Knowledge-Based Systems 331 (2025) 114830

Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

DeepRadar: A cyber-defence interceptor for early warning and defusing malware injection attacks

Danial Javaheri [a,*], Hassan Chizari [b], Mahdi Fahmideh [c], Mohammad H. Nadimi-Shahraki [d], Junbeom Hur [e]

[a] Department of Computing, Engineering and Physical Sciences, University of the West Scotland, London Campus, London E14 2BA, United Kingdom
[b] School of Business, Computing and Social Sciences, University of Gloucestershire, The Park Campus, Cheltenham GL50 2RH, Gloucester, United Kingdom
[c] School of Business, University of Southern Queensland, Springfield, QLD 4300, Australia
[d] International Graduate School of Artificial Intelligence, National Yunlin University of Science and Technology, Douliu, Yunlin, Taiwan
[e] Department of Computer Science and Engineering, Korea University, Seoul 02841, the Republic of Korea

ARTICLE INFO

Keywords:
Deep learning
Early warning system
Fast Fourier convolution
Association rule mining
Malware detection

ABSTRACT

Malware injection attacks are among the most sophisticated and elusive threats in cybersecurity, characterised by their capacity for privilege escalation, obfuscation, and the ability to deceive antivirus software. This paper introduces a multi-layer architecture, featuring innovative deep neural networks, *fast Fourier convolution*, and association rule mining strategies, designed for the early detection and defusal of malware injection attacks. We then propose a proactive AI-enabled malware detection platform, *DeepRadar*, as a novel real-world defence mechanism. This early warning functionality capable of anticipating the attack a few cycles before occurrence represents a novel idea and unique approach to detecting malware injection attacks. The experimental results validate *DeepRadar's* superior performance compared to not only previous related studies but also a standard benchmark of well-reputed antivirus applications under various scenarios and accredited datasets, including heavily obfuscated emerging malware variants and adversarial samples. It demonstrates higher Accuracy, F-score, ROC, and AUC metrics in early detection and classification of malware injection attacks while *DeepRadar* consumes significantly fewer system resources, including processor and memory during long-term scalable operation. The proposed early warning system succeeded in repelling up to 97.2% of attacks before malware could complete their malicious sequence. Lastly, the evaluation results were substantiated by formal statistical analysis using Friedman and Wilcoxon tests. The findings of this research and *DeepRadar's* runtime scanner provide vital early warnings against stealthy malware and injection attacks, offering robust protection for sensitive systems and critical infrastructure.

1. Introduction

Modern cyber-attacks are complex, stealthy, and in some cases backed by well-resourced organisations with expert teams and substantial budgets. Such advanced threats often evade detection and neutralisation, even with state-of-the-art defensive strategies and tools [1], particularly those that rely on predefined signatures or static behaviours, such as firewalls, intrusion detection systems (IDS), and antivirus (AV) programs [2]. A 2025 study [3] reported that cyber incidents impact roughly one-third, 32%, of companies across all sectors, underscoring the pervasive threat landscape. Consequently, cybercrime has evolved into a complex ecosystem, with attackers moving from isolated activities to sophisticated, coordinated operations. A significant portion of recent cyber-attacks involves various types of malware, which are responsible for about 90% of system failures [4]. According to the data released by AV-Atlas,[1] more than 1.2 billion malware pieces were detected as of the end of 2024 and the growth rate of malware for Windows was 29 times faster than Android and 134 times faster than Mac. This dramatic increase is attributed to the widespread availability of obfuscation and metamorphic engines [5]. The severity of attacks is

* Corresponding author.
E-mail addresses: Danial.Javaheri@uws.ac.uk (D. Javaheri), hchizari@glos.ac.uk (H. Chizari), Mahdi.Fahmideh@unisq.edu.au (M. Fahmideh), nadimi@yuntech.edu.tw (M.H. Nadimi-Shahraki), jbhur@korea.ac.kr (J. Hur).

[1] Available on https://portal.av-atlas.org/malware

https://doi.org/10.1016/j.knosys.2025.114830
Received 11 March 2025; Received in revised form 3 October 2025; Accepted 2 November 2025
Available online 3 November 2025

further complicated by the increasing sophistication, intelligence, evasion, and target-specific nature of these malware programs [6,7].

One of the most prevalent techniques for concealing malware activity involves injecting its code and libraries, including dynamic link library (DLL) and System (Sys) files, into the executable memory of other active programs. This approach allows the injected code to exploit the privileges and signatures of the programs it infects. Moreover, sophisticated new malware types use injection to deceive monitoring tools, especially antivirus software, by distributing their destructive actions across different running processes [8]. Injection attacks, including code and library injection, query injection, and script injection [9], were listed on the top ten cyber-attacks worldwide between 2017 and 2021, according to OWASP.[2] For example, code injection, data injection, and fault injection [10] have been widely disseminated in various systems, such as wireless sensor networks, cyber-physical systems [11], smart grids, and modern power plant systems [12].

Therefore, this research answers the following research questions: What are the steps of a malware injection attack and how to formally model the behaviour of injection attacks? How to accurately predict an or multiple imminent injection attacks carried out by today's extremely obfuscated malware programs, before the attack steps are completed? How to halt and defuse an early detected attack and protect the system resources and processes against running attacks?

This paper introduces a novel cyber-defence system, *DeepRadar*, designed for early and precise detection of malware injection attacks, particularly targeting code injection and library injection. The key contributions and novelties of this work are: (a) a novel trained model for detecting injection attacks using deep neural networks that benefit from *fast Fourier convolution* with a dedicated architecture, (b) incorporating a generative adversarial network to bridge the scarcity of the number of rare malware, (c) an early warning system based on a trained association rules mining algorithm to anticipate the injection attack a few cycle before taking place, (d) a dynamic scanner that intervenes to halt injection attacks that may have evaded initial detection and begun their operation. This scanner acts as a final line of defence, ensuring comprehensive protection. To the best of our knowledge, our proactive early warning system is the first approach for accurate anticipation and neutralisation of injection attacks in the literature, repelling up to 96% of code and DLL injection attacks before they are completed.

This paper is organised as follows. Section 2 reviews the literature, providing background on malware injection attacks and examining related mitigation studies. Section 3 presents the proposed approach for anticipating and early detecting such attacks. This includes training various deep learning models, implementing rule mining, and introducing a novel neural network architecture, with all these components integrated into *DeepRadar*, a runtime scanner. Section 4 rigorously evaluates *DeepRadar's* efficacy, performance, and efficiency under various scenarios, comparing it from various perspectives to several well-known solutions and real-world tools. Finally, Section 5 concludes the paper and outlines future research directions.

2. Background and related work

Obfuscation techniques are widely used to manipulate control flow and deceive anti-malware tools, including both static- and dynamic-based malware detection methods [13]. In static detection, behaviour analysis is performed without executing binary files stored on hard drives. However, a dynamic analysis involves running malware binary files or scanning memory-resident codes that are already running [14]. One of the most effective methods in obfuscation is distributing all or part of the malicious code into the body of other benign programs and then executing that malicious code under the cover of trusted programs. This technique can deceive malware detection strategies by exploiting the signatures and privileges of other programs, i.e., privilege escalation. In such scenarios, each program may seem harmless in isolation, yet collectively, they create a pattern indicative of malicious intent. This form of malware is referred to as distributed malware [15]. This injection method is prevalent among advanced persistent threats (APTs) as it allows the malware to remain anonymous for a long course of time [16,17]. An APT attack is a long-term, covert intrusion targeting enterprises, national infrastructure, or government departments, often leveraging advanced techniques, hacker organisations, and state-sponsored resources - areas where traditional defences face clear limitations. The evolving nature of such attacks necessitates innovative approaches to detection and attribution, with cyber threat intelligence (CTI) sharing playing a crucial role in harnessing expert knowledge, enhancing intelligence, strengthening detection, and resisting network threats [18]. The attackers typically hold a time and resource advantage, placing defenders in a passive position. Due to this information asymmetry, defenders often lack visibility into the attacker before an APT attack, while attackers can prepare by gathering intelligence on their targets [19]. To bridge this gap, proactive deception defence mechanisms, such as lightweight Honeypoint [20], can expose covert threats and APT behaviours, underscoring the importance of integrating deception strategies into long-term cybersecurity defence.

A significant gap in related literature is the lack of detailed discussion on the interception and analysis of distributed interactions among running processes in cases involving this type of malware. Another way to perpetuate a destructive system attack is by injecting a fake library into the memory of a victim program at runtime. The primary objective of library injection is to alter a program's behaviour by redirecting and obstructing system calls, thus hijacking its control flow [21]. The challenge becomes even more formidable when attackers employ obfuscation strategies such as dead-code insertion, code encryption, packing, polymorphism/metamorphism, and anti-debugging/sandboxing techniques [22].

In a library injection attack, the first step is to inject a library containing fake routines of system APIs into the memory of the victim program. The injection process is performed using an injector tool. A redirector stub then redirects the victim's requests - created for accessing system APIs - to fake functions that are already loaded into the victim's process memory. The redirector stub completes the hooking process by changing the addresses stored in the Import Address Table (IAT). The IAT is a table in which the labels for system API calls, references to memory locations, and the names of the modules that own those functions are stored. A Linker program can call OS APIs by reading their addresses from the IAT. Using this information, malware can replace the original addresses with new ones that point to its own functions already loaded into the memory of the victim program by the injected library. Therefore, the victim program calls injected malicious functions instead of the original OS functions. This is the process through which a malware program can take over the execution of an application and force it to execute a malicious code [23]. Infectors and Binders are types of malware that use this strategy to distribute malicious code among trusted processes [24,25].

2.1. Deep learning techniques in malware detection

Deep learning models, including, such as RNNs, CNNs, and GNNs, stands as a core in malware detection, offering sophisticated methods for both detection and analysing today's extremely obfuscated malware programs. DL models lie in the ability to automatically learn intricate behavioural features and patterns from large-size and multi-dimensional datasets of malware, going far beyond traditional signature-based methods that struggle with evolving threats and are ineffective against ever-increasing number of malware attacks nowadays. This capability to adapt and recognise indicators of malicious behaviour provides a strong foundation for malware detection using deep learning and artificial neural networks. The field has been witnessing a surge in research and

[2] Available on https://owasp.org/www-project-top-ten/

development, with plenty studies showcasing the effectiveness of deep learning models in accurately detection and classification even previously unseen and zero-day malware [26]. A recent comprehensive survey by Song et al. [27] on the application of deep learning in malware detection highlights the pressing need for advanced tools capable of early detection of malware and its variants through behavioural pattern analysis in large-scale malicious data, with deep learning offering substantial research potential in this domain. This survey categorises deep learning models for malware detection into three main groups: (a) deep learning algorithms, with CNNs leading the list followed by RNNs and GANs; (b) data augmentation techniques, such as the Synthetic Minority Over-sampling Technique (SMOTE) and the Bat Algorithm–based approach; and (c) imaging methods, including Image Vectorisation, Mean Normalisation, and Hamming Distance, drawing on studies published between 2018 and 2025.

However, despite this rapid progress and the substantial body of research, significant challenges and open problems still remain. Malware programs, in particular stealth classes, continues to swiftly evolve, employing increasingly sophisticated techniques like polymorphism and metamorphism to evade detection [28]. A particularly pressing issue is the rise of malware injection attacks, where malicious code is seamlessly inserted into legitimate applications or processes. These attacks are notoriously difficult to detect as they leverage trusted software to mask their true nature. Addressing these evolving threats requires ongoing research into more robust and adaptable machine learning and AI-enabled models, capable of not only identifying known malware, but also recognising the subtle anomalies indicative of injection attacks and precisely predict them before they take place [29]. Another significant challenge with deep learning models is their reliance on large datasets, a limitation that becomes critical in the case of zero-day malware or rare classes where only a few samples are available. This scarcity makes DL models particularly vulnerable to such threats. To address this, Chai et al. [30] introduced the concept of Few-Shot Learning (FSL), which aims to learn effectively from limited examples in malware detection. Inspired by the human ability to generalise new knowledge from only a few experiences, the authors formalised malware detection as an FSL problem, offering a novel perspective for tackling data scarcity. Nevertheless, FSL still faces two major challenges, including (a) catastrophic forgetting, where newly acquired knowledge erodes previously learned knowledge; and (b) decision boundary confusion, where repeated incremental sessions weaken the model's discriminative power. To address these limitations, MalFSCIL [31] was later introduced as a novel Few-Shot Class Incremental Learning framework. It combines a decoupled training strategy with a variational autoencoder to mitigate catastrophic forgetting and employs a class-prototype-based dynamic boundary method to improve the accuracy of incremental decision boundaries.

Furthermore, the computational cost, including memory and processor utilisation, of training and deploying deep learning models in real-world scenarios must be carefully considered and addressed.

2.2. Prior work

This section investigates and reflects on the strengths and limitations of related studies, including those directly focused on the detection of injection attacks and those that have developed malware detection approaches capable of identifying malware classes that include injection attacks.

Korczynski and Yin introduced a unified automated approach using a malware analysis environment called *Tartarus* [32]. The goal was to trace and detect malware propagation and execution through OS and inside the memory of benign processes by abstracting the execution trace. This study intended to detect malware programs that used novel code injection methods, code-reuse attacks, and dynamic code generation techniques. The results of experiments with real-world malware samples showed improvements in the accuracy of detecting malware execution traces. They also claimed that their method was able to catch intrinsic features in modern code injection attacks. However, this approach only considers malware propagation within a single system, meaning it is ineffective against malware that can spread through networks—a behaviour commonly observed in modern ransomware attacks. Additionally, since this method is based on QEMU virtualisation technology, it is vulnerable to malware classes that can detect the presence of QEMU, allowing them to conceal their malicious actions or self-destruct.

Wei and Zhu presented an in-depth defensive framework they called *KQguard* to address queue injection attacks at the OS kernel level [33]. Within this framework, kernel callback queues (KQs) are targeted by malware for performing kernel queue injection (KQI) attacks. KQs are used as a solution for event handling in recent OS kernels. The proposed method utilised a hybrid static and dynamic analysis of device drivers at the kernel space to learn the specifications of legitimate event handlers. Their proposed strategy declined unknown KQ requests that could not be verified at runtime. This strategy was effective for both Windows and Linux kernels with low false positive (FP) and low false negative (FN) rates when running about 1500 real-world kernel-level malware samples. However, *KQguard* provides protection against injection attacks only on 32-bit operating systems, meaning it cannot be installed on the more dominant 64-bit operating systems used today. Furthermore, the system has been trained and tested on a limited sample of malware and small datasets, making it unsuitable for scaling to handle today's large, multi-dimensional malware datasets.

Spoto et al. presented a method for recognising five different types of injection attacks against Java applications and Android OS [34]. The authors used abstract interpretation, a form of static analysis, as their primary approach to detect and analyse the malicious code responsible for injection behaviour. The detection accuracy was reported between 87% and 92%. One of the main limitations of this method is that it can only detect injection attacks in programs developed in Java, making it ineffective against malware samples written in other programming languages.

Dai et al. proposed a novel detection method using a combination of I/O Request Packet (IRP) sequence features and local alignment algorithms for recognising distributed malware [35]. In the first step, main IRP requests were filtered and extracted from the OS. A comparison between these requests and the malware's IRP sequences was the pivotal process for detecting the hidden pieces of distributed malware. Real malware samples were used in this study, and the results demonstrated that this approach was able to identify distributed malware with an accuracy ranging from 86% to 93%, surpassing previously proposed methods up to that point. However, since this method relies solely on IRPs as features for behavioural modelling, it tends to produce a high number of false positive errors, particularly when dealing with polymorphic malware that mimics the behaviour of benign applications.

Tyng Ling et al. proposed an ML-based method for identifying metamorphic malware. Their algorithm relied on structural analysis of statistical metrics and information-theoretic measures [36]. In this study, several features such as Jaccard coefficient, entropy, compression ratio, nonnegative matrix factorisation, and Chi-square tests were used to represent the byte information of malware pieces. The experiments indicated that the Jaccard coefficient could detect a metamorphic class of malware, capable of injection attacks, designed for Windows OS with an accuracy of 99.7% and an F-score of 99.5%. However, these figures were obtained through evaluation on a dataset with a small number of malware samples, and the scalability of this method has not been demonstrated to show whether it is effective against today's large and multi-dimensional malware datasets.

