTE (Synthetic Minority Over-sampling Technique) to mitigate class imbalance, which led to an improvement to detect rare evasion techniques. While this worked well in such context, it would not apply to every dataset, specifically when there is very few access to labels or there are highly imbalanced classes. In this case, other variations such as semi-supervised learning and unsupervised anomaly detection may be studied. Furthermore, additional work would need to ensure there is no class bias in the model and it can adapt to malware evolution.

Ethical considerations also represent a major challenge for the application of this model in real-world environments. Although the model's predictions are promising, incorporating it into a fully automated security system raises significant issues around transparency, accountability, and fairness. It is essential to establish appropriate ethical standards for automated malware detection systems to be used in situations where the outcome may affect user privacy or impact organizational operations. These systems must be fair and crucially avoid collateral damage, for example, excessive false positives that would prevent users from doing legitimate tasks.

In addition, the convergence of this model with sandbox environments, such as Cuckoo Sandbox, has limitations in scalability and real-time performance. However, although the model demonstrated its ability to detect evasion techniques from reports of behavior, work will be required to optimize and fit the model to the real-time analysis needs in practice. Given the rise in the number of malware samples, it will be vital to achieve high throughput and accuracy to successfully deploy this model at scale.

In summary, although the performance of the LSTM model is promising in terms of detecting sandbox evasion techniques there are a few areas that require improvement and additional consideration. Unseen malware challenges, the balance and quality of the dataset, ethical considerations and careful analysis, and improvement of real-time optimization are some areas of consideration for further investigation. Future work will need to address these elements, utilize more advanced models, develop improved data collection techniques and ethical standards to provide the best capable solution in terms of an ethical and reliable solution.

5. CONCLUSION

In summary, the research conducted in this thesis presents a significant advancement in the detection of sandbox evasion techniques used by modern malware. By employing a hybrid model that combines rule-based methods with Long Short-Term Memory (LSTM) networks, this study has demonstrated the potential of machine learning in overcoming the limitations of traditional malware detection systems. The LSTM model, specifically designed for sequential data, effectively captured the behavioral patterns of malware in sandbox environments, enabling the detection of various evasive tactics. This approach not only addresses the growing sophistication of malware but also provides a scalable and adaptable solution for real-time malware analysis.

The findings of this research emphasize the importance of advanced data preprocessing techniques, such as tokenization, padding, and multi-label encoding, which enabled the model to handle complex sequential data more effectively. Moreover, balancing the dataset using techniques like SMOTE proved to be critical in ensuring that rare evasion techniques were adequately represented, leading to improved model performance. Despite the challenges encountered, such as class imbalance and the limitations of initial attempts with Variational Autoencoders (VAE) and tree-based models, the LSTM-based approach proved to be successful in detecting evasion strategies with high accuracy and minimal overfitting.

Yet this study brings to light areas of inquiry to pursue and improve, specifically with respect to the generalization of the model to unseen malware and real-time performance in sandboxed environments. Furthermore, ethical problems such as transparency and fairness in automated malware detection should be considered in future developments. As malware continues to evolve, this study begins a pathway to the development of more advanced and ethical approaches to malware detection that will adapt to new forms of cyber threats.

At the ending the findings of this study contribute to the rapidly expanding field of cybersecurity by providing guidance for detection of evasive behavioral characteristics associated with malware. The work here clearly provides a strong method for supplementing existing sandbox detection techniques that are intended to combat the growing number of potential threats in modern computing environments. Moving forward, additional research should include clarifying this model through further refinement, scalability, and real word applicability of the findings to provide protection against an ever-increasing landscape of cyber threats.

6. REFERENCES

[1] T. Porutiu, How Sandbox Security Can Boost Your Detection and Malware Analysis Capabilities, ENDPOINT PROTECTION & MANAGEMENT, March 28, 2024.

[2] "Sandbox detection and evasion techniques. How malware has evolved over the last 10 years," Ptsecurity.com. [Online]," [Online]. Available: Available: https://global.ptsecurity.com/analytics/antisandbox-techniques. [Accessed: 11-Apr-2025]..

[3] "S. G. a. M. Picado, "Agent Tesla amps up information stealing attacks," 02 02 2021. [Online].," [Online]. Available: Available: 02Feb2021.https://news.sophos.com/enus/2021/02/02/agent-tesla-amps-up-informati on-stealing-attacks.

[4] Anishnama, "Understanding LSTM: Architecture, pros and cons, and implementation," Medium, 28-Apr-2023. [Online]," [Online]. Available: Available: https://medium.com/@anishnama20/understanding-lstm-architecture-pros-and-cons-and-implementation-3e0cca194094. [Accessed: 11-Apr-2025]..

[5] A. S. Abhijit Mohanta, Malware Analysis and Detection Engineering A Comprehensive Approach to Detect and Analyze Modern Malware, 2022.

[6] Y. S. D. D. X. L. a. M. L. F. Xiao, Novel Malware Classification Method Based on Crucial Behavior, Mathematical Problems in Engineering, no. doi:10.1155/2020/6804290, 2020.

[7] S. Morgan, Cybercrime To Cost The World $10.5 Trillion Annually By 2025"," Cybercrime magazine website. Cybersecurity ventures, 2022.

