International Research Journal of Multidisciplinary Scope (IRJMS), 2025; 6(2): 896-918 IRJMS

Original Article | ISSN (O): 2582-631X DOI: 10.47857/irjms.2025.v06i02.03576

Utilizing Virtual Machine Introspection and Memory Forensics to Identify Different Forms of Process Injection in a Virtualized Environment

Darshan Tank[1]*, Miral Patel J[2], Hasmukh Koringa P[2], Divyesh Keraliya[2], Jaydeep Tadhani R[1], Sunil Soni J[1]

[1]Department of Information Technology, Government Polytechnic, Rajkot, India, [2]Department of Electronics and Communication, Government Engineering College, Rajkot, India. *Corresponding Author's Email: dmtank@gmail.com

Abstract

Sophisticated malware frequently employs advanced evasion techniques to remain undetected by traditional security mechanisms. One of the most commonly used tactics is process injection, where malicious code is covertly inserted into the address space of legitimate processes. This allows the malware to operate under the guise of trusted applications, making detection significantly more challenging. In response to this issue, the present study introduces a novel detection methodology that functions entirely outside the virtual machine (out-of-VM). This technique leverages advanced memory introspection to identify and analyze different forms of process injection within virtualized environments. Notably, the approach is agentless, meaning it does not require any software to be installed within the guest VM, thereby eliminating the risk of the detection system itself being compromised or bypassed by the malware. Instead, it analyzes memory from the hypervisor level, providing a more secure and isolated vantage point. Experimental evaluations validate the effectiveness of the proposed method, demonstrating superior performance when compared to existing detection frameworks. Specifically, the method achieves higher detection accuracy, with more true positives and fewer false positives. It is capable of precisely identifying injected memory regions and detecting a broader spectrum of malware types, thereby outperforming current state-of-the-art solutions across all major evaluation metrics.

Keywords: Malware Detection, Memory Analysis, Process Injection, Security, Virtual Machine Introspection, Volatility, Windows.

Introduction

Distributed computing has become a dominant paradigm in recent years, with virtualization serving as a critical foundation for cloud computing. Virtual Machine Monitors (VMMs) allow multiple virtual machines (VMs) to operate on a single physical host, but this flexibility introduces significant security risks. Virtual machine security remains one of the primary challenges in cloud infrastructure, as adversaries often exploit VMs to gain unauthorized access to virtualized environments. Traditional security measures are insufficient against modern malware, which has evolved to be more persistent and adaptive. Among the tactics employed by malware, process injection is a powerful method for evading detection by concealing malicious code within legitimate processes. Process injection allows attackers to access system resources, memory, and network assets of the target process while gaining elevated privileges (1). Numerous process injection techniques exist, including Remote DLL Injection, Remote Thread Injection, Hollow Process Injection, Reflective DLL Injection, and others. Detecting process injection within virtualized environments is particularly challenging due to the lack of direct access to the VMs' physical memory. This study addresses this gap by proposing an automated approach to detect various process injection techniques in virtualized systems. A tool named Hashtest has been described in the GitHub repository (2), which is designed to validate the integrity of in-memory code through the use of hashes. A dynamic malware analysis framework, VEDefender, was introduced to detect dormant, suspicious, or concealed processes in a monitored virtual machine without modifying the guest OS kernel on the host (3). A number of techniques

(Received 20th December 2024; Accepted 23rd April 2025; Published 30th April 2025)