Panker and Nissim in [37], designed a framework for detecting unseen malware and malware evasion techniques, including injection, in Linux VM cloud environments. The authors collected volatile memory dumps from the inspected VM by securely querying the hypervisor to extract several behavioural features from over 218,000 samples across 9

malware classes. Using machine learning (ML) and deep learning (DL) classifiers, including LR, SVM, KNN, RF, and DNN, their proposed framework successfully detected unseen malware with evasion capabilities, achieving high true positive rates and low false positive rates. However, since this work focuses on feature extraction at the hypervisor level, it can only be deployed in virtualised environments. Additionally, the volatile memory acquisition process requires briefly freezing the VM, which may cause delays in client services. Another limitation is the passive-based learning approach, which lacks real-time malware scanning and detection; in contrast, recent malware detection methods are increasingly incorporating active learning-based solutions.

Lie et al. [38] introduced a dynamic graph-based learning approach to automatically capture evolving malware and detect six key categories of malware attacks, including injection attacks. Their proposed system, MalIRL, features a dynamic heterogeneous graph representation learning method that enhances detection accuracy by learning state representations of different attack stages and forensically analysing the malware execution event stream. In experiments with three real-world datasets, the model achieved accuracy ranging from 91% to 98% across various malware classes. Although this model improves accuracy, it imposes significant computational overhead due to ineffective exploration paths in inverse reinforcement graph learning and the need for exploring transfer learning techniques. Additionally, the approach lacks real-time responsiveness in network security defence, which is crucial for addressing today's ever-evolving malware threats.

Among the studies perused in this paper, approaches based on static code analysis for malware detection are not effective for dealing with obfuscated and packed malware classes and file-less [6] malware. There are also limitations to many of the related studies. Studies that used manual analysis for feature extraction were not scalable to handle the massive amount of malware produced daily and the large number of cyber-attacks that usually take place in a short timeframe. Further, using a small number of samples in model training does not enable the creation of a scalable model able to detect a wide range of malware classes with different behaviour patterns. Some related studies were limited to detecting injection attacks in specific programming languages [34] or only on 32-bit versions of operating systems [33]. In contrast, our proposed architecture overcomes these limitations by detecting injection attacks regardless of the programming language used to develop the malware and supporting both 32-bit and 64-bit OS versions.

The distributed nature of injection attacks is a major challenge that complicates the process of feature extraction for accurate and reliable modelling [8]. The adoption of obfuscation and evasion tactics, such as dead-code insertion, packing and code encryption, runtime decompression, polymorphism, and anti-debugging, exacerbates the challenge and further hinders effective detection and analysis [22].

Our system offers an accurate dynamic solution for early detection of injection attacks by creating reliable models using a new convolutional neural network and association rule mining. A unique aspect of *DeepRadar* is its ability to facilitate early detection and defusal of attacks before they are completed, a feature not discussed in the existing literature. This capability represents a significant contribution to the field.

3. The proposed approach

The objective of code and library injection is to hijack the victim's execution flow control. This level of control allows the malware to indirectly execute any part of its malicious code. This makes it very difficult for anti-malware programs to detect the responsible malicious file and its processes. Early detection of code injection patterns is our solution to this problem that facilitates tracing culprit processes. It is critical that detecting such a pattern should happen before the stage in which malware completes the injection process. This is because discovering the source of the attack after this stage is impossible since the connection between the malware program and the victim process terminates. Moreover, the victim's process might lose control over the execution flow. In this situation, restoring the control requires terminating the process or sometimes restarting the OS, which are costly decisions for many servers.

3.1. Modelling injection to binary files

The injection attack against binary files stored on a hard drive is commenced by creating a handle on the target file using the *CreateFile* or *OpenFile* system functions. Then, the address of the program's entry point is calculated. The entry point (EP) is an address from which the executable code of a program starts to execute. It is calculated according to Eq. (1).

$$EP = ImageBase + C \quad (1)$$

where *EP* is the program's entry point, *ImageBase* is the address in virtual memory in which the executable code is loaded and set according to the operating system, and *C* is a constant offset [39]. For the Windows OS family (up to Windows NT 6.0), the value of *ImageBase* was fixed at 400, 000, so calculating *EP* was quite simple. Microsoft used Address Space Layout Randomization (ASLR) in later versions of Windows OS to resolve buffer and heap overflows. ASLR randomly changes the space allocated to stack and heap at each execution. This guarantees that the value of *ImageBase* changes at every execution. This strategy has been widely used by other OS vendors, including Linux with 2.6+ kernel, Android Version 4.0+, Solaris version 11.1+, and iOS Version 4.3+. Although the use of ASLR significantly reduces code injection attacks, coding shells, and buffer/heap overflows, it complicates the calculation of *EP* and increases the entropy of the binary file.

After obtaining the EP of the executable file, a new section is added to the Portable Executable (PE) file in which malicious code is written by calling *WriteFile* system function. Next, the value of the Original Entry Point (OEP) is set to the initial address of the new PE file section. Therefore, at the beginning of each execution of the program, the malicious code is also executed [8]. Fig. 1 demonstrates sections and flags of the PE structure of an application targeted by an injection attack before and after the injection process. This information was obtained using the PEiD[3] tool. As shown in Fig. 1, by creating, renaming, and changing the addresses of other sections, the malware executes its malicious code at the beginning of the execution phase of the victim's program and distorts the behavioural features required for malware detection.

3.2. Modelling injection to running processes

Code and library injection into a running process without causing interruptions or failures in the process's activities requires several key countermeasures and considerations. A successful injection attack against a running process ensures that the malware can run its code immediately after the injection process is completed [40].

One way to accomplish an injection attack against a running process is by hijacking the *AppInit_Dlls* and *SvcHost_Dlls* registry keys in Windows OS family. A malware program can enrol the name and address of its fake DLL into these registry keys. This puts the malware's fake DLL into the list of libraries loaded by the OS after reboot. This method is used by many malware instances for injecting into OS modules [41]. With the release of Windows NT version 6.0+, Microsoft introduced several defensive mechanisms, including User Access Control (UAC) and Kernel Path Protection (KPP), and Address Space Layout Randomisation (ASLR) to prevent malware activities, in particular, unauthorised access to other processes' memory [16,42]. These new policies restricted the ability to hijack these registry keys for running nefarious code [8]. However, malware developers deployed new techniques for injection attacks.

[3] https://github.com/wolfram77web/app-peid

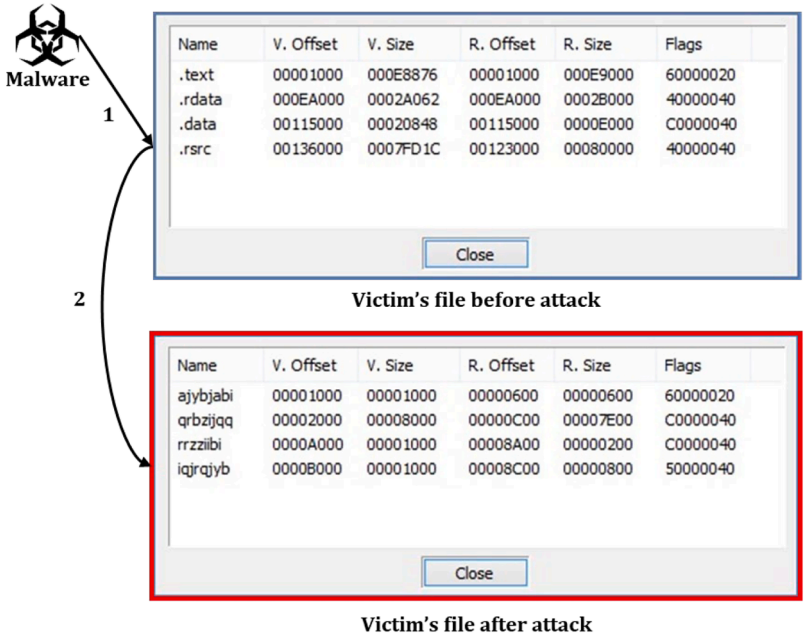


Fig. 1. The PE structure of a victim file before and after a code injection attack.

We meticulously analysed thousands of malware instances with the capability of code and library injection collected from Adminus [43], VirusShare [44], and VirusSign [45] malware datasets between 2015 and 2024. Our analysis revealed that creating a handle on a running victim process using *CreateProcess* or *OpenProcess* system functions is a common alternative injection strategy. The process is followed by a request to access the allocated memory of the running victim process by calling the *ReadProcessMemory* system function. The malware then requests to add a certain amount of free space to the allocated memory of the victim process using the *VirtualAllocEX* system function. This space is equal to the size of the code or library that will be injected. Using *WriteProcessMemory* system function, the malicious code is then injected into the reserved memory space.

Library injection attack occurs through a sequence of 6 steps: (1) To stop the main thread, the malware injects the assembly equivalent of the *SuspendThread* system function - along with the address of the fake library - into the victim's process. This occurs through code injection, which was described previously. (2) The assembly equivalent of the *LoadLibrary* function with the address of the desired fake library is injected. (3) The assembly equivalent of the *CreateRemoteThread* system function is injected to create a remote thread in the victim's process memory. (4) The injected codes reset the EIP flag and PC register so that the address of the victim's process points to the address of the injected library. (5) The malware uses the *WaitForSignalObject* system function to detect when the OS completes loading the fake library into the victim's process. (6) Finally, the malware calls *ResumeThread* system function to resume the originally interrupted process.

To mitigate the possibility of abusing the *CreateRemoteThread* system function through malicious code injection, Microsoft has enforced checks to curtail malicious use of this critical function in recent versions of the Windows OS - from Windows 7 onwards. Calling this function is restricted so that remote threads between two applications with different access levels or owners (for example, one application with user-level privilege and the other application with administrator-level privilege) cannot be created. Although the restrictions imposed proved effective in preventing the abuse of this system function, there are still methods that can be adapted to bypass these arrangements to create remote threads. For example, instead of *CreateRemoteThread,* malware developers use a combination of the *GetThreadContext* and *SetThreadContext* system functions. These functions change the context of the victim's process in a way that mimics the functionality of the *CreateRemoteThread* function. Therefore, a remote thread is created without directly invoking the *CreateRemoteThread* function. Further, the OS restrictions on calling the *CreateRemoteThread* function do not cover the kernel space. Therefore, malware classes designed to run within the kernel space can pursue their aims regardless of this restriction.

3.3. Detection and classification of injection attacks

The high-level architecture and workflow of our proposed approach is illustrated in Fig. 2. Four credible malware datasets between 2018 and 2025, including Adminus [43], VirusShare [44], VirusSign [45], and MaleVis [46], were used for feature extraction, including IRPs, APIs along with their call parameters and frequency as APIs are still the standard and core work of the most widely adopted malware detection methods [47,48] besides kernel-level IRPs required for self-defence and malware removal. The features were extracted in a hybrid manner consisting of both static (without execution) and dynamic (with execution in a Sandbox) methods, aiming to take advantage of the swift scanning process offered by static methods as well as the high analysis depth of dynamic approach required for dealing with malware evasion and obfuscation techniques. Static features are used to train a Logistic Regression model while the dynamic features are fed to the APRIORI model for generating early warning signals and to a Fast Fourier Convolutional Neural Network after being converted to RGB images, aiming to leverage the high accuracy of deep neural networks in classification.

3.3.1. Logistic regression (LR)

This mechanism involves a swift static analysis of the portable executable (PE) file to detect malicious behaviour corresponding to code injection attacks. To accomplish this, the LR scanner module searches for the names of functions corresponding to the pattern created by code injection attacks and uses them to detect malicious behaviour. *DeepRadar* scans the PE header and extracts the name and address of APIs registered in the IAT table to determine if any artefacts (footprints) related to injection attacks exist.

Using an LR classifier, the pattern of the attack is modelled as a sequence of system APIs. The LR classifier also examines the parameters of each API. The main factors used for modelling injection attacks in the proposed LR module are (1) the order of system calls, (2) parameters

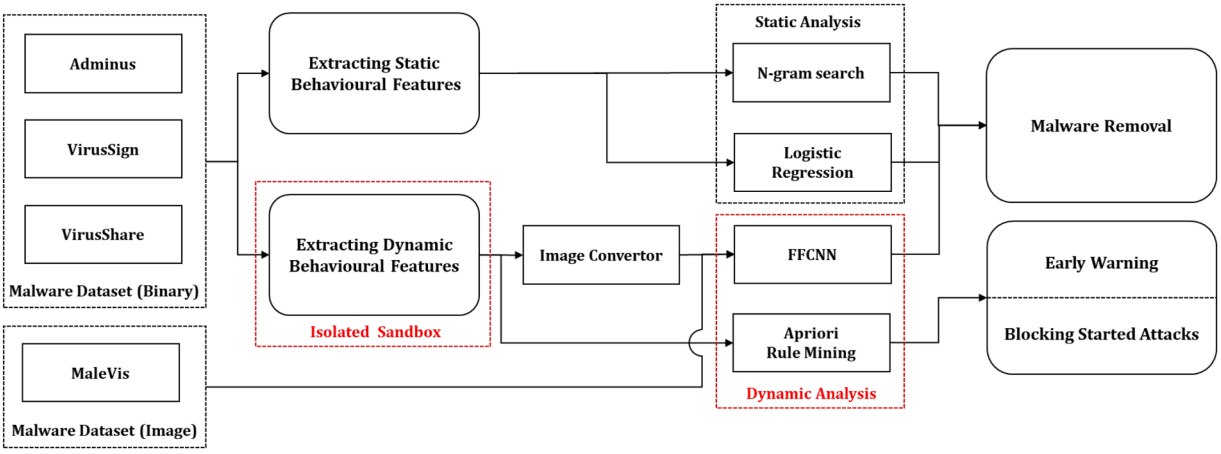


Fig. 2. The architecture of the proposed approach for training injection attacks detector models.

used for system calls, and (3) the frequency of the calls. In our LR model, malicious behaviour M, which represents code injection, is defined using Eq. (1).

$$M_1 = A \times X_A + B \times X_B + C \times X_C + \ldots \tag{1}$$

where A, B, C are system calls, X_A, X_B, X_C are their repetition frequency and M_1 is a sequence of chronically ordered function calls. This LR model creates a signature indicating the pattern of injection attacks, such that if a process shows such a chain through its system calls, the LR model immediately recognises it as malicious code with the aim of injection. Considering d as the number of features, the time (computational) complexity of this task can be computed as $O(n \times d) \simeq O(n)$ since d is constant.

Fig. 3 presents the sequence of APIs used for detecting malicious code and library injection attacks into running processes. The injection chain consists of eleven steps, including code injection (API calls from 1 to 8, inclusively), as well as library injection (API calls from 1 to 11) into a running process. The first eight steps indicate malicious code injection, while the total eleven steps together indicate library (DLL) injection. Certain malware programs use *CreateProcess* system function instead of *OpenProcess* to create a handle for a running process or *LdrLoadDll* instead of *LoadLibrary* to load a library into the system memory. These functions are considered equivalent. Fig. 4 shows the IAT for a sample malware with the capability of code injection. This figure was generated using the DIE[4] analysis tool.

Although the LR model is fast and effective for recognising malicious chains of injection attacks, a more robust detection model is required to deal with packed instances of malware as well as those that are capable of evasion techniques like PSP-Mal [49]. Packer and obfuscation tools, e.g. ASPack, NSPack, UPX, Themida, PETite, UPack, and ExeStealth, modify the body of malware programs, smash or encrypt the IAT and PE header in so that neither structural nor behavioural signatures can be reliably detected, thereby thwarting static analysis [50]. To deal with this threat, dynamic behaviour analysis modules based on a new convolutional neural network and APRIORI rule mining were also utilised in *DeepRadar*.

3.3.2. APRIORI association rule mining

One of the main contributions of our proposed system is the early detection of injection attacks. Previous methods were not equipped with early detection mechanisms. The early detection strategy makes it possible to defuse the attack and prevent subsequent damage to the OS and targeted programs. It also ensures that the execution flow of the targeted program is still maintained. If the process of attacking and loading the forged library is completed, the executive control of the program would be lost as a result of hooking APIs performed by the malware.

Our proposed method uses APRIORI rule mining as a bottom-up learning approach based on association rules [51]. As a major technique in data mining, association rules attempt to find frequent patterns or subsets among sets of objects in information repositories [52]. Association rule algorithms evaluate *Support* and *Confidence* of the *itemsets* as key measures for rule creation [53]. Using this algorithm, a model was trained for the early detection of attacks, which could be used to generate early warning signals. Algorithm 1 indicates the pseudo-code of the procedure executed on a dataset of kernel-based malware APIs to generate *itemsets*.

Rule generation is the next step performed after creating *itemsets*. Algorithm 2 shows how the rule generation phase was developed.