[8] C. TRAP, "The Evolution of Malware: From Intricacies to Solutions," ANUARY 19, 2024. [Online]," [Online]. Available: https://www.canarytrap.com/blog/malware-evolution/#:~:text=A%20Brief%20History,prank%20than%20a%20security%20threat...

[9] B. Alsulami, A. Srinivasan, H. Dong and S. Mancoridis, Lightweight behavioral malware detection for windows platforms, 2017 12th International Conference on Malicious and Unwanted Software (MALWARE), 2017.

[10] S. O. C. -.. SecureOps, How Malware Uses Encryption To Evade Cyber Defense, Problems And Solutions For The SOC, AUGUST 13, 2019.

[11] B. Alsulami, A. Srinivasan, H. Dong and S. Mancoridis, Lightweight behavioral malware detection for windows platforms, 2017 12th International Conference on Malicious and Unwanted Software (MALWARE), 2017.

[12] I. Alsmadi, B. Al-Ahmad and M. Alsmadi, Malware analysis and multi-label category detection issues: Ensemble-based approaches, 2022 International Conference on Intelligent Data Science Technologies and Applications (IDSTA), 2022.

[13] C. Wagner, G. Wagener, R. State and T. Engel, Malware analysis with graph kernels and support vector machines, 2009 4th International Conference on Malicious and Unwanted Software (MALWARE), 2009.

[14] o. A. Marpaung, M. Sain and H.-J. Lee, Survey on malware evasion techniques: State of the art and challenges, 2012 14th International Conference on Advanced Communication Technology (ICACT), 2012.

[15] S. Jain, T. Choudhury, V. Kumar and P. Kumar, - Detecting Malware & Analysing Using Sandbox Evasion, 2018 International Conference on Communication, Computing and Internet of Things (IC3IoT), 2018.

[16] T.-H. Cheng, Y.-D. Lin, Y.-C. Lai and P.-C. Lin, Evasion Techniques: Sneaking through Your Intrusion Detection/Prevention Systems, IEEE Communications Surveys & Tutorials , 2019.

[17] A. K. Sinha and S. Sai, Integrated Malware Analysis Sandbox for Static and Dynamic Analysis, 2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT), 2023.

[18] S. S. Darshan, M. A. Kumara and C. Jaidhar, Windows malware detection based on cuckoo sandbox generated report using machine learning algorithm,

2016 11th International Conference on Industrial and Information Systems (ICIIS), 2016.

[19] M. P. J, A. C. D, A. AS, S. P. B. R and R. S.M, Malware Detection using Machine Learning, 2024 Second International Conference on Advances in Information Technology (ICAIT), 2024.

[20] B. Bokolo, R. Jinad and Q. Liu, A Comparison Study to Detect Malware using Deep Learning and Machine learning Techniques, 023 IEEE 6th International Conference on Big Data and Artificial Intelligence (BDAI) , 2023.

[21] M. Sewak, S. K. Sahay and H. Rathore, Comparison of Deep Learning and the Classical Machine Learning Algorithm for the Malware Detection, 2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2019.

[22] M. Ficco, alware Analysis by Combining Multiple Detectors and Observation Windows, june 2022.

[23] A. K. S. a. S. Sai, Integrated Malware Analysis Sandbox for Static and Dynamic Analysis, 2023.

[24] S. J. Z. Y. L. W. a. Z. Y. X. Jianhua, Research on Malware Variant Detection Method Based on Deep Neural Network, 2021.

[25] A. V, Malware Detection using Dynamic Analysis,, 023 International Conference on Advances in Intelligent Computing and Applications (AICAPS), Kochi, India,, 2023.

[26] A. O. a. R. J. Sassi, Behavioral Malware Detection Using Deep Graph Convolutional Neural Networks," TechRxiv. Preprint, no. https://doi.org/10.36227/techrxiv.10043099.v1, 2019., 2019.

[27] F. Ç. E. G. AF. Yazı, Classification of Metamorphic Malware with Deep Learning (LSTM), IEEE Signal Processing and Applications Conference, 2019.

[28] Dambra, "Kaggle," GREIMAS, 2023. [Online]. Available: https://www.kaggle.com/datasets/greimas/malware-and-goodware-dynamic-analysis-reports..

[29] Khas-Ccip, "GitHub," API Sequences Malware Datasets, 2023. [Online]. Available: https://github.com/khas-ccip/api_sequences_malware_datasets..

[30] A. Oliveira and R. J. Sassi, "Behavioral Malware Detection Using Deep Graph Convolutional Neural Networks," *TechRxiv. Preprint,* no. https://doi.org/10.36227/techrxiv.10043099.v1, 2019.

[31] F. Ç. E. G. AF. Yazı, "Classification of Metamorphic Malware with Deep Learning (LSTM)," *IEEE Signal Processing and Applications Conference,* 2019.

[32] F. Y. A. Catak, "A Benchmark API Call Dataset for Windows PE Malware Classification,," vol. arXiv:1905.01999, 2019.

[33] Dambra, "Kaggle," GREIMAS, 2023. [Online]. Available: https://www.kaggle.com/datasets/greimas/malware-and-goodware-dynamic-analysis-reports.

[34] Y. S. N. P. R. G. T. a. G. R. K. R. S. Jamalpur, Dynamic Malware Analysis Using Cuckoo Sandbox, 018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), 2018.