exist for identifying process injection, including as process monitoring, system calls, named pipes, Windows API calls, events from DLL/PE files, and more. To identify typical injection strategies, one may examine API call grouping patterns as OpenProcess → VirtualAllocEx → WriteProcessMemory → CreateRemoteThread (4). The method of malware analysis was proposed, involving the monitoring of processes running in a virtual system. In this approach, the software within the virtual machine is monitored using a virtual machine introspection method (5). In addition, it was demonstrated that Virtual Machine Introspection could identify malicious processes in virtual machines by collecting system call data from memory pages examined during VM execution (6). Finding areas of memory in a process' virtual address space that might be injected is possible using a number of methods. One such approach is Malfind, a popular component of the Volatility memory analysis system that can identify regions of memory that have been artificially enhanced (7). Nevertheless, the CreateRemoteThread → LoadLibrary function cannot be used by Malfind to identify DLLs that are injected into a process. Several methods of process hollowing may be found with the help of the Hollowfind plugin for Volatility (8). These methods have several drawbacks, yet they are nonetheless helpful. While Malfind has a high probability of false positives, Hollowfind only finds some of the potential host-based code injection threats. Also, instead of finding malicious memory addresses inside a process, Membrane only displays the processes that are impacted (9). Jared Atkinson's Get-InjectedThread.ps1 PowerShell script is another method; it checks all running threads for evidence of memory injection and terminates them if they do (10). In some contexts, it could be hard to tell the difference between malicious and authorized uses of Windows API calls. As an example, most typical applications do not need the use of CreateRemoteThread, which is why many security scanners detect it and may possibly discover the suspicious DLL on disk (11). Analyses that look for CreateRemoteThread calls from any process often provide false positives. Code injection may also be indicated by certain Windows API calls, such as VirtualAllocEx and WriteProcessMemory, which are used to manipulate the memory of another process (11). A method for identifying host-based code injection vulnerabilities in memory dumps was developed and named Quincy (12). The supervised machine learning-based Quincy was made available as a Volatility plugin and was made compatible with three versions of Windows (12). To locate all executable pages relevant to an investigation, a technique for individual user-space processes was suggested (13). Malware detection using API calls, recurrent neural networks, and Long Short Term Memory (LSTM) was proposed (14). Detection systems for process injection can generate a large volume of data, which may not be immediately useful for defense unless collected under specific conditions (15). Malware classification systems require a substantial number of samples to function effectively (16).

According to the AV-TEST Security Report 2016/17, the cybersecurity landscape witnessed a significant escalation in the number of malware samples, with over 640 million malicious programs identified by the end of 2016, underscoring the persistent and growing threat to users worldwide (17). Kaspersky Lab's annual security bulletin also echoed these concerns, reporting a substantial rise in cyberattacks, including over 758 million malicious attacks from online resources located in 203 countries and territories throughout the year (18). Symantec's Internet Security Threat Report further highlighted the increasing sophistication of threats, noting the rise of zero-day vulnerabilities and advanced persistent threats (APTs), along with a marked surge in ransom ware attacks targeting both individuals and organizations (19). Among the advanced techniques employed by threat actors, process injection remains a prevalent method for evading detection and maintaining persistence. Tools and techniques such as those detailed in the InjectProc repository illustrate the variety of process injection strategies leveraged by attackers, including remote thread injection, process hollowing, and DLL injection, emphasizing the need for enhanced defensive mechanisms against such low-level exploits (20).

Finally, the aforementioned difficulties severely restrict the effectiveness of the various approaches and tools now available for identifying process injection attacks in memory

dumps.mAs with any detection system, adversaries may attempt to understand and bypass the detection heuristics. To develop an effective detection system, additional context is still necessary. In this study, we present the VMI-based Process Injection Detection (*VMIPID*) approach to address the limitations of the previously mentioned solutions. Process injection is a long-standing tool in the arsenal of attackers, enabling the manipulation of legitimate processes or hiding malware's presence. Current detection frameworks are often limited in scope, unable to adapt to modern injection techniques or reliably identify malicious memory regions. To address these limitations, this study presents an advanced memory introspection technique that leverages Virtual Machine Introspection (VMI) to analyze live memory data in virtualized environments dynamically. Here, we zero in on eight distinct process injection implementations: Atom Bombing, Thread Execution Hijacking, Reflective DLL Injection, Portable Executable Injection, Remote Thread Injection, and Asynchronous Procedure Call (APC) Injection. A Volatility plugin named *ProcInjectionsFind* was developed to detect injected memory regions, with the code made available in a public repository to facilitate reproducibility (21).

Threat Model and Assumptions

Threat Model

Process injection vulnerabilities in virtualized settings are the major emphasis of this study. Malware continues to target Windows-based systems in particular. Attacks on Infrastructure as a Service (IaaS) cloud architecture, in which the host operating system has little control over guest systems, are assumed to occur within this scope.

Assumptions

Inter-VM attacks are not considered.

The hypervisor, cloud provider, and underlying infrastructure are assumed to be secure.

Zero-filled or empty VAD regions are treated as safe to reduce false positives.