API chains are considered as *itemsets* in this algorithm, where M indicates malware sequence calls considering the threshold of ϵ and length of Y, while A and B are labels for APIs. In these algorithms, Q_y is the candidate subset of APIs for the level of Y. The algorithm tries to find frequent *itemsets* between candidates of APIs used in injection attacks until no more extensions are discovered. Having *itemsets* and APIs frequencies, an early warning system can predict injection attacks before the malicious chain of API calls is completed. By calculating the *confidence* and *lift* criteria for the chain presented in Fig. 3, the 6th and 7th steps of the sequence can be predicted. This triggers a warning signal before the full injection of the library occurs in the 5th step. *Confidence* and *lift* criteria are calculated based on Eq. (2) and Eq. (3) [51].

$$Confidence\ (A \rightarrow B) = \frac{Sup\ (A \cup B)}{Sup\ (A)} \tag{2}$$

$$Lift\ (A \rightarrow B) = \frac{Conf\ (A \rightarrow B)}{Sup\ (B)} \tag{3}$$

in which A and B are system call labels according to the chain presented in Fig. 3. The *confidence* criterion indicates the degree of interdependence between calls, which shows the probability of calling Steps 8 to 11 - provided that the first seven steps were completed. The *support* criterion indicates the ratio of the number of calls involving both A and B function calls. The *lift* criterion indicates the degree of independence between calls. Rule mining in this study was conducted through Weka[6] version 3.8.5. This model provides the capability of accurately

[4] https://horsicq.github.io/
[5] https://docs.remnux.org/install-distro/get-virtual-appliance.
[6] https://www.cs.waikato.ac.nz/ml/weka/

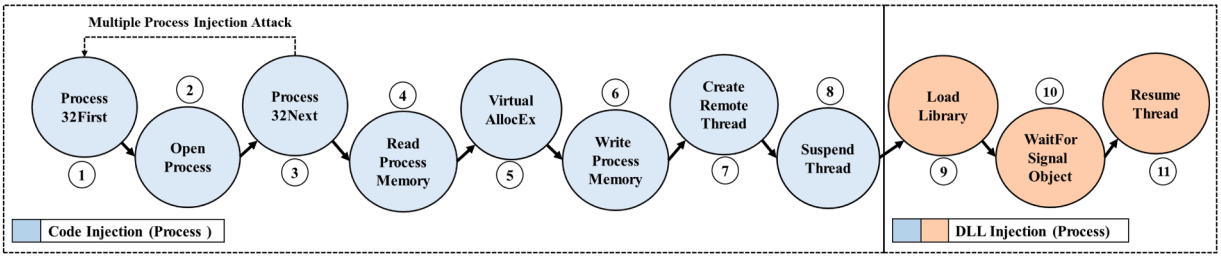


Fig. 3. The chain of APIs for dynamic detection of code and also DLL injection in the proposed method.

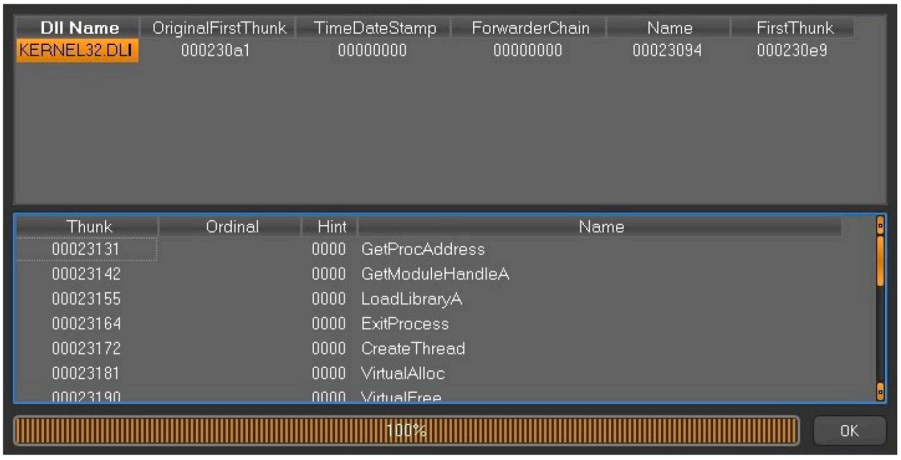


Fig. 4. Functions in the IAT of a malware sample with injection capability, obtained using DIE under REMnux.[5].

Algorithm 1
Itemset generation.

```
APRIORI_Itemset_Gen (M, ϵ)
    X_1 ← {large 1 − itemsets}
    Y ← 2
    While X_{y−1} ≠ ∅
        Q_y ← {A ∪ {B} | A ∈ X_{y−1} ∧ B ∉ A} − {Q | {S | S ⊆ Q ∧ |S| = Y − 1} ⊄ X_{y−1}}
        for transactions t ∈ T
            D_t ← {Q | Q ∈ Q_y ∧ Q ⊆ t}
            for candidates Q ∈ D_t
                count[Q] ← count[Q] + 1
            end for
        end for
        X_y ← {Q | Q ∈ Q_y ∧ count[Q] ≥ ϵ}
        Y ← Y + 1
    return ⋃_y X_y
```

predicting injection attacks at step 7 of the API sequence - four steps before the point attacks can be considered successful and complete.

Efficiency and speed are critical factors in generating early warning signals at runtime. Equally important is the scalability of the system when handling large-scale malware datasets. To demonstrate the scalability and efficiency of the proposed EWS system, we have measured its computational (time) complexity. The complexity of the APRIORI algorithm depends on the number of items (malware APIs), the number of transactions (malicious chains), and the size of *itemsets*. The computational or time complexity (TC) of our early warning system arises from the complexity of Algorithm 1 plus the complexity of Algorithm 2. This combined complexity can be calculated through Eq. (4).

$$TC = O(\textit{Itemset generation}) + O(\textit{Rule generation}) \tag{4}$$

According to the procedure for *support* counting in APRIORI algorithm, each transaction of length ω produces $\binom{\omega}{k}$ *itemsets* of size k [54]. Hence, Eq. (4) can be extended to Eq. (5) for the time complexity of our early warning system:

$$TC = O\left(N \sum_{k} \binom{\omega}{k} \alpha_k\right) + O\left(\sum_{k=1}^{d-1} \left[\binom{d}{k} x \sum_{j=1}^{d-k} \binom{d-k}{j}\right]\right) \tag{5}$$

where α_k is the cost of updating the support count of a candidate k-

Algorithm 2
Rule generation.

```
APRIORI_Rule_Gen (f_k, H_m)
    for each frequent k_itemset F_k, K ≥ 2 do
        H_1 = {i | i ∈ f_k}   {1 − item consequents of the rule}
        k = |f_k|   {size of frequent itemset}
        m = |H_m|   {size of rule consequent}
        if (K > m + 1) then
            H_{m+1} = APRIORI_gen(H_m)
            for each h_{m+1} ∈ H_{m+1} do
                Conf = σ(f_k) / σ f_k − h_{m+1}
                if Conf ≥ minconf then
                    output the rule (f_k − h_{m+1}) → h_{m+1}
                else
                    delete h_{m+1} from H_{m+1}
                end if
            end for
            call APRIORI_Gen_Rule (f_k, H_{m+1})
        end if
    end for
```

itemset in the hash tree [55], d is the number of attributes, and N indicates the number of transactions in the malware dataset. Since the number of iterations is constant, the TC of the early warning system would be equal to Eq. (6).

$$TC = O(N \times d \times 2^d) + O(3^d - 2^{d+1} + 1) \tag{6}$$

Eventually, after simplification, the TC for early malware detection in our proposed method is $O(N \times d)$. In the worst case, it would be $O(N^2)$, where n indicates the number of unique APIs in the malware dataset. However, in our proposed method, the time complexity is $O(N)$ since d is constant. This analysis confirms that the proposed EWS system remains computationally feasible and scalable for large-scale malware datasets, unlike the traditional APRIORI algorithm whose complexity may become prohibitive as both N and d grow. Furthermore, we evaluated the system's performance in real-world scenarios over a 21-day operation to substantiate the durability of *DeepRadar*, as reported in Section 4.6.

3.3.3. Fast Fourier Convolutional Neural Network (FFCNN)

Dimensionality reduction, weight sharing, and local connectivity are among the features that have made CNN a popular DL method in different domains [56]. Gibert et al. in [57] have demonstrated the efficacy and robustness of convolutional neural networks (CNN) in the detection of novel malware classes. Furthermore, CNNs have also achieved significant performance in detecting various types of today's sophisticated cyber-attacks, including at least 15 attack classes in Mobile Ad Hoc Networks [58] and 9 types of modern DoS attacks in the Industrial Internet of Things [59,60].

In our work, due to the advantages of *fast Fourier convolution* introduced in [61], we decided to use it for training a model for the *Detection Subsystem* of *DeepRadar*. FFC can process large-size images faster than standard CNN, which is a key factor in dealing with malware injection attacks. It has also demonstrated superior performance in object recognition [62]. To align the extracted malware behaviour with the input of the FFCNN module, the assembly (ASM) file of each malware was converted into a 3-channel RGB 300 × 300 image based on the method described in [63] due to its higher accuracy compared to other available methods. The ASM file for each malware was obtained by reversing its binary code using IDA Pro[7] v 7.40 disassembler.

We selected RGB image conversion over grayscale or raw byte sequences for the following reasons. (a) RGB channels capture three dimensions of information for each pixel, enabling the network to learn richer feature representations and spatial patterns in malware binaries than would be possible with grayscale or raw byte inputs [64]. (b) Most CNN architectures are designed and optimised for three-channel image inputs, so using RGB makes it easier to take advantage of existing, high-performing and - and in some cases pre-trained - vision models [65]. (c) Previous studies such in [63,66] have also shown that RGB-based ASM image representations achieve higher accuracy than grayscale or raw byte approaches. (d) Finally, although RGB introduces some processing overhead, today's GPUs and processors handle this efficiently, ensuring that the method remains scalable even for large malware datasets.

After conversion from assembly language to RGB images, we used Python with PyTorch and Keras libraries to implement our neural network model. A major challenge to training this network was the lack of sufficient samples for certain malware classes, called rare malware [16]. To bridge this gap, we included a generative adversarial neural network (GAN) in the proposed system to generate additional image instances from the few samples available in these minority classes. The GAN in *DeepRadar* has partially borrowed its architecture from MIGAN, detailed in [67]. MIGAN produces images based on the Malimg[8] dataset but with a higher Inception Score compared to the original malware images. We adopted MIGAN because it has proved its remarkable efficacy by successfully synthesising 50 K malware images for training a ResNet50v2 network. We then trained the model on ml.c5.2xlarge instance type in H2O-3[9] version 3.32.

For the FFCNN network hyper-parameters, including step size (learning rate), were manually adjusted between 0.80 to 0.85, with 45 iterations, a maximum depth of {5, 10, 15, 20}, the batch size of 64, and momentum in ranges of 0.99 down to 0.80, with steady weight decay. In our experiments, the best accuracy was achieved at a depth of 9 to 10 for the neural network, while 10-fold cross-validation was employed to validate the proposed model. Softmax [68] was used as an activator function within our deep learning network, according to Eq. (7).

$$Softmax\ \sigma\left(\vec{Z}^{\rightarrow}\right)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \tag{7}$$

in which $\vec{Z}_i^{\rightarrow}$ values indicate elements of the input vector, and k in-

[7] https://hex-rays.com/ida-pro/

[8] https://www.kaggle.com/datasets/manmandes/malimg

[9] https://github.com/h2oai/h2o-3

dicates the number of classes. In this equation, e^{z_i} is the standard exponential function that is applied to input vector elements, and finally, $\sum_{j=1}^{k} e^{z_j}$ is the normalisation term. Other hyper-parameters, including multiplications matrix dimensions, the number of neurons, and epochs, were set and tuned by H2O-3 using a grid search and Gradient Descent optimisation approaches. With a new architecture, as shown in Fig. 5, this is the first time FFCNN has been leveraged for malware recognition.

3.4. The runtime scanner

The architecture of *DeepRadar's* runtime canner consists of three main subsystems: Validation, Detection, and Confronting subsystems, as shown in Fig. 6. Each subsystem and its workflow are elaborated on in the following sections, respectively.

3.4.1. Validation subsystem

As shown in Fig. 6, the input of the scanner system is the address of a binary file or a running process ID (PID). The address is delivered to the *Whilelist Check* module in the *Validation Subsystem*. This module tasks to distinguish the legitimate use of code injection by Windows modules for backward capability as well as debugger applications for debugging a faulty program. It stores the SHA-1 signature of Windows modules and provides an actual user with the capability of including any benign application in its whitelist. Then, the *Accessibility Check* module performs the task of checking the accessibility of input files or processes. If a file or process is inaccessible, the file path or PID is given to the *File System Filter Driver* or *Process Filter Driver* to make it accessible. This strategy enables the scanner to function beyond malware patch guards that might have been installed in the kernel space of the OS. The file is then delivered to the *PE Validation Check* module to verify that the input is a valid PE32 or PE32+ (64) in *.exe*, *.dll,* or *.sys* formats. Therefore, the scanner can scan input files so that non-PE files are excluded.

Files with valid PE32 and PE32+ structures are delivered to the *Packing Status Check* module, which determines whether or not a file is packed and, as a result, obfuscated. The first test is performed by calculating the entropy of the code section inside the PE file [16]. When entropy-based detection is insufficient, the module can also recognise the signatures of a wide range of packing tools, such as ASPack, NSPack, UPack, Themida, and UPX, using N-gram signature according to method described in [69]. If the input file is not packed, it is forwarded to the first layer of the *Detection Subsystem*. Otherwise, it is passed to the *Packer Detector* module to identify the type and name of the packing tool used.

If the *Packer Detector* successfully identifies the packer, the file is returned to the *Static Unpacking* module, where a suitable unpacking algorithm is applied to normalise the file. The unpacked file then re-enters the scanning process at the first layer of the *Detection Subsystem*. If, however, the packing tool cannot be identified or static unpacking fails, the file is delivered to the *Dynamic Unpacking* module. In this module, the file is executed in a controlled environment (in our experiments, Cuckoo Sandbox[10] and a VM under VirtualBox, where its memory is dumped and a report of malware API calls is generated in JSON format, as described in the following section. The extracted information is rewritten into a new PE file, producing an unpacked version of the original. This file is then passed to the Detection Subsystem for a hybrid scan.

3.4.2. Dynamic unpacking and de-obfuscation

The main difficulty in detecting novel malware, particularly through static analysis, arises from the obfuscation techniques employed by packers. This is the most common method used to evade both manual and automated analysis, protecting malicious code from detection by AV, IDS, and EDR systems. Packers typically apply reversible algorithms to compress, modify, or encrypt binary code, rendering it unintelligible to analysts [70]. Fig. 7 illustrates how the header of a PE structure has been fully obfuscated by NSPack v3.7. We used the PeID v0.95 and DIE v3.02 tools for this and subsequent experiments. As shown in Fig. 7, standard PE sections generated by compilers, such as *.code, .text*, and *.data*, are replaced with unknown sections. This prevents the extraction of essential features for behavioural modelling, including the names, numbers, sizes, and offsets of sections [41].

Another common obfuscation technique is the encryption of the IAT and EAT tables, where DLL and API names are stored. This disrupts many malware detection methods, which rely on intercepting API and system calls to infer the behaviour of a binary. By encrypting these tables, malware conceals its system API calls, preventing the disclosure of its malicious behaviour and intent. Fig. 8 shows a malware sample that applies IAT encryption to hide its APIs. Control-flow graph (CFG) obfuscation is another widely used technique. It commonly involves the insertion of junk or dead code, false conditions, NOPs, and fake jumps. These modifications aim to hinder reverse engineering of the binary, particularly its disassembly into assembly code, therefore hinder many malware detection methods [71,72].

DeepRadar is capable of dynamic unpacking via memory dumping. Packers must eventually unpack an application into memory at runtime in order for execution to proceed. This necessity provides an opportunity for analysis: by capturing the allocated memory of a process at carefully chosen moments, it is possible to reconstruct an unpacked version of the executable and recover structural features that are otherwise hidden by obfuscation. Our method therefore employs a dynamic memory dumping strategy to normalise the behaviour of packed malware and enable subsequent static and hybrid analysis.