Methodology

In order to examine virtual machine memory and locate injected areas, the detection framework incorporates the Volatility framework (22), the KVM hypervisor (23), and the LibVMI library (24). Figure 1 shows the overall layout of our suggested detection architecture. Once installed and set up on the host system, KVM acts as a hypervisor or Virtual Machine Monitor (VMM), supervising virtual machines that run guest operating systems like Windows 7, Windows 8.1, or Windows 10. For low-level insights, the LibVMI package allows access to the memory of a running virtual machine. We install the Volatility framework (version 2.6.1) and the LibVMI Python bindings (version 3.4) on the host operating system. When used in tandem, these instruments dissect dynamic malware. For the purpose of conducting memory forensic analyses in real-time, our suggested solution makes use of Virtual Machine Introspection (VMI).

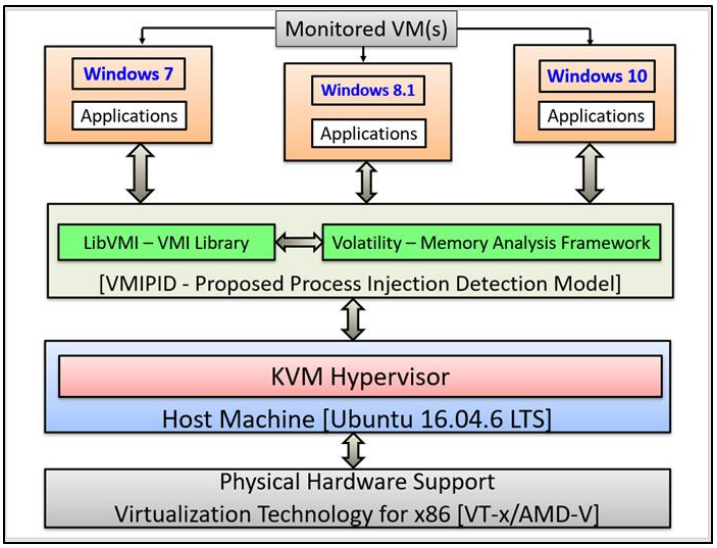

Figure 1: The Architecture of Proposed Framework

Use of this virus allows attackers to compromise Windows virtual computers using process injection vulnerabilities. Shortly after the injection, a dump file is generated to capture the exact state of the virtual machine's core memory for further examination. On the other hand, you may use the LibVMI Python bindings to directly access the memory of a running VM and examine it for signs of process injection immediately. In order to locate memory areas that have been injected by process injection, the VMI-based Process Injection Detection (*VMIPID*) model was developed. By employing Volatility, a free and open-source memory forensics tool, we were able to test the model with both real-life virtual computers and memory snapshots that included malware.

Research Approach

Numerous techniques exist for inserting and executing malicious code into running processes. The allocation of memory inside the victim process's address space is a common characteristic across process injection techniques, which use various execution styles and effect the victim process's memory-resident data structures and API calls differently. Malware often inserts itself into a process's virtual address space by creating a new memory area, also known as a Virtual Address Descriptor (VAD). This research suggests a new way to find possible injected memory in a victim process's virtual address space. For the purpose of identifying code injection, the proposed detection approach thoroughly checks all currently operating processes' memory regions. An add-on for Volatility called *ProcInjectionsFind* uses specially designed algorithms to identify different forms of process injection in virtualized settings. Following are the steps to identify Suspicious Processes.

Table 1: Identifying Suspicious Processes

Input: VM's Primary Memory or A Memory Image that Has Been Compromised with Malware **Output: A List of Suspicious Processes with the Process-Thread Id**
1: Examine all running processes 2: List process' handles in each running process 3: Refine process' handles of type 'THREAD' 4: Examine a thread that isn't being handled or produced by the process it's running in (*) 5: Update the suspicious process list with the thread's handle PID and TID (*) Step 4 exempted the thread' handles of the following • Handles produced by csrss.exe • Handles produced by its parent process • In addition to its own operation and the processes that were started before it, **csrss.exe** is involved in the creation of every process and thread (25). • A parent process may legitimately create a handle of type **THREAD** in its child process

Table 2: Lists the Proposed Techniques for Detection

Algorithm 1: "Remote DLL Injection Via Createremotethread And Load library" Detection
Input: VM's primary memory OR a memory image that has been compromised with malware Output: Indicate the injected process ID, process name, full DLL name, and associated VAD information 1: With the use of the procedures in Table 1, identify suspicious processes 2: Make the following checks for each thread listed in the suspicious processes list 3: Link the thread to the relevant VAD Verify the file's mapping on the disc Thread is mapped to kernel32.dll 4: Look for the LoadLibrary (or LoadLibraryEx) API method during thread execution. 5: Connect DLLs to the thread (which is in charge of injecting the malicious DLL) by tying the load time of the DLL to the thread's creation time using a predetermined time period, and add it to the list of suspect DLLs 6: Verify if the injected DLL has a corresponding entry in the process' IAT, i.e., No entry for the injected DLL exists in the process' IAT 7: Mark the DLL and the associated memory area as suspect 8: Dump the complete VAD associated with a suspicious memory area

9: Verify injection by comparing the dumped VAD with VirusTotal score

Algorithm 2: "Thread execution hijacking" detection

Input: VM's primary memory OR a memory image that has been compromised with malware.

Output: Show different characteristics for each injected memory region.

1: Check all running processes.

2: List every thread in each running process.

3: Discovered thread id in the list of suspicious processes.

4: Check if a thread is suspended, i.e., 'Waiting' is the thread's State and 'Suspended' is the Wait Reason.

5: Run the following memory region (VAD) check by traversing the process' VADs.

- Any VAD region which marked as private carries the VadS tag and executes permission.

6: Mark the corresponding area of memory as suspicious.

7: Dump the complete VAD associated with a suspicious memory area.

8: Verify injection by comparing the dumped VAD with VirusTotal score.

Algorithm 3: To recognize the following injection type

a. Remote thread injection using b. PE injection CreateRemoteThread

c. Reflective DLL injection d. Hollow process injection

e. APC injection f. Atom Bombing

Input: VM's primary memory OR a memory image that has been compromised with malware.

Output: Show different characteristics for each injected memory region.

1: Check all running processes.

2: List every thread in each running process.

3: Get the Win32StartAddress attribute's entry point for the thread

4: At the thread's entry point, implement the subsequent injection filters

- Any process thread that did not have a file object was mapped to a VAD
- The memory is committed and any thread in the process is mapped to a VAD with a file object, but the kind of file object is not an IMAGE FILE
- Any thread in the process that is mapped to a VAD that has an executable file object that is distinct from the image file for the loaded process
- Any thread in the loaded process that is mapped to a VAD that contains an identical exe file object, but a thread is suspended, i.e., 'Waiting' is the thread's State and 'Suspended' is the Wait Reason

5: Analyse VADs for processes

6: Implement the subsequent injection filters to the VAD area

- Any VAD area that represents a memory-mapped file (type _MMVAD (Vad) or _MMVAD_LONG (VadL)), but the fields VadImageMap and Image are not set in the Vad Type and Control Flag fields, respectively
- Any VAD region having the characteristics: VadS tag, execute permission, private, committed, memory-resident, and VadNone type

7: Dump the complete VAD associated with a suspicious memory area

8: Verify injection by comparing the dumped VAD with VirusTotal score

The proposed approach determines whether a running process is the result of process injection by analyzing its threads and memory segments. Table 2 outlines the proposed techniques for detecting process injection. The detection techniques outlined in Tables 1 and 2 have been integrated into a single Volatility plugin/module named *ProcInjectionsFind*, which can be executed from the Volatility command line. This module conducts multiple tests to identify malicious or injected memory regions and provides detailed information about each identified memory region that aligns with the rules defined by the proposed methods. The *ProcInjectionsFind* plugin can analyze either a Windows memory image or the memory of a live virtual machine to identify signs of process injection. It examines the threads and memory regions of each process to detect anomalies. The described methods have been successfully applied to both memory snapshots and live virtual machines infected with malware, and the results have been verified.