In particular, our approach targets four critical points in the malware execution timeline: (1) immediately after *CreateProcess* is called and before the malware reaches its Original Entry Point (OEP), when the *.idata* section is unpacked and the IAT/EAT tables are accessible; (2) during calls to *AdjustTokenPrivilege*, when the packer unpacks executable instructions into the code section, making it possible to capture them; (3) just after *VirtualAllocEx* is invoked, when additional memory is allocated for the expansion of compressed code; and (4) immediately before *TerminateProcess*, when some malware attempts self-destruction or repacking, allowing the recovery of otherwise lost sections. Fig. 9 shows the timeline of the four stages of memory dumping in our dynamic unpacking method.

The dumps collected at these points are rewritten into valid PE file format using PROCEXEDUMP and PROCMEMDUMP commands of Volatility 2.5[11] and merged into a single PE file using IDA Pro v7.40. Although dynamic unpacking is slower than static methods and requires a controlled environment, e.g. sandbox or virtual machine, it avoids the need for packer-specific algorithms and decryption key and is therefore more generic. Our experiments demonstrate that the method is effective against a wide range of packer and protector tools, including customised and previously unseen (zero-day) variants. By capturing and preserving unpacked states throughout execution, this approach significantly improves the ability to analyse highly obfuscated malware samples.

3.4.3. Detection subsystem

The proposed detection modules were integrated into a three-layer Detection Subsystem in *DeepRadar's* architecture for early detection and defusing of injection attacks: N-gram Search (L1), Static Scan (L2), and Dynamic Scan (L3). The Detection Subsystem starts the first scanning layer (L1) by running a swift 4-gram search to find unique subsequences of already known injector stubs, making the detection possible without deep examination of the input. If any malicious injection stub is detected, its path will be delivered to the *Confronting*

[10] https://github.com/cuckoosandbox

[11] https://github.com/volatilityfoundation/volatility

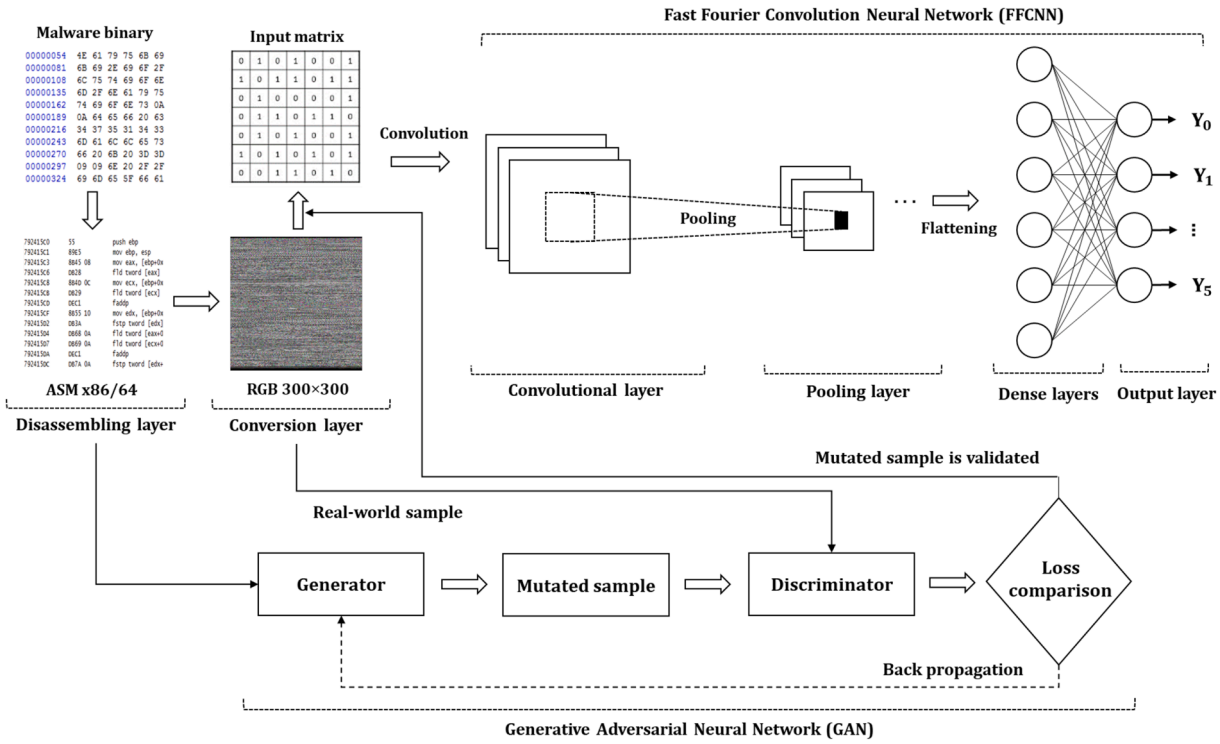


Fig. 5. The structure of the proposed neural networks to classify malware in *DeepRadar*.

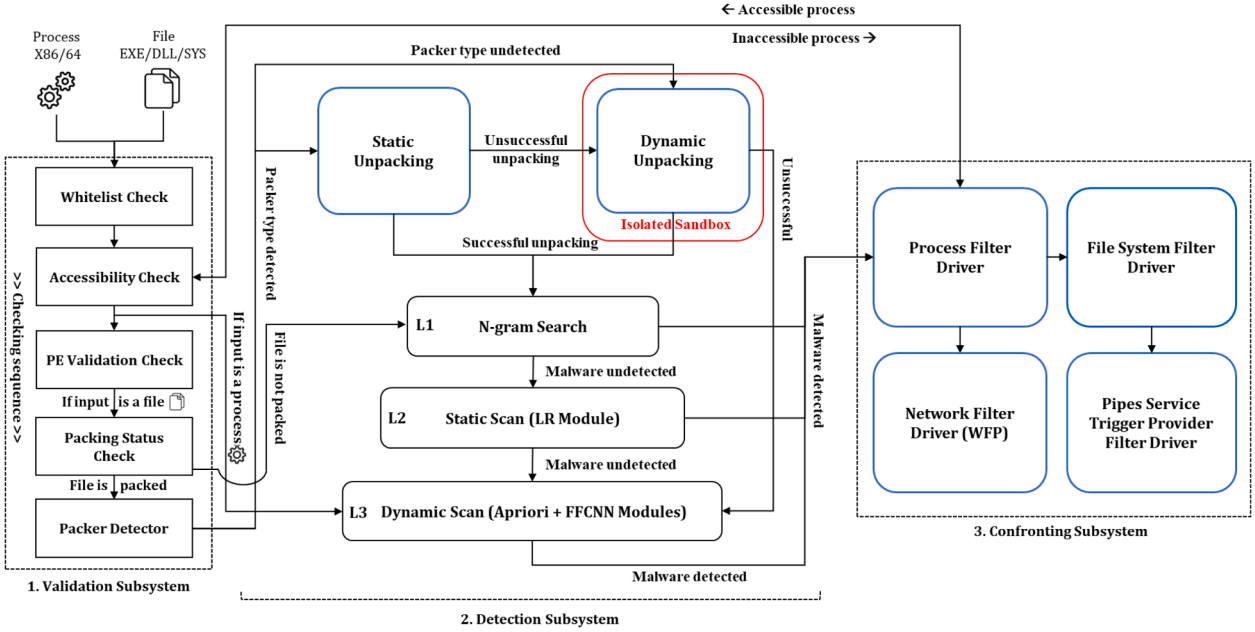


Fig. 6. The main architecture of *DeepRadar's* runtime scanning system.

Subsystem. Otherwise, the file is delivered to the second scanning layer (L2), which is the *Static Scan*. In this layer, the LR classifier is responsible for malware detection, which performs merely by examining the properties of the PE's structure without executing the file. During this process, the API names are extracted from the IAT of the PE header, and if malicious chains are found, the file path will be delivered to the *Confronting Subsystem* to forcefully stop the malware activity and eliminate its processes and source files. L1 and L2 layers have been tailored for rapid and efficient malware detection capable of injection attacks, aiming to save time and resources. However, they are vulnerable to obfuscation and evasion techniques, particularly IAT Encryption, Polymorphism, Control Flow Graph (CFG) smashing, Code-reuse attacks, and Runtime Code Generation [32]. To tackle this obstacle, the third layer (L3), *Dynamic Scan*, tracks IRPs and API calls at runtime (dynamic analysis) using kernel-level filter drivers and detects injection attacks by APRIORI and FFCNN models. Even the extremely obfuscated malware samples are unlikely to evade this layer as they must reveal their behaviour at runtime.

The input L3 layer can be either a process resulting from the accessibility check performed in the Validation Subsystem or a file from the

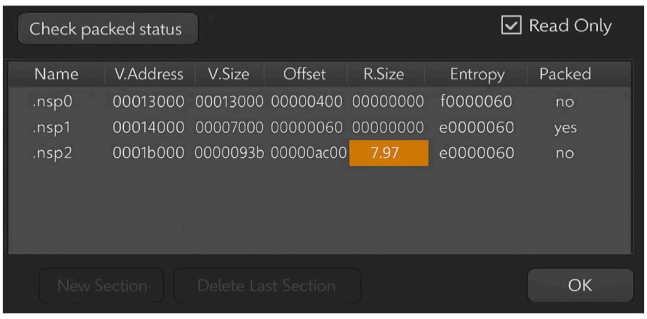


Fig. 7. Sample PE header of an obfuscated and packed malware binary, analysed with DIE on REMnux.

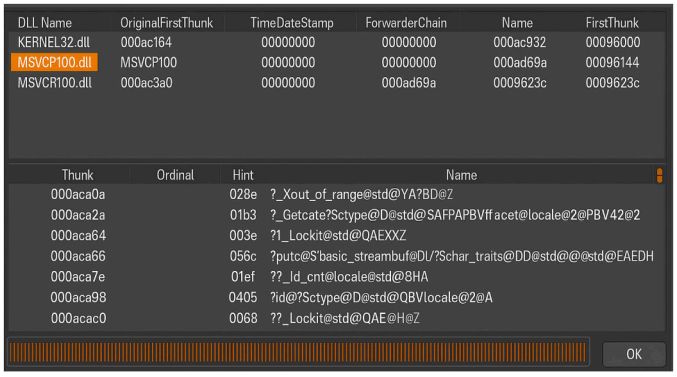


Fig. 8. IAT table of an obfuscated malware sample with encrypted API names, analysed with DIE on REMnux.

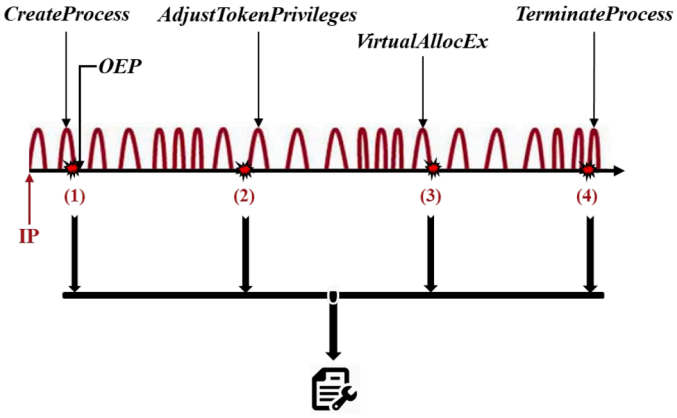


Fig. 9. Timeline of memory dumping in the proposed approach, where IP indicates the instruction pointer.

L2 layer. If the input is a file, it is executed in an isolated and controlled sandbox - or a virtual machine equipped with filter drivers. Filter Drivers install kernel-level hooks to intercept IRPs and APIs alongside their relevant invocation parameters. This essentially makes use of the features of the APRIORI classifier to predict or detect malicious chains of injection attacks. The settings of these filter drivers, such as Group Order and Altitude, were adjusted to 36,000 towards the base of the I/O stack. DeepRadar employs kernel-level hooks due to the high analysis depth and transparency [69], which allows for the analysis of active malware in both user and kernel spaces. If L3 does not detect any malicious chains, the system will regard the input as benign; however, if classifiers detect malicious behaviour, the file or process will be immediately reported to the Confronting Subsystem to defuse the attack. At this stage, Inter-Process Communications (IPC), including shared memory, file mapping, message queue, and pipelines, are also tracked. This enables *DeepRadar* to work independently from the programming language used for developing the malware.

3.4.4. Confronting subsystem

The *Confronting Subsystem* takes advantage of four kernel-level filter drivers, i.e., *Process Filter Driver, File System Filter Driver, Network Filter Driver*, and *Pipe Service Filer Driver* to track various malware activities and remove different pieces of malware from the system, as demonstrated in Fig. 6. The confronting action starts with *Process Filter Driver* module tracking the malware's process to obtain the path of the malware's source file. Then, the *Process Filter Driver* uses *ZwTerminateProcess* kernel routine to forcefully terminate the malware's process(es). At the same time, PID is received by a *Network Filter Driver* created by the Windows Filtering Platform (WFP) facility. This filter driver traces and interrupts all network connections established by the malware. After the *Process Filter Driver* completes its task, a signal is sent to the *File System Filter Driver*. This filter driver is responsible for eliminating malware files from hard disks by sending direct *IRP_MJ_Close* and *IRP_MJ_CLEANUP* requests to the I/O Manager. These IRPs close any open handles associated with the malware file. Next, the filter driver sends *IRP_MJ_SET_INFORMATION* request with the *FileDispositionInformation* parameter to eliminate files associated with the detected malware. Lastly, *Pipe Service Filer Driver* uninstalls all pipelines and services created by the malware. Since IRPs have the highest privilege level to eliminate files in the OS, malware defensive path guards cannot stop sending IRPs except in the case of hypervisor-level and hardware-level malware programs. We have provided the list of system functions intercepted by *DeepRadar* along with their details from [73,74] in Table A-1 in Appendix.

3.5. Defusing the attacks and self-defence

Defusing or blocking injection attacks is another important innovation of our proposed method, making *DeepScan* robust. Self-defence was not meticulously explored in the literature. There are some related works [75,76] that rely on the discrepancy between the number of libraries loaded in the allocated space of a program at runtime and that of required libraries (already placed in the PE header by the compiler). In such cases, if the number of libraries loaded in the process memory exceeds the required number, it can then be concluded that a library injection attack has been attempted. Defusing such attacks requires flushing the injected codes and unloading fake libraries from memory dedicated to the victim process without causing interruptions.

Our approach can discover the victim process immediately after injection is commenced (before completion) and locate the fake library in the memory allocated to the victim process by interpreting the parameters of APIs used by the malware. To this aim, *DeepRadar* restricts some system functions that can be abused by malware programs. Function restrictions are administered by installing a *Pre-Callback* kernel-level hook on the *CreateRemoteThread* function. This type of hook makes it possible for a Callback function to be called before calling the primary function. In this way, the scanner is given the opportunity to prevent calling *CreateRemoteThread* if any attacks are detected. However, it is also possible that malware targets the proposed scanner program for code and library injection to hijack the program's execution control. Therefore, the proposed system should be able to deal with active malware and defend itself against injection attacks so that it operates correctly in an infected environment. To protect DeepRadar's running process and files against injection attacks, we coded a Pre-Callback function to check the first parameter of the function (i.e., *HANDLE hProcess* of the *CreateRemoteThread)* to find out if a handle to the scanner's process exists. If so, the attack could be halted by prohibiting incoming IRPs from calling *CreateRemoteThread*. Therefore, the attacker would never be able to create a remote thread in the scanner's process, and therefore, the injection chain of Fig. 3 will be halted at step 7.

4. Evaluation and comparison

In this section, we evaluate *DeepRadar* from multiple perspectives and across various real-world scenarios, focusing on its accuracy and robustness, early detection and self-defence capabilities, as well as resource efficiency. All experiments were conducted on an x64 Intel Xeon E5–2620 2.4 GHz machine with 12 logical cores, 2 GeForce GTX 1080 Ti GPU, 32 GB of RAM DDR5 1866 MHz, and 1 TB of SDD memory. Windows 10 × 64 was the host OS. Eight VMs were run on a VM-Ware virtualisation hypervisor for conducting experimental experiments. Each VM had 4 cores of CPU and 4 GB of RAM. *DeepRadar* and its relevant modules were implemented in *C++/C#* programming language, *C++* for kernel-level modules and C# for the user interface. The full list of tools used in this research with details and access links can be found in Table A-2 in Appendix A.