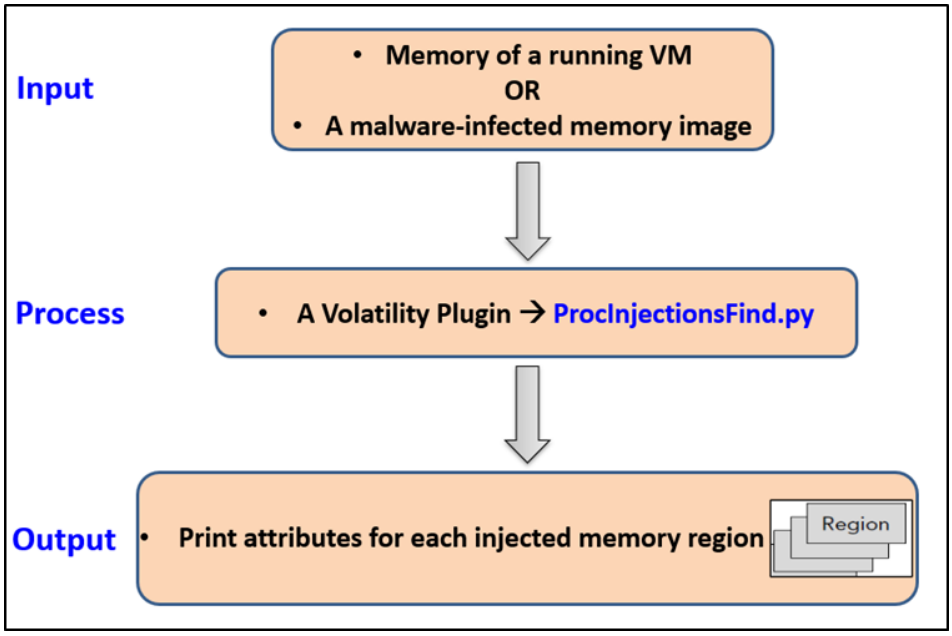

Figure 2: Workflow for the Proposed System

Our proposed framework, illustrated in Figure 2, operates in three stages: input, process, and output. In the input stage, the framework accepts either an infected memory image or the live memory of an active virtual machine. This input is then analyzed in the process stage by the *ProcInjectionsFind* module, which examines memory regions for signs of injection. In the output stage, the module displays various properties of each identified injected memory location, providing detailed insights into potential process injection activities.

Experimental Setup

An outline of our experimental apparatus is given in this section. All of the research takes place on a host machine using the settings listed in Table 3. Ubuntu 16.04.6 LTS is installed and configured using KVM to set up the virtualization environment.

Table 3: Setups for Test Platforms

Host OS	**Ubuntu 16.04.6 LTS**
Host OS Type	64-bit
Linux Kernel	Linux 4.15.0-74-generic
Architecture	X86_64
Processor	Intel(R) Core™ i5-8265U CPU @ 1.60GHz x 8
Disk	1 TB
Number of cores and threads	4 and 8
Physical memory (RAM)	8 GB
Hypervisor (VMM)	KVM
Virtual Machine – 1	OS – Windows 7, vCPU - 1 Memory – 2 GB, Storage – 40 GB
Virtual Machine – 2	OS – Windows 8.1, vCPU - 1 Memory – 2 GB, Storage – 40 GB
Virtual Machine – 3	OS – Windows 10, vCPU - 1 Memory – 2 GB, Storage – 40 GB
Tools / Framework used	LibVMI python bindings (version-3.4) and Volatility framework (version-2.6.1) (Both are open-source tools)

The IaaS cloud model is used to simulate a possible assault setting. Running Windows 7, Windows 8.1, and Windows 10 in guest mode resulted in the creation of three separate virtual machines. The Volatility framework and the LibVMI Python bindings are two examples of the open-source technologies that we use in our studies. Dynamic malware analysis and the extraction of higher-level semantic information from live memory data inside the virtual machines were accomplished using the Volatility framework and the Virtual Machine Introspection (VMI) application LibVMI.

Malware Hiding Technique Covered in this Work

The Virtual Address Descriptor's (VAD) protection field just displays the initial protection that was specified when memory was allocated. An adversary might take advantage of this by first creating memory without the WRITE or EXECUTE privileges, and then changing the protection to permit these rights for the region of memory that contains malicious code. It is common for malicious executables to deliberately modify the memory section's security from READONLY to EXECUTE_READWRITE. The VirtualProtectEx API method, which lets you change the protection of a memory area in a process's virtual address space, is used to accomplish this adjustment (26).

Results and Discussion

ProcInjectionsFind is a standalone Volatility plugin/module that integrates the detection techniques outlined in Tables 1 and 2. It can be executed directly from the Volatility command prompt. This module performs a series of tests to detect malicious or injected memory regions and provides detailed information about each region that meets the criteria defined by the proposed detection techniques. The proposed *VMIPID* model analyzes the memory of a virtual machine in real-time, inspecting the memory of each active process for signs of injected code. The model classifies each memory region as either benign or malicious based on its findings. To detect malicious or injected memory regions, the model performs a series of rigorous tests.