4.1. Dataset composition and characteristics

We created a dataset comprising 41,331 instances in total, including 31,531 malware samples across seven classes and 9800 benign files. Malware samples with the capability of code or library injection were collected from 2018 to 2025, inclusively from Adminus [43], VirusShare [44], VirusSign [45], and MaleVis [46] datasets. These included diverse classes such as infectors, evaders, spyware, rootkits, and banking trojans. All malware samples from these sources were heavily obfuscated: 100% were packed with common packers such as UPX, Themida, PECompact, FSG, MPRESS, and UPack as well as customised and unknown packers. These packers employed a wide range of obfuscation and evasion techniques, such as polymorphism, junk/dead code injection, control-flow obfuscation, and IAT encryption. As a result, *DeepRadar's* unpacking module was engaged in the analysis of every malware sample.

The benign batch was formed from software applications from Softpedia[12] and SourceForge[13] repositories. To make the test more rigorous, 50% of the benign files were randomly packed to ensure that *DeepRadar* could correctly distinguish legitimate commercial applications that use packers to prevent reverse engineering or to preserve copyright of their intellectual property. Accordingly, the output layer in our FFCNN model consisted of seven neurons to detect and classify six malware classes and a benign class. Table 1 presents the distribution of classes and details of dataset diversity, including the number and percentage of each malware class as well as their packing status.

To further assess the generalisation ability of *DeepRadar*, we analysed the subset of injection-capable emerging malware variants within our dataset. This subset comprised 2530 samples - 8% of the malware category - collected between 2023 and 2025, representing novel or less-studied families that approximate zero-day conditions. It was isolated during evaluation to test the robustness of *DeepRadar* against previously unseen injection-based variants.

To evaluate the robustness of *DeepRadar* against adversarial malware, we generated adversarial injection samples using the Metasploit Framework v6.4 under Kali[14] and Parrot[15] Linux distributions. We employed payloads supporting injection attacks for both x64 and x86 architectures, *meterpreter/reverse_tcp, reverse_http, reverse_https, shell_reverse_tcp,* combined with common encoders such as *Shikata-ga-nai, xor, countdown, call4_dword_xor*, applied for 1–5 rounds across a grid of 20 base payload combinations × 9 encoder settings × 3 templates. To further simulate evasion, these samples were additionally packed with packers including UPX, Themida, ASPack, NsPack, PECompact, FSG. For each case, the packer was randomly selected from a repository of 300 packers available to us.

All adversarial binaries were processed through the same disassembly → ASM → RGB pipeline as the clean injection samples. In addition, to better approximate real-world adversarial conditions, we applied small-budget perturbations with a magnitude of $\varepsilon \leq 4/255$ - equivalent to at most 1.6% of the pixel value range - to the RGB image representations of malware samples using both the Fast Gradient Sign Method (FGSM) and Projected Gradient Descent (PGD) attacks. FGSM serves as a fast, single-step baseline attack, whereas PGD is a stronger, iterative approach that poses a more challenging adversarial scenario.

These degradations (encoder + packer chains and gradient-based perturbations) enable us to assess the robustness of our model in real-world scenarios where *DeepRadar* faces with zero-day injection attacks, adversarial samples as well as noises and perturbation at the feature-representation level.

4.2. Benchmark pool

Several AVs were selected to create a standard benchmark group for comparison. It is important to note that we were unable to access the source code or executable version of the related works discussed in Section 2, and as a result, we could not include them in the benchmark group for the subsequent real-world test scenarios. However, we have conducted several comparisons with the related works in Section 4.5. Table 2 lists the names and the dates of the latest updates of the AVs used in our benchmark. These tools were chosen based on the scores reported by AV-Test[16] for 2024.

4.3. Performance measures

The accuracy of detecting code and library attacks was the primary aim that reflects how *DeepRadar* successfully protects against malware programs capable of code and library injection. To evaluate the accuracy of our proposed system, a 70–15–15 train-validation-test split of the mixed dataset was used. Then, each of *DeepRadar's* detection modules (i.e., LR and FFCNN) and the benchmark group were tested on the same test dataset and condition for generating performance figures. The detection accuracy of each tool (Accuracy) and the misclassification rate (Error) were then computed using Eq. (8) [77].

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{8}$$

in which TP (true positive) indicates the number of malware samples that have been accurately detected as malware, TN (true negative) indicates the number of benign samples that have been accurately identified as benign. FP (false positive) indicates the number of benign files that have been misdetected as malware, and FN (false negative) indicates the number of malware samples that have been misdetected as benign. F-score (F1-score) was also reported to study the harmonic average of the Precision and Recall since the dataset is imbalanced [36] due to the existence of certain rare classes of malware. Eq. (9) refers to the calculation of F-score.

$$F - score = 2 \times \frac{Recall \times Precision}{Recall + Precision} \tag{9}$$

where Precision and Recall - also known as true positive rate (TPR) - are calculated using Eq. (10) and Eq. (11), respectively.

$$Precision = \frac{TP}{TP + FP} \tag{10}$$

$$Recall = \frac{TP}{TP + FN} \tag{11}$$

[12] https://win.softpedia.com/
[13] https://sourceforge.net/
[14] https://www.kali.org/get-kali/#kali-platforms
[15] https://parrotsec.org/download/
[16] https://www.av-test.org

Table 1
Distribution of classes in the mixed dataset used for the experiments.

#	Category	Class	# Samples	# Category	Packing status	% Category	% All
1	Malware	Spyware	10,059	31,531	Packed (100%)	31.9%	24.3%
2		Banking trojan	8187		Packed (100%)	26.0%	19.8%
3		Binder	3624		Packed (100%)	11.5%	8.8%
4		Evader	2914		Packed (100%)	9.2%	7.1%
5		Rootkit	2517		Packed (100%)	8.0%	6.1%
6		Metasploit	1700		Packed (100%)	5.4%	4.1%
7		Emerging injection variants (2023–2025)	2530		Packed (100%)	8.0%	6.1%
8	Benign	System binaries	3800	9800	Packed (50%)	38.8%	9.2%
9		Application binaries	6000		Packed (50%)	61.2%	14.5%

Table 2
The benchmark group used for comparison in the experimental scenarios.

#	Index	Anti-malware program	Update
1	FFCNN	DeepRadar - FFCNN module	2025-Q3
2	LR	DeepRadar - LR module	2025-Q3
3	AV1	Kaspersky	2025-Q3
4	AV2	McAfee	2025-Q3
5	AV3	Eset Node 32	2025-Q3
6	AV4	ClamAV	2025-Q3
7	AV5	Panda	2025-Q3
8	AV6	Sophos	2025-Q3
9	AV7	Dr. Web	2025-Q3

We also explored receiver operating characteristic (ROC) curves and AUC values for the conducted experiments. An ROC plots TPR against FPR for a certain range. ROC curves assist in studying the trade-off between true positive rate (TPR) and false positive rate (FPR). Each ROC has a corresponding numeric value of the area under the curve (AUC) and can be computed using Eq. (12). AUC is used to evaluate and indicate the stability of created models [78,79].

$$AUC = \int_0^1 \frac{TP}{TP + FN} \, d \, \frac{FP}{TN + FP} \tag{12}$$

To evaluate the proposed early warning system, we used success rate (SR), indicating the ratio of the defused attacks by this module. SR was calculated according to Eq. (13).

$$SR = \frac{Number\ of\ Defused\ Attacks}{Total\ Number\ of\ Attacks} \tag{13}$$

Lastly, the averaged value for Accuracy, and other metrics such as F-score, AUC, and Error were calculated using Eq. (14).

$$Avg\ Accuracy\ = \frac{\sum_{i=0}^{N} Accuracy_i}{N} \tag{14}$$

where N indicates the total number of classes and Accuracy is replaced for computing the average values for other metrics.

4.4. Experimental design and results

The performance and efficacy of *DeepRadar* in detecting and blocking injection attacks were put to the test through two experimental scenarios. Scenario 1 covers the performance of trained models, i.e., LR and FFCNN modules. This scenario does not study active attacks that have already been started; therefore, the performance of the early warning system is outside the scope of Scenario 1. Scenario 2 was dedicated to the evaluation of *DeepRadar's* early warning system. This scenario reveals how many active attacks were detected and defused (blocked) before completing their mission.

4.4.1. Scenario 1: performance evaluation of detection models

This scenario focuses on the classification performance of *DeepRadar* by putting LR and FFCNN classifiers to test and comparing their performance with the benchmark group. As mentioned before, the LR model was developed as a light scanner to detect unpacked malware instances. The FFCNN model deals with more complicated classes of obfuscated and packed malware and activates if malware passes the LR test.

We ran LR, FFCNN, and other benchmark tools. The percentage of Accuracy, F-score, AUC, and Error for LR and FFCNN models, as well as the benchmark group for every malware class, were presented in Fig. 10. To provide a detailed comparison, numeric values were presented in Table 3.

As shown in Table 3, the proposed FFCNN model consistently outperformed all methods in detecting diverse classes of malware capable of injection attacks, including emerging variants as well as adversarial samples generated with Metasploit. While AV1 obtained competitive results in Accuracy, F-score, and AUC for families such as Spyware, Binder, Banking, Evader, and Emerging variants, and AV2 showed relative strength on Rootkit, Metasploit, and benign detection, neither matched the overall robustness of our approach.

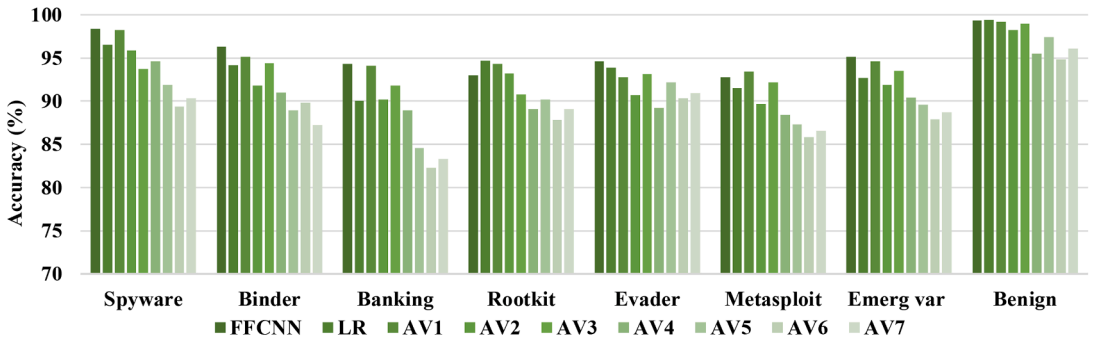
In particular, our FFCNN achieved an Accuracy of 95.1% and an F-score of 95.7% on emerging injection-capable malware samples, comparable to its performance on well-represented families. On adversarial Metasploit samples, the model reached 92.8% Accuracy and an F-score of 93.9%, maintaining resilience despite deliberate evasion and obfuscation. These results confirm that the proposed method sustains strong detection capability even under zero-day scenarios, underscoring its generalisation ability.

Moreover, the findings demonstrate that *DeepRadar* preserves robust performance against adversarial injection attacks combined with packer-based obfuscation, reinforcing its reliability as a practical malware early-warning system. Taken together, these results show that under heavy obfuscation - 100% of malware packed; 50% of benign packed - and explicit adversarial stress - Metasploit encoders and packers, FFCNN provides the most reliable detection across injection-oriented malware, with consistent gains over LR and all AV baselines in Accuracy, F-score, and AUC. For a holistic comparison, Fig. 11 presents the average Accuracy, F-score, and AUC results from this experiment.

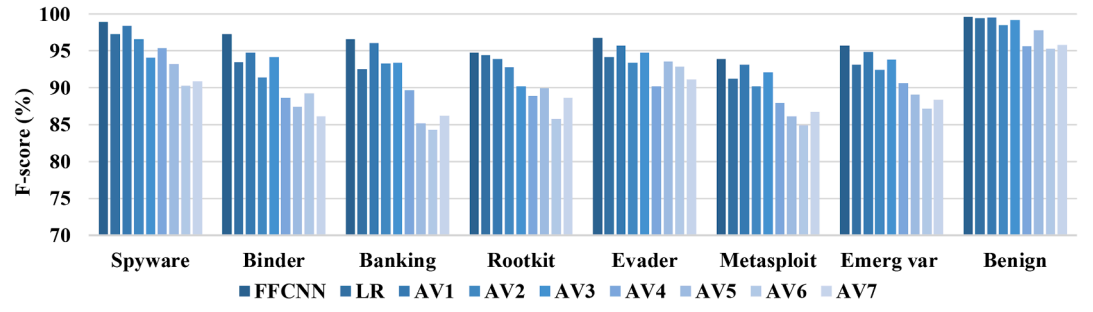
The ROC curves and corresponding AUC values are presented in Fig. 12 for our proposed models and the benchmark group. Analysis shows that FFCNN and AV1 achieved the highest performance, with AUC values of 0.925 and 0.921, respectively. Although the margin between them is small, FFCNN ranked first overall. In contrast, AV6 and AV7 recorded the lowest AUC values among all models, highlighting their weaker discrimination ability.

From the ROC curves in Fig. 12, Evader and Rootkit appear as the most challenging classes in ROC space, with FFCNN AUC values of 0.876 and 0.889, respectively. Nevertheless, their F-scores remained high, with Evader reaching 0.968 and Rootkit 0.943, reflecting a strong precision-recall balance despite more difficult threshold behaviour.

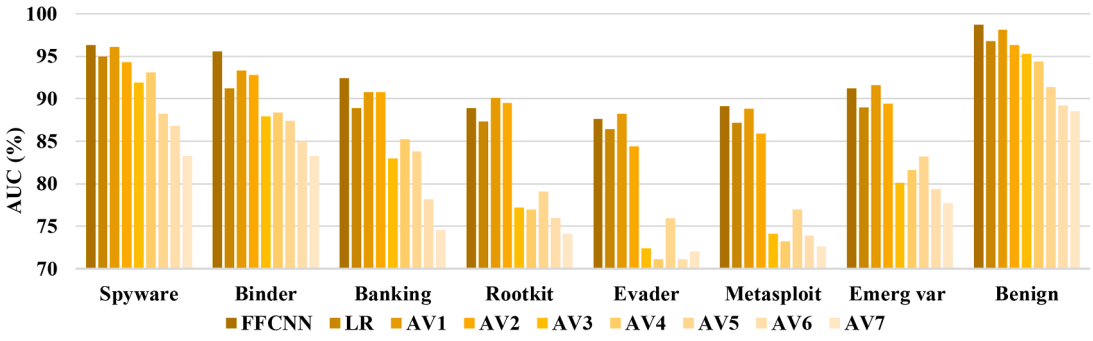
For a formal verification, we statistically tested the significance of difference between the resulting figures presented in Table 3. Initially, we conducted the Friedman test to study the significance of difference



(a) in terms of Accuracy.



(b) in terms of F-score.



(c) in terms of AUC.

(d) in terms of Error.

Fig. 10. Comparison of results between *DeepRadar* and the benchmark group.

between averaged values for different detection strategies [80]. Friedman test rejected the null hypotheses for Accuracy, F-score, and AUC considering the significance level of 0.05. It indicates that there is a significant difference among these measures for different models used in this scenario. Wilcoxon signed-rank test (for a significance level of 0.05) was then used as post hoc analysis to statistically explore the results in detail [81]. According to test results, FFCNN performs significantly better than LR, AV2, AV3, AV4, AV5, AV6, and AV7. The test rejected the significance of difference between FFCNN and AV1 in terms of Accuracy. Similarly, although LR has better mean Accuracy compared to AV2, the test rejected the significance of difference in this case too. LR significantly resulted in a higher Accuracy compared to AV4, AV6, and AV7; while it was significantly less accurate than FFCNN and AV1. The interpretation of the test results for the Error is the same as Accuracy.

Table 3
The Accuracy, F-score, AUC, and Error values resulted from the experiment.

Metric	Class	FFCNN	LR	AV1	AV2	AV3	AV4	AV5	AV6	AV7
Accuracy	Spyware	**0.984**	0.965	0.982	0.959	0.937	0.946	0.919	0.894	0.903
	Binder	**0.963**	0.942	0.951	0.918	0.944	0.910	0.889	0.898	0.872
	Banking	**0.943**	0.900	0.941	0.902	0.918	0.889	0.846	0.823	0.833
	Rootkit	0.930	**0.947**	0.943	0.932	0.908	0.891	0.902	0.878	0.891
	Evader	**0.946**	0.939	0.928	0.907	0.931	0.892	0.922	0.903	0.909
	Metasploit	0.928	0.915	**0.934**	0.897	0.922	0.884	0.873	0.858	0.866
	Emerg var	**0.951**	0.927	0.946	0.919	0.935	0.904	0.896	0.879	0.887
	Benign	0.993	**0.994**	0.992	0.982	0.990	0.955	0.974	0.948	0.961
Average		**0.955**	0.941	0.952	0.927	0.936	0.909	0.903	0.885	0.890
F-score	Spyware	**0.989**	0.973	0.984	0.966	0.941	0.954	0.932	0.903	0.909
	Binder	**0.973**	0.935	0.948	0.914	0.942	0.886	0.874	0.892	0.861
	Banking	**0.966**	0.925	0.961	0.933	0.934	0.897	0.852	0.843	0.862
	Rootkit	0.943	**0.944**	0.939	0.928	0.902	0.889	0.899	0.858	0.886
	Evader	**0.968**	0.942	0.957	0.934	0.948	0.902	0.936	0.929	0.911
	Metasploit	**0.939**	0.912	0.931	0.902	0.921	0.879	0.861	0.849	0.867
	Emerg var	**0.957**	0.931	0.949	0.924	0.938	0.906	0.891	0.872	0.884
	Benign	0.995	0.994	**0.996**	0.985	0.992	0.956	0.978	0.953	0.958
Average		**0.966**	0.944	0.958	0.936	0.940	0.909	0.903	0.887	0.892
AUC	Spyware	**0.963**	0.950	0.961	0.943	0.919	0.931	0.882	0.868	0.833
	Binder	**0.956**	0.912	0.933	0.928	0.879	0.884	0.874	0.850	0.833
	Banking	**0.924**	0.889	0.908	0.908	0.830	0.852	0.838	0.782	0.746
	Rootkit	0.889	0.873	**0.901**	0.895	0.772	0.770	0.791	0.760	0.741
	Evader	0.876	0.864	**0.882**	0.844	0.724	0.711	0.759	0.711	0.720
	Metasploit	**0.891**	0.872	0.888	0.859	0.741	0.732	0.770	0.739	0.726
	Emerg var	0.912	0.890	**0.916**	0.894	0.801	0.816	0.832	0.794	0.777
	Benign	**0.987**	0.968	0.981	0.963	0.953	0.944	0.914	0.892	0.885
Average		**0.925**	0.902	0.921	0.904	0.827	0.830	0.832	0.800	0.783
Error	Spyware	0.017	0.040	**0.015**	0.034	0.062	0.053	0.080	0.105	0.096
	Binder	**0.036**	0.057	0.048	0.081	0.055	0.089	0.110	0.101	0.127
	Banking	**0.057**	0.089	0.058	0.097	0.081	0.110	0.153	0.176	0.166
	Rootkit	0.069	**0.052**	0.056	0.067	0.091	0.108	0.097	0.121	0.108
	Evader	**0.053**	0.060	0.071	0.092	0.068	0.107	0.077	0.096	0.090
	Metasploit	0.072	0.085	**0.066**	0.103	0.078	0.116	0.127	0.142	0.134
	Emerg var	**0.049**	0.073	0.054	0.081	0.065	0.096	0.104	0.121	0.113
	Benign	0.006	**0.005**	0.007	0.017	0.009	0.044	0.025	0.051	0.038
Average		**0.045**	0.058	0.047	0.072	0.064	0.09	0.097	0.114	0.109

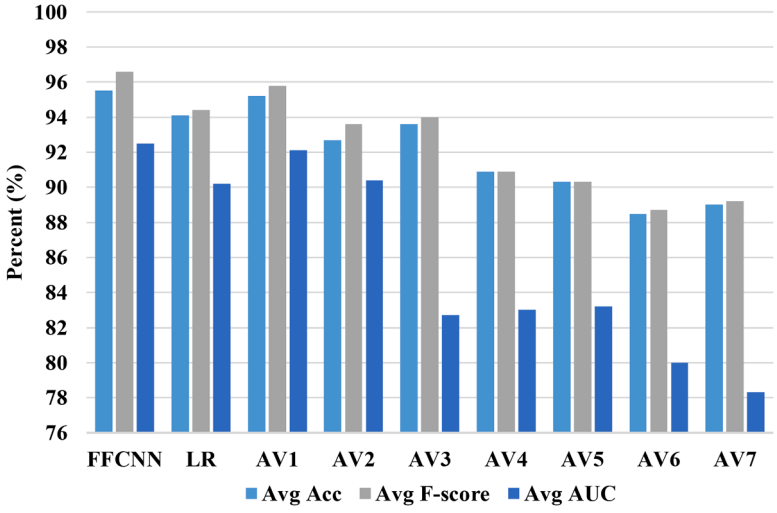


Fig. 11. Comparison between the proposed models and the benchmark group in terms of averaged Accuracy, averaged F-score, and averaged AUC.

FFCNN significantly overtook LR, AV2, AV3, AV4, AV5, AV6, and AV7, considering the F-score measure. The test showed that F-score values for FFCNN, AV1, LR, and AV2 are not significantly different. The performance of the FFCNN injection detection model was not significantly challenged by other benchmark models, and therefore, FFCNN, AV1, LR, and AV2, could be considered high-performing models for detecting injection attacks. The Wilcoxon signed-rank test shows that the FFCNN model significantly results in higher AUC values compared with all other models except AV1. Therefore, in this experiment, FFCNN and AV1 are the top models with significantly better AUC. However, the performance of other AVs and LR model for AUC values is significantly lower than FFCNN and AV1. The difference in AUC for LR and AV2 is not significant, and LR performs significantly better than AV3, AV4, AV5, AV6, and AV7. Table 4 presents Wilcoxon signed-rank test results for FFCNN and LR models considering Accuracy, F-Score, and AUC measures, respectively.

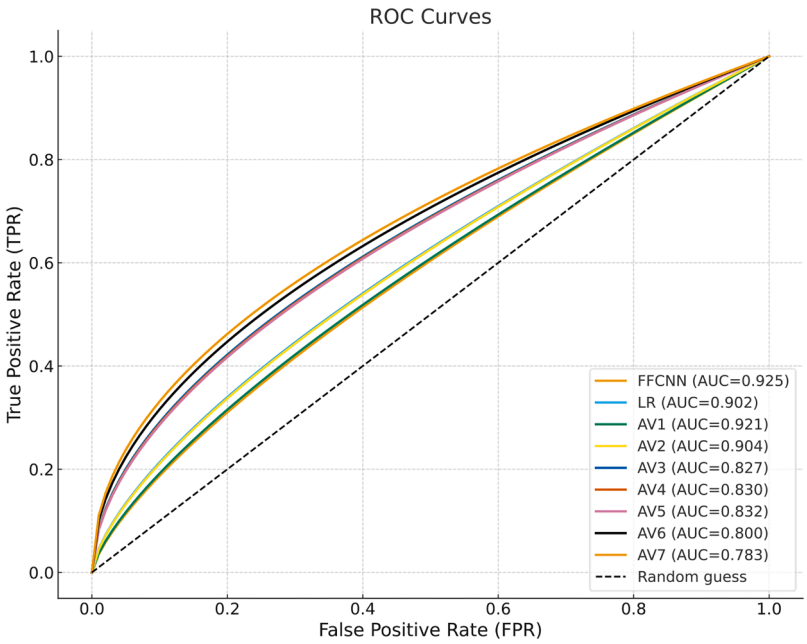


Fig. 12. ROC curves and AUC values for the proposed models and benchmark group.

Table 4
Wilcoxon Signed Ranks Test of Accuracy, F-score, and AUC for FFCNN and LR models.

			FFCNN	LR	AV1	AV2	AV3	AV4	AV5	AV6	AV7
Accuracy	FFCNN	Z	-	−2.201[a]	−0.631[b]	−0.734[a]	−1.782[a]	−2.207[a]	−2.201[a]	−2.201[a]	−2.201[a]
		Asymp. Sig. (2-tailed)	-	**0.028**	0.528	0.463	0.075	**0.027**	**0.028**	**0.028**	**0.028**
	LR	Z	−2.201[a]	-	−1.992[a]	−1.572[a]	−0.210[a]	−2.201[b]	−1.782[b]	−2.201[b]	−1.992[b]
		Asymp. Sig. (2-tailed)	**0.028**	-	**0.046**	0.116	0.833	**0.028**	0.075	**0.028**	**0.046**
F-score	FFCNN	Z	-	−2.207[b]	0.000[c]	−1.782[b]	−2.201[b]	−2.201[b]	−2.201[b]	−2.201[b]	−2.201[b]
		Asymp. Sig. (2-tailed)	-	**0.027**	1.000	0.075	**0.028**	**0.028**	**0.028**	**0.028**	**0.028**
	LR	Z	−2.207[b]	-	−2.023[b]	−1.472[b]	−0.105[b]	−2.201[c]	−1.992[c]	−2.201[c]	−2.201[c]
		Asymp. Sig. (2-tailed)	**0.027**	-	**0.043**	0.141	0.917	**0.028**	**0.046**	**0.027**	**0.028**
AUC	FFCNN	Z	-	−2.201[b]	−0.841[b]	−1.992[b]	−2.201[b]	−2.207[b]	−2.201[b]	−2.207[b]	−2.201[b]
		Asymp. Sig. (2-tailed)	-	**0.028**	0.400	**0.046**	**0.028**	**0.027**	**0.028**	**0.027**	**0.028**
	LR	Z	−2.201[b]	-	−2.201[b]	−1.572[b]	−0.524[c]	−2.201[c]	−2.201[c]	−2.201[c]	−2.201[c]
		Asymp. Sig. (2-tailed)	**0.028**	-	**0.028**	0.600	**0.028**	**0.028**	**0.028**	**0.028**	**0.028**

[a] . Based on positive ranks.
[b] . Based on negative ranks. "-" indicates a not applicable test.

4.4.2. Scenario 2: generating early warning signals

This experiment put *DeepRadar's* early warning mechanism to the test and measures its robustness against malware evasion techniques in realistic conditions. It focuses on the detection of incomplete injection chains and the capability of blocking and defusing attacks prior to completion. As mentioned earlier, SR was used as the performance measure. To conduct this scenario, a pool of malware samples with the capability of code and/or library injection was created using a wide range of injector tools. Some of these tools and malware programs were able to target and inject into the body of PE binaries stored on the hard drive (these victim files were selected randomly from the system drive), while others were able to inject into the memory of running processes. The results are presented in Table 5 in terms of the number of attacks, the number of defused attacks, and the success rate (SR) for both processes and files across the attack vector.

As shown in Table 5, *DeepRadar's* dynamic scanner was able to protect processes and files against the majority of injection attempts (code/library). For processes, it achieved a 100% success rate against the API Mon x64, EasyHook and Binder classes of malware, followed closely by the Banking Trojan class at 99%. API Hijack represented the lowest process-based detection rate at 93%. In the case of file-based attacks, Marshal SDK injection samples were fully detected, 100%, while NtCore and Rootkit were the most challenging, with detection rates of 82% and 85% respectively.

The evaluation was also extended to include the Metasploit and Emerg Variant classes, which represent more sophisticated and diverse injection strategies. *DeepRadar* maintained high levels of resilience against these newer categories, achieving detection rates above 95% for both process- and file-level attacks. This outcome suggests that the system is capable of adapting effectively to fresh and evolving threats, underscoring its robustness for real-world deployment.

In total, *DeepRadar* early detected and neutralised 1384 of 1424 process-targeted attacks, giving a weighted average SR of 97.2%, and blocked 577 of 615 file-targeted attacks, corresponding to an SR of 93.8%. Across all attack classes combined, the system successfully defused 1961 out of 2039 attempts, representing an overall SR of 96.2%.

4.5. Comparison to related work

In this section, we provide a comprehensive comparison between the studies reviewed in Section 2 and our proposed system, *DeepRadar*. The comparison covers several key aspects, including the platform, classifier algorithm, depth of analysis, features used for behavioural modelling, datasets and sample sizes, early detection and self-defence capabilities,

Table 5
Evaluation of *DeepRadar's* early warning system in terms of success rate for defusing code/library injection attacks.

Attack vector	Attack Target (Process)			Attack Target (File)		
	# Attacks	# Defused	SR	# Attacks	# Defused	SR
API Hijack	88	82	0.932	-	-	-
API Mon x86[17]	34	32	0.941	-	-	-
API Mon x64[10]	28	28	1.000	-	-	-
Marshal SDK	-	-	-	48	48	1.000
Graphics-Hook SDK	-	-	-	78	75	0.962
NtCore[18]	-	-	-	90	74	0.822
EasyHook	85	85	1.000	67	62	0.925
Binder (Infector)	108	108	1.000	110	107	0.973
Evader	154	149	0.968	-	-	-
Spyware	320	315	0.984	-	-	-
Banking Trojan	205	204	0.995	-	-	-
Rootkit	118	112	0.949	39	33	0.846
Metasploit	124	117	0.944	85	82	0.965
Emerg var	160	152	0.950	98	96	0.980
Average SR	1424	1384	0.972	615	577	0.938

"-" indicates that a given type of attack is not the corresponding tool.
[17] http://www.rohitab.com/apimonitor.
[18] http://www.ntcore.com/files/inject2exe.htm.

and overall accuracy. The detailed comparison is presented in Table 6.

It is worth mentioning that we could not include some related studies on real-time dynamic scanning and resource consumption measurement due to limitations in accessing their source code and executables, preventing us from running them under the same environment and conditions to make a fair comparison.

As shown in Table 6, *DeepRadar* is the only system that offers early detection and self-defence capabilities. Although [36] reports higher accuracy, this is based on a small dataset and employs only traditional shallow learning models. In contrast, cutting-edge deep learning models offer both higher accuracy and greater stability. Additionally, training models on a small number of samples fails to create a scalable solution capable of detecting a wide variety of malware with differing behaviour patterns. Some related studies were limited to detecting injection attacks in specific programming languages, such as [25], or only on 32-bit operating system versions, as in [24]. *DeepRadar* overcomes these limitations by detecting injection attacks regardless of the programming language used and supporting both 32-bit and 64-bit OS versions. While [37] reports high accuracy, it lacks a standard, publicly available dataset, so we cannot make a fair comparison to its reported accuracy, as the experimental dataset is unavailable to us. Moreover, this work focuses solely on the Linux operating system, whereas our primary focus is on Windows, as the majority of malware programs are designed to target Windows. Alongside [38], we have incorporated the largest set of malware samples to train and test our detection models. Compared to [38], our system captures both IRPs and APIs as features for behavioural modelling and, by employing both static and dynamic analysis, offers a hybrid approach that balances accuracy, real-time scan speed, and resource consumption. Additionally, the lower bound of our accuracy is much higher, demonstrating the robustness of our model. The other models produce accuracy of 93% or lower, making them insufficient for comparison.

4.6. Resource efficiency and long-term durability

Our early warning mechanism operates as a real-time defensive system; therefore, resource usage is a critical measure for verifying scalability and usability. CPU utilisation and memory usage of *DeepRadar* were logged continuously over a 21-day period of real-time scanning, detecting, and confronting injection attacks, and the results are plotted in Fig. 13. In this figure, the vertical axis represents system resource usage, with CPU in blue and memory in orange, while the horizontal axis indicates the scanning time, where each unit corresponds to a 6-hour window (504 h in total). To obtain accurate measurements of resource utilisation, we monitored all threads, including child threads, associated with our early warning scanner. Over the 21-day run without any crash or malfunction, the average CPU utilisation was 20.1%, while memory consumption averaged 7.8%. The experiment was then repeated in parallel, on a separate cloned VM, for each detection model/tool in the benchmark group under identical conditions. Each detection tool was executed in CLI scanning interface mode, with GUI modules excluded, ensuring that CPU and memory consumption reflected only the scanner modules. The average results are presented in Fig. 14.

As shown in Fig. 14, our proposed scanner, *DeepRadar*, achieved the best performance among the benchmark group, ranking first in both CPU and memory efficiency. Log analysis revealed that the majority of excessive resource utilisation was attributable to scanning IPC

Table 6
A comparison of related works, eighter detecting injection attacks or malware classes capable of injection attacks, with *DeepRadar* from various perspectives.

Method	Platform	Classifier	Analysis Type - Depth	Feature(s)	Datasets	No. of samples	Early Detection	Self-defence	Accuracy
[32]	Windows	NR*	Dynamic - Hypervisor	ASM Code	Experimental	NR	×	×	NR
[33]	Linux, Windows	NR	Dynamic - Kernel	Device Driver Objects	FireEye: World's Top Malware, Spamfighter, Damballa	2K	×	×	NR
[34]	Android	Abstract Interpretation	Static - NR	Byte Codes, Data Flow	NIST Juliet Suite, OWASP, Benchmark	NR	×	×	84–90%
[35]	Windows	GRU, N-grams-DBT	Dynamic - Kernel	IRP	MalwareBenchmark, theZoo	27K	×	×	86–93%
[36]	Linux, Windows	RF, SVM, KNN, DT	Dynamic - NR	Jaccard, Entropy	Kaggle, Experimental	12K	×	×	98–99%
[37]	Linux	LR, SVM, KNN, RF, DNN	Hybrid -Kernel	Memory dumps, Application Binary Interface	Experimental	21.8K	×	×	91–98%
[38]	Window	GNN	Dynamic - Hypervisor	API	ACT-KingKong, API Call Sequences, RANSOMWARE	37K	×	×	91–98%
DeepRadar	Windows	LR, N-grams APRIORI, FFCNN	Hybrid - Kernel	API + IRP	Adminus, MaleVis, VirusSign VirusShare, Metasploit	41.3K	✓	✓	95–97%

* "NR" (Not Reported) indicates that the relevant information was not reported.

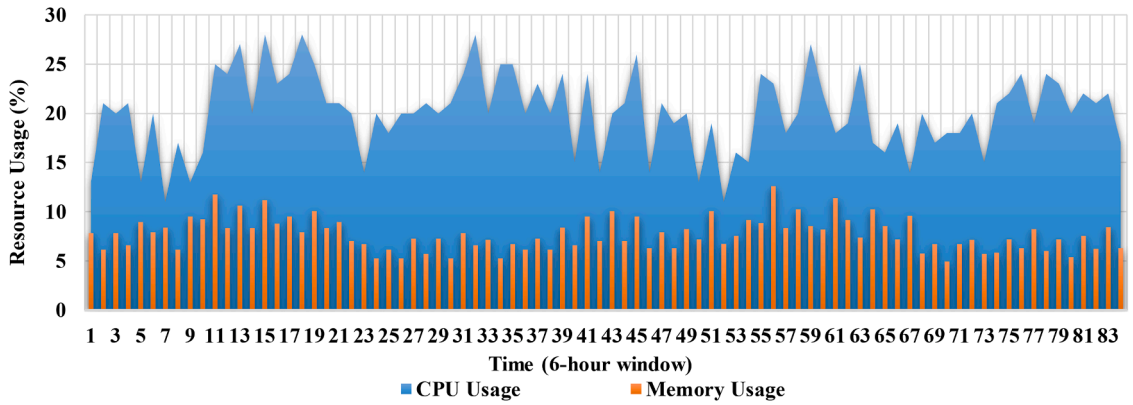


Fig. 13. System resource utilisation of the runtime scanner during a period of 21 days.

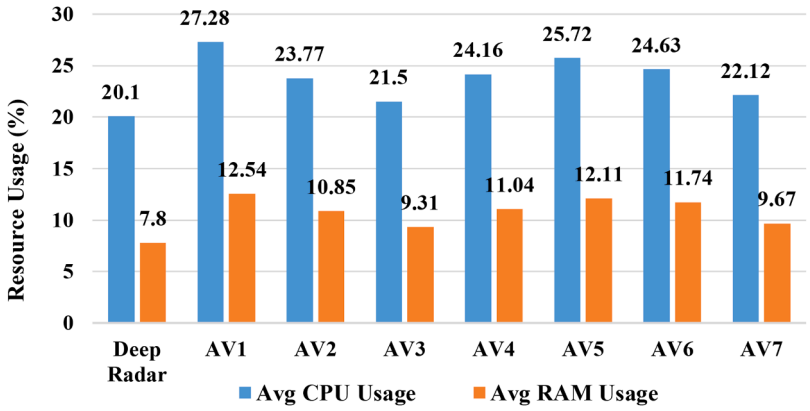


Fig. 14. Comparison of resource usage between the proposed scanner and the benchmark group (values are averaged).

communications, which is inherently time-consuming. We evaluated *DeepRadar* over an extended period of three weeks to demonstrate its reliability in performing and completing tasks during long-term missions. This capability is of paramount importance for service providers and businesses, where sustained performance and consistency are critical.

4.7. Limitations and future work

We conducted a series of experiments across various scenarios and real-world exercises to comprehensively assess *DeepRadar*, identifying its strengths and weaknesses. A broad range of metrics, including accuracy, F-score, misclassification error, ROC, AUC, and resource efficiency, were evaluated. Comparisons were made not only with previous related work but also against benchmarks set by leading AVs, with validation through Friedman and Wilcoxon tests. The results showed that *DeepRadar* outperformed both state-of-the-art studies and globally recognised AVs in detecting malware injection attacks. Furthermore, to the best of our knowledge, and based on our literature review, *DeepRadar* is the first cyber defence solution to implement the concept of early detection of malware injection attacks.

In brief, the early detection and neutralisation of 1960 out of 2038 injection attacks brings about a success rate of 96.2%, which represents a substantial achievement for a cyber-defence system addressing an often overlooked yet highly sophisticated threat. It is widely recognised within the cybersecurity community that the detection of cyberattacks can never be guaranteed at 100%. Nevertheless, our experiments demonstrated that *DeepRadar* consistently outperformed well-established antivirus solutions, none of which were able to achieve complete detection. These findings suggest that *DeepRadar* significantly narrows a critical gap in protection and offers an effective safeguard against complex code and library injection attacks.

Although *DeepRadar* has proved itself to be a well-equipped interceptor capable of dealing with a broad spectrum of malware evasion techniques, the root cause behind a small portion of successful attacks is mainly the behavioural obfuscation and detection evasion techniques that the novel malware classes are practising these days. Furthermore, the difference in detection rate between various classes of malware depends on the intrinsic nature and the technology that has been leveraged to design and develop such malware programs. We are practising to further improve the detection accuracy of *DeepRadar* and its capabilities to recognise a broader spectrum of malware evasion techniques.

It is worth noting that *DeepRadar* has been purposefully designed to detect injection attacks - i.e., malware families capable of code or library injection. While many malware families employ injection techniques, particularly heavily packed samples seeking obfuscation, privilege escalation, or detection (AV) evasion, not all malware classes do. In future work, we aim to extend *DeepRadar* to cover a broader spectrum of malware types beyond injection-capable families.

DeepRadar has primarily been developed based on the Windows operating system. In view of the growing prevalence of malware on other platforms, particularly in Linux-based could environments, Industrial IoT systems [82], and Android [26], we plan to extend future iterations of *DeepRadar* to support these operating systems, thereby broadening its applicability and strengthening its role as a cross-platform cyber-defence solution.

5. Conclusion

Injection attacks, a sophisticated technique used by contemporary malware classes, aim to obfuscate malicious activities, evade AVs, and bypass OS security by exploiting the privileges of trusted applications. Our work addressed this crucial issue by introducing *DeepRadar*, a robust multi-layer architecture capable of accurately anticipating code and library injection attacks a few cycles before occurrence and neutralising the attacks. *DeepRadar* incorporates several modules designed to monitor kernel-level APIs, call parameters, and IRPs, utilising these as behavioural features. To effectively detect and predict malware activities, we implemented logistic regression, deep neural networks enhanced with *fast Fourier convolution*, and APRIORI association rule mining for the training and validation of malware detection models, as well as for the development of an early warning and self-defence system. We evaluated the performance of *DeepRadar* against destructive malware families, including extremely obfuscated, emerging, and adversarial samples, from credible sources and benchmarked against leading antivirus tools. Our experiments demonstrated that *DeepRadar* consistently outperformed the benchmark group and prior studies, achieving higher Accuracy, F-score, AUC, and ROC results while demanding less memory and processor. The success rate in preventing code and library injection attacks and creating immunity against such attacks were also measured. Our early warning system generated alarm signals and blocked anticipated threats for 97.2% of process-level and 93.8% of file-level attacks. Statistical analysis using Friedman and Wilcoxon tests further substantiated these results. These findings underscore the significance of *DeepRadar's* capabilities, which provide a robust and scalable defence for sensitive systems and offer dynamic early-warning signals that pre-emptively counter stealthy, obfuscated, zero-day, and adversarial injection attacks.

CRediT authorship contribution statement

Danial Javaheri: Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Hassan Chizari:** Writing – review & editing, Validation, Methodology, Investigation, Formal analysis, Conceptualization. **Mahdi Fahmideh:** Writing – review & editing, Validation, Methodology, Investigation, Formal analysis, Conceptualization. **Mohammad H. Nadimi-Shahraki:** Writing – review & editing, Validation, Methodology, Investigation, Formal analysis, Conceptualization. **Junbeom Hur:** Writing – review & editing, Validation, Supervision, Resources, Investigation, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant (No.2022-0-00411, IITP-2025-RS-2021-II211810), and the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science and ICT (RS-2025-00563143) and the Ministry of Education (RS-2021-NR060143), Korea.

Appendix

Table A-1
The name and details of system functions used in methods introduced in this research [73,74].

#	Name of API	Owner Module	Function in the system
1	CloseHandle	Kernel32.dll	Eliminating an existing handle to an object
2	CreateFileA/W	Kernel32.dll	Creating or opening a file or I/O device
3	CreateProcessA/W	Kernel32.dll	Creating a new process in the system
4	CreateRemoteThread	Kernel32.dll	Creating a new thread running in the memory of another process
5	CreateRemoteThreadEx	Kernel32.dll	Creating a new thread running in the memory of another process
6	CreateSection	Kernel32.dll	Creating a new section in the PE files
7	FindFirstFileA/W	Kernel32.dll	Searching a file in a specific patch
8	FindNextFileA/W	Kernel32.dll	Continuing searching a file
9	GetThreadContext	Kernel32.dll	Retrieving the context of a running thread
10	LdrLoadDll	Ntdll.dll	Loading a library into the system memory
11	LoadLibraryA	Kernel32.dll	Loading a library into the system memory
12	OpenFile	Kernel32.dll	Opening an existing file from the hard disk
13	OpenProcess	Kernel32.dll	Creating a handle to a running process
14	OpenThread	Kernel32.dll	Creating a handle to a running thread
15	Process32First	Kernel32.dll	Getting a snapshot of a running process
16	Process32Next	Kernel32.dll	Continuing getting a snapshot of a running process
17	ReadProcessMemory	Kernel32.dll	Getting access to the allocated memory of a process
18	ResumeThread	Kernel32.dll	Recovering the execution of a paused thread
19	SetFileAttributesA/W	Kernel32.dll	Defining attributes to a file on the hard disk
20	SetProcessInformation	Kernel32.dll	Setting information for a process in the system
21	SetThreadContext	Kernel32.dll	Setting a context for a specified thread
22	SuspendThread	Kernel32.dll	Pausing the execution of a thread in the system
23	VirtualAlloc	Kernel32.dll	Performing modifications in the virtual address space of a process
24	VirtualAllocEX	Kernel32.dll	Performing modifications in the virtual address space of a process
25	WaitForMultipleObjects	Kernel32.dll	Pausing until multiple signaled obj. complete their tasks
26	WaitForSignalObject	Kernel32.dll	Pausing until a signaled object completes its task
27	Wow64GetThreadContext	Kernel32.dll	Retrieving the context of a 64-bit thread
28	Wow64SetThreadContext	Kernel32.dll	Setting a context for a thread of a 64-bit process
29	WriteFile	Kernel32.dll	Writes data to a file or I/O device
30	WriteFileEx	Kernel32.dll	Writes data to a file or I/O device
31	WriteProcessMemory	Kernel32.dll	Writing data to the memory of a process

Table A-2
The name and specifications of analysis tools used in this study.

#	Name and Version	Main Function	Access Link
1	Cuckoo Sandbox	Isolated environment for executing malware	https://github.com/cuckoosandbox/cuckoo
2	Device Tree	Dynamic analysis of kernel drivers	https://www.osronline.com/article.cfm%5Earticle=97.htm
3	DIE V3.02	Static analysis of PE file	https://horsicq.github.io/
4	H2O-3 V3.34	AI-based model training	https://github.com/h2oai/h2o-3
5	IBM SPSS Statistics V28.0	Statistical analysis of results	https://www.ibm.com/products/spss-statistics
6	IDA Pro V7.40	Binary to ASM disassembler	https://hex-rays.com/ida-pro/
7	Kali Linux V2025.2	Penetration testing and ethical hacking	https://www.kali.org/get-kali/#kali-platforms
8	Parrot Linux V6.4	Linux distro for ethical hacking	https://parrotsec.org/download/
9	Olly dbg V1.10	Debugging user-level programs	https://www.ollydbg.de/
10	PeID V0.95	Static analysis of PE file	https://github.com/wolfram77web/app-peid
11	REMnux	Malware analysis Linux distro	https://docs.remnux.org/install-distro/get-virtual-appliance
12	VirusTotal	Multi-AV scanner	https://www.virustotal.com/
13	Volatility 2.5	Memory forensics tool	https://github.com/volatilityfoundation/volatility
14	Weka V3.8.5	AI-based rule mining	https://www.cs.waikato.ac.nz/ml/weka/
15	Win dbg V10.0	Debugging kernel-level programs	https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools
16	WM-ware Workstation V14.1.1	Virtualisation tool for running VMs	https://www.vmware.com/

Data availability

Data will be made available on request.

References

[1] S.R. Davies, R. Macfarlane, W.J. Buchanan, Evaluation of live forensic techniques in ransomware attack mitigation, Forensic Sci. Int. Digit. Investig. 33 (2020) 300979, https://doi.org/10.1016/j.fsidi.2020.300979.

[2] Z. Wang, Y. Zhou, H. Liu, J. Qiu, B. Fang, Z. Tian, ThreatInsight: innovating early threat detection through threat-intelligence-driven analysis and attribution, IEEE Trans. Knowl. Data Eng. 36 (12) (2024) 9388–9402, https://doi.org/10.1109/TKDE.2024.3474792.

[3] J. Gan, C. Luo, W. Shi, Y. Liu, X. Liu, Z. Tian, An attack exploiting cyber-arm industry, IEEE Trans. Dependable Secure Comput. 22 (2) (2025) 1686–1702, https://doi.org/10.1109/TDSC.2024.3451129.

[4] L. Han, S. Liu, S. Han, W. Jia, J. Lei, Owner based malware discrimination, Future Gener. Comput. Syst. 80 (C) (2018) 496–504, https://doi.org/10.1016/j.future.2016.05.020.

[5] D. Javaheri, M. Hosseinzadeh, A. Masoud Rahmani, Detection and elimination of spyware and ransomware by intercepting kernel-level system routines, IEEE Access 6 (2018), https://doi.org/10.1109/ACCESS.2018.2884964.

[6] Sudhakar, S. Kumar, An emerging threat Fileless malware: a survey and research challenges, Cybersecurity 3 (2020) 1–12.

[7] I. Finder, E. Sheetrit, N. Nissim, Time-interval temporal patterns can beat and explain the malware, Knowl. Based Syst. 241 (2022) 108266, https://doi.org/10.1016/J.KNOSYS.2022.108266.

[8] A. Mohanta and A. Saldanha, Malware analysis and detection engineering: a comprehensive approach to detect and analyze modern malware. 2020. doi: 10.1007/978-1-4842-6193-4.

[9] G.E. Rodríguez, J.G. Torres, P. Flores, D.E. Benavides, Cross-site scripting (XSS) attacks and mitigation: a survey, Comput. Netw. 166 (2020) 106960, https://doi.org/10.1016/j.comnet.2019.106960.

[10] H. Yang, Y. Xia, H. Yang, Event-based distributed state estimation for linear systems under unknown input and false data injection attack, Signal Process. 170 (2020) 107423, https://doi.org/10.1016/j.sigpro.2019.107423.

[11] Q. Wang, W. Tai, Y. Tang, M. Ni, Review of the false data injection attack against the cyber-physical power system, IET Cyber-Phys. Syst. Theory Appl. 4 (2) (2019) 101–107, https://doi.org/10.1049/iet-cps.2018.5022.

[12] G. Liang, J. Zhao, F. Luo, S.R. Weller, Z.Y. Dong, A review of false data injection attacks against modern power systems, IEEE Trans. Smart Grid 8 (4) (2017) 1630–1638, https://doi.org/10.1109/TSG.2015.2495133.

[13] F. Martinelli, F. Mercaldo, V. Nardone, A. Santone, A.K. Sangaiah, A. Cimitile, Evaluating model checking for cyber threats code obfuscation identification, J. Parallel Distrib. Comput. 119 (2018) 203–218, https://doi.org/10.1016/j.jpdc.2018.04.008.

[14] D. Javaheri, M. Fahmideh, H. Chizari, P. Lalbakhsh, J. Hur, Cybersecurity threats in FinTech: a systematic review, Expert Syst. Appl. 241 (2024) 122697, https://doi.org/10.1016/j.eswa.2023.122697.

[15] D. Jain, S. Khemani, G. Prasad, Identification of distributed malware, in: Proceedings of the IEEE 3rd International Conference on Communication and Information Systems (ICCIS), 2018, pp. 242–246, https://doi.org/10.1109/ICOMIS.2018.8644789.

[16] D. Javaheri, P. Lalbakhsh, M. Hosseinzadeh, A novel method for detecting future generations of targeted and metamorphic malware based on genetic algorithm, IEEE Access 9 (2021), https://doi.org/10.1109/ACCESS.2021.3077295.

[17] C.M. Chen, Z.Y. Lin, Y.H. Ou, J.W. Lin, A hybrid malware analysis approach for identifying process-injection malware based on machine learning, Int. J. Secur. Netw. 19 (1) (2024) 20–30, https://doi.org/10.1504/ijsn.2024.137312.

[18] Y. Ren, Y. Xiao, Y. Zhou, Z. Zhang, Z. Tian, CSKG4APT: a cybersecurity knowledge graph for advanced persistent threat organization attribution, IEEE Trans. Knowl. Data Eng. 35 (6) (2023) 5695–5709, https://doi.org/10.1109/TKDE.2022.3175719.

[19] Y. Zhou, et al., CDTier: a Chinese dataset of threat intelligence entity relationships, IEEE Trans. Sustain. Comput. 8 (4) (2023) 627–638, https://doi.org/10.1109/TSUSC.2023.3240411.

[20] R. Wang, C. Yang, X. Deng, Y. Zhou, Y. Liu, Z. Tian, Turn the tables: proactive deception defense decision-making based on Bayesian attack graphs and Stackelberg games, Neurocomputing 638 (2025) 130139, https://doi.org/10.1016/j.neucom.2025.130139.

[21] S.J. Bu, S.B. Cho, Triplet-trained graph transformer with control flow graph for few-shot malware classification, Inf. Sci. 649 (2023) 119598, https://doi.org/10.1016/J.INS.2023.119598.

[22] Ç. Yücel, A. Koltuksuz, Imaging and evaluating the memory access for malware, Forensic Sci. Int. Digit. Investig. 32 (2020) 200903, https://doi.org/10.1016/j.fsidi.2019.200903.

[23] J. Geng, J. Wang, Z. Fang, Y. Zhou, D. Wu, W. Ge, A survey of strategy-driven evasion methods for PE malware: transformation, concealment, and attack, Comput. Secur. 137 (2024) 103595, https://doi.org/10.1016/j.cose.2023.103595.

[24] Z. Chen, M. Roussopoulos, Z. Liang, Y. Zhang, Z. Chen, A. Delis, Malware characteristics and threats on the internet ecosystem, J. Syst. Softw. 85 (7) (2012) 1650–1672, https://doi.org/10.1016/j.jss.2012.02.015.

[25] G. Peng, Y. Shao, T. Wang, X. Zhan, H. Zhang, Research on android malware detection and interception based on behavior monitoring, Wuhan Univ. J. Nat. Sci. 17 (2012), https://doi.org/10.1007/s11859-012-0864-x.

[26] Z. Wang, K. Zeng, J. Wang, D. Li, FAGnet: family-aware-based android malware analysis using graph neural network, Knowl. Based Syst. 289 (2024) 111531, https://doi.org/10.1016/J.KNOSYS.2024.111531.

[27] Y. Song, D. Zhang, J. Wang, Y. Wang, Y. Wang, P. Ding, Application of deep learning in malware detection: a review, J. Big Data 12 (1) (2025) 99, https://doi.org/10.1186/s40537-025-01157-y.

[28] Q. Sun, E. Abdukhamidov, T. Abuhmed, M. Abuhamad, Leveraging spectral representations of control flow graphs for efficient analysis of windows Malware, in: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, New York, NY, USA, Association for Computing Machinery, 2022, pp. 1240–1242, https://doi.org/10.1145/3488932.3527294. ASIA CCS '22.

[29] V. Rasikha, P. Marikkannu, An ensemble deep learning-based cyber attack detection system using optimization strategy, Knowl. Based Syst. 301 (2024) 112211, https://doi.org/10.1016/J.KNOSYS.2024.112211.

[30] Y. Chai, L. Du, J. Qiu, L. Yin, Z. Tian, Dynamic prototype network based on sample adaptation for few-shot malware detection, IEEE Trans. Knowl. Data Eng. 35 (5) (2023) 4754–4766, https://doi.org/10.1109/TKDE.2022.3142820.

[31] Y. Chai, et al., MalFSCIL: a few-shot class-incremental learning approach for malware detection, IEEE Trans. Inf. Forensics and Secur. 20 (2025) 2999–3014, https://doi.org/10.1109/TIFS.2024.3516565.

[32] D. Korczynski, H. Yin, Capturing malware propagations with code injections and code-reuse attacks, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, Association for

Computing Machinery, 2017, pp. 1691–1708, https://doi.org/10.1145/3133956.3134099. CCS '17.
[33] J. Wei, F. Zhu, Binary-centric defense of production operating systems against kernel queue injection attacks, J. Comput. Virol. Hacking Tech. 15 (2019), https://doi.org/10.1007/s11416-019-00337-8.
[34] F. Spoto, et al., Static identification of injection attacks in Java, ACM Trans. Program. Lang. Syst. 41 (3) (2019), https://doi.org/10.1145/3332371.
[35] Y. Dai, H. Li, Y. Qian, Y. Guo, R. Yang, M. Zheng, Using IRP and local alignment method to detect distributed malware, Comput. Secur. 100 (2021) 102109, https://doi.org/10.1016/j.cose.2020.102109.
[36] Y. Tyng Ling, N.F. Mohd Sani, M.T. Abdullah, N.A. Wati Abdul Hamid, Structural features with nonnegative matrix factorization for metamorphic malware detection, Comput. Secur. 104 (2021) 102216, https://doi.org/10.1016/j.cose.2021.102216.
[37] T. Panker, N. Nissim, Leveraging malicious behavior traces from volatile memory using machine learning methods for trusted unknown malware detection in Linux cloud environments, Knowl. Based Syst. 226 (2021) 107095, https://doi.org/10.1016/J.KNOSYS.2021.107095.
[38] C. Liu, B. Li, X. Liu, C. Li, J. Bao, Evolving malware detection through instant dynamic graph inverse reinforcement learning, Knowl. Based Syst. 299 (2024) 111991, https://doi.org/10.1016/J.KNOSYS.2024.111991.
[39] C. Petzold, Programming Windows, 6th ed., Microsoft Press, New York, NY, USA, 2013.
[40] Y. Shin, J. Yun, Runtime randomized relocation of crypto libraries for mitigating cache attacks, IEEE Access 9 (2021) 108851–108860, https://doi.org/10.1109/ACCESS.2021.3101638.
[41] M. Sikorski, A. Honig, Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software, 1st ed., No Starch Press, USA, 2012.
[42] Y. Li, Y.C. Chung, J. Xing, Y. Bao, G. Lin, MProbe: make the code probing meaningless, in: Proceedings of the 38th Annual Computer Security Applications Conference, in ACSAC '22, New York, NY, USA, Association for Computing Machinery, 2022, pp. 214–226, https://doi.org/10.1145/3564625.3567967.
[43] "Adminus malware dataset." 2025 [Online]. Available: https://www.adminuslabs.net. [Accessed: Sep. 2025].
[44] "VirusShare malware dataset." 2025 [Online]. Available: http://www.virusshare.com. [Accessed: Sep. 2025].
[45] "VirusSign malware dataset." 2025 [Online]. Available: http://www.virussign.com.[Accessed: Sep. 2025].
[46] "MaleVis dataset." 2019 [Online]. Available: https://web.cs.hacettepe.edu.tr/~selman/malevis. [Accessed: Sep. 2025].
[47] D. Rabadi, S.G. Teo, Advanced windows methods on malware detection and classification, in: Proceedings of the 36th Annual Computer Security Applications Conference, in ACSAC '20, New York, NY, USA, Association for Computing Machinery, 2020, pp. 54–68, https://doi.org/10.1145/3427228.3427242.
[48] A. Bensaoud, J. Kalita, CNN-LSTM and transfer learning models for malware classification based on opcodes and API calls, Knowl. Based Syst. 290 (2024) 111543, https://doi.org/10.1016/J.KNOSYS.2024.111543.
[49] D. Zhan, et al., PSP-Mal: evading malware detection via prioritized experience-based reinforcement learning with shapley prior, in: Proceedings of the ACM International Conference Proceeding Series, 2023, pp. 580–593, https://doi.org/10.1145/3627106.3627178.
[50] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, H. Yin, MAB-Malware: a reinforcement learning framework for blackbox generation of adversarial malware, in: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, New York, NY, USA, Association for Computing Machinery, 2022, pp. 990–1003, https://doi.org/10.1145/3488932.3497768. ASIA CCS '22.
[51] M.J. Zaki, W. Meira Jr, Data Mining and Analysis: Fundamental Concepts and Algorithms, Cambridge University Press, USA, 2014.
[52] S. Solanki and J. Patel, "A survey on association rule mining," 2015, pp. 212–216. doi: 10.1109/ACCT.2015.69.
[53] S. Gupta, R. Medara, and R. Singh, "Data mining in cloud computing: survey," 2020, pp. 48–56. doi: 10.1007/978-981-15-6067-5_7.
[54] P.N. Tan, M. Steinbach, V. Kumar, Association analysis: basic concepts and algorithms, Introd. Data Min. (2005) 327–414.
[55] M. Hegland, The apriori algorithm–a tutorial. Mathematics and Computation in Imaging Science and Information Processing, World Scientific Publishing, 2007, pp. 209–262, https://doi.org/10.1142/9789812709066_0006.
[56] Z. Li, F. Liu, W. Yang, S. Peng, J. Zhou, A survey of convolutional neural networks: analysis, applications, and prospects, IEEE Trans. Neural Netw. Learn. Syst. (2021) 1–21, https://doi.org/10.1109/TNNLS.2021.3084827.
[57] D. Gibert, G. Zizzo, Q. Le, Certified robustness of static deep learning-based malware detectors against patch and append attacks, in: Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security, New York, NY, USA, Association for Computing Machinery, 2023, pp. 173–184, https://doi.org/10.1145/3605764.3623914. AISec '23.
[58] S. Gayathri Devi, S. Chandila, V. Savithri, K. Saraswathi, Optimized high-dimensional memristive hopfield neural network for DoS attack detection in Mobile Adhoc Network, Knowl. Based Syst. 310 (2025) 112698, https://doi.org/10.1016/J.KNOSYS.2024.112698.
[59] D. Javaheri, S. Gorgin, J.A. Lee, M. Masdari, Fuzzy logic-based DDoS attacks and network traffic anomaly detection methods: classification, overview, and future perspectives, Inf. Sci. 626 (2023) 315–338, https://doi.org/10.1016/j.ins.2023.01.067.
[60] K. Hassini, S. Khalis, O. Habibi, M. Chemmakha, M. Lazaar, An end-to-end learning approach for enhancing intrusion detection in Industrial-Internet of Things, Knowl. Based Syst. 294 (2024) 111785, https://doi.org/10.1016/J.KNOSYS.2024.111785.
[61] L. Chi, B. Jiang, Y. Mu, Fast Fourier convolution. Advances in Neural Information Processing Systems, Curran Associates, Inc., 2020, pp. 4479–4488.
[62] V. Nair, M. Chatterjee, N. Tavakoli, A.S. Namin, C. Snoeyink, Optimizing CNN using Fast Fourier Transformation for object recognition, in: Proceedings of the19th IEEE International Conference on Machine Learning and Applications, ICMLA 2020, 2020, pp. 234–239, https://doi.org/10.1109/ICMLA51294.2020.00046.
[63] L. Liu, B. Wang, B. Yu, Q. Zhong, Automatic malware classification and new malware detection using machine learning, Front. Inf. Technol. Electron. Eng. 18 (2017) 1336–1347, https://doi.org/10.1631/FITEE.1601325.
[64] F. Rustam, I. Ashraf, A.D. Jurcut, A.K. Bashir, Y. Bin Zikria, Malware detection using image representation of malware data and transfer learning, J. Parallel Distrib. Comput. 172 (C) (2023) 32–50, https://doi.org/10.1016/j.jpdc.2022.10.001.
[65] H. Deng, C. Guo, G. Shen, Y. Cui, Y. Ping, MCTVD: a malware classification method based on three-channel visualization and deep learning, Comput. Secur. 126 (2023) 103084, https://doi.org/10.1016/j.cose.2022.103084.
[66] E. Eroğlu Demirkan, M. Aydos, Enhancing malware detection via RGB assembly visualization and hybrid deep learning models, Appl. Sci. 15 (13) (2025), https://doi.org/10.3390/app15137163.
[67] O. Sharma, A. Sharma, A. Kalia, MIGAN: GAN for facilitating malware image synthesis with improved malware classification on novel dataset, Expert Syst. Appl. 241 (2024) 122678, https://doi.org/10.1016/j.eswa.2023.122678.
[68] D. Zhu, S. Lu, M. Wang, J. Lin, Z. Wang, Efficient precision-adjustable architecture for softmax function in deep learning, IEEE Trans. Circuits Syst. II Express Briefs 67 (12) (2020) 3382–3386, https://doi.org/10.1109/TCSII.2020.3002564.
[69] D. Javaheri, M. Hosseinzadeh, A framework for recognition and confronting of obfuscated malwares based on memory dumping and filter drivers, Wirel. Pers. Commun. 98 (1) (2018), https://doi.org/10.1007/s11277-017-4859-y.
[70] T. Ball, et al., Thorough static analysis of device drivers, ACM SIGOPS Oper. Syst. Rev. 40 (4) (2006) 73–85, https://doi.org/10.1145/1218063.1217943.
[71] I. You, K. Yim, Malware obfuscation techniques: a brief survey, in: Proceedings of the 2010 International Conference on Broadband, Wireless Computing Communication and Applications, BWCCA 2010, 2010, pp. 297–300, https://doi.org/10.1109/BWCCA.2010.85.
[72] X. Xie, B. Lu, D. Gong, X. Luo, F. Liu, Random table and hash coding-based binary code obfuscation against stack trace analysis, IET Inf. Secur. 10 (1) (2016) 18–27, https://doi.org/10.1049/IET-IFS.2013.0137.
[73] S.B. Schreiber, Undocumented Windows 2000 Secrets: A Programmer's Cookbook, Addison-Wesley Longman Publishing Co., Inc., USA, 2001.
[74] "Microsoft documentation.", 2025 [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api. [Accessed: Feb. 2025].
[75] M.Z. Shafiq, S.M. Tabish, F. Mirza, and M. Farooq, "PE-Miner: mining structural information to detect malicious executables in realtime BT–recent advances in intrusion detection," E. Kirda, S. Jha, and D. Balzarotti, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 121–141.
[76] M. Narouei, M. Ahmadi, G. Giacinto, D. Takabi, A. Sami, DLLMiner: structural mining for malware detection, Secur. Commun. Netw. 8 (2015), https://doi.org/10.1002/sec.1255 n/a-n/a.
[77] J.M. White, D. Conway, Machine Learning for Hackers, O`Reilly, CA, USA, 2012.
[78] P. Lalbakhsh, Y.P.P. Chen, TACD: a transportable ant colony discrimination model for corporate bankruptcy prediction, Enterp. Inf. Syst. 11 (2015) 1–28, https://doi.org/10.1080/17517575.2015.1090630.
[79] T. Fawcett, An introduction to ROC analysis, Pattern Recognit. Lett. 27 (8) (2006) 861–874, https://doi.org/10.1016/j.patrec.2005.10.010.
[80] I.S. MacKenzie, Chapter 6–hypothesis testing, in: I.S. MacKenzie (Ed.), Human-Computer Interaction, Morgan Kaufmann, Boston, 2013, pp. 191–232, https://doi.org/10.1016/B978-0-12-405865-1.00006-6.
[81] C. Davis, SPSS for Applied Sciences: Basic Statistical Testing, CSIRO Publishing, 2013. CSIRO PublishingAccessed: Apr. 30, 2024. [Online]. Available: https://www.publish.csiro.au/book/6900/.
[82] H. Liu, Y. Zhou, B. Fang, Y. Sun, N. Hu, Z. Tian, PHCG: PLC honeypoint communication generator for industrial IoT, IEEE Trans. Mob. Comput. 24 (1) (2025) 198–209, https://doi.org/10.1109/TMC.2024.3455564.