Table 4: Metrics for Evaluation Definition

Measures	Definition
True Positive (TP)	The number of correctly identified injected memory regions.
False Positive (FP)	The number of incorrectly identified injected memory regions.
True Negative (TN)	The number of correctly identified benign memory regions.
False Negative (FN)	The number of incorrectly identified benign memory regions.
Metric	Formula
Accuracy	(TP + TN) / (TP + FP + TN + FN)
Detection Rate	TP / (TP + TN + FP + FN)
F1-Score	2 * (P * R) / (P + R)
False Positive Rate (FPR)	FP / (FP + TN)
Precision (P)	TP / (TP + FP)
Recall (R)	TP / (TP + FN)

The effectiveness of the proposed *VMIPID* model was evaluated using multiple assessment metrics, including Accuracy, Detection Rate, F1-Score, False Positive Rate (FPR), Precision (P), and Recall (R). The definitions of these metrics are provided in Table 4.

Evaluation Using Process Injection PoCs

Process injection techniques were implemented using the Proofs of Concept (PoCs) outlined in Table 5. Where necessary, minor modifications were made to the original authors' code to ensure it was build- and run-ready. The source code was compiled using Microsoft Visual Studio Community 2017, Version 15.9.37. Additionally, Table 5 includes the PoCs for the malware concealment method described in Section 5.2.

Table 5: Evaluation Using Process Injection PoCs

Sr No	Process Injection Techniques	PoCs Used
1	Remote DLL injection	● Methods for injecting. The Evil Bit's Injection on GitHub (27). ● Methods for Injecting Processes. secrary/InjectProc on GitHub (20). ● DLL injection methods number seven. fdiskyou/injectAllTheThings on GitHub (28). ● Windows Injection for Processes. Some basic process injection methods for the Windows platform may be found in the following GitHub repository: CptGibbon/Windows-Process-Injection (29).
2	Remote thread injection	● Methods for injecting. The Evil Bit's Injection on GitHub (27).
3	PE injection	● Methods for injecting. The Evil Bit's Injection on GitHub (27). ● Windows Injection for Processes. Some basic process injection methods for the Windows platform may be found in the following GitHub repository: CptGibbon/Windows-Process-Injection (29). ● Tools for Submitting Code. at the main branch of DFRWS-USA-2019 on GitHub f-block/DFRWS-USA-2019 (30).
4	Reflective DLL injection	● DLL Injection via Reflection. Stephen Lester's Reflective DLL Injection on GitHub (31). ● DLL injection methods number seven. fdiskyou/injectAllTheThings on GitHub (28). ● Tools for Submitting Code. master/f-block/DFRWS-USA-2019 tools in DFRWS-USA-2019 on GitHub (30).
5	Hollow process injection	● Hollowing out the process. GitHub repository: m0n0ph1/Process-Hollowing (32) ● Methods for injecting. The Evil Bit's Injection on GitHub (27). ● Windows Injection for Processes. Some basic process injection methods for the Windows platform may be found in the following GitHub repository: CptGibbon/Windows-Process-Injection (29). ● Tools for Submitting Code. at the main branch of DFRWS-USA-2019 on GitHub f-block/DFRWS-USA-2019 (30).
6	Thread execution hijacking	● Methods for injecting. The Evil Bit's Injection on GitHub (27). ● Windows Injection for Processes. Some basic process injection methods for the Windows platform may be found in the following GitHub repository: CptGibbon/Windows-Process-Injection (29).
7	APC injection	● Methods for injecting. The Evil Bit's Injection on GitHub (27). ● Methods for Injecting Processes. secrary/InjectProc on GitHub (20).
8	AtomBombing	● Bombing using atomic weapons. New Windows Code Injection Tool Available at BreakingMalwareResearch/atom-bombing on GitHub (33). ● Tools for Submitting Code. at the main branch of DFRWS-USA-2019 on GitHub f-block/DFRWS-USA-2019 (30).
	Malware Hiding Technique	
1	PE injection	● Tools for Submitting Code. at the main branch of DFRWS-USA-2019 on GitHub f-block/DFRWS-USA-2019 (30)

2	Reflective DLL injection	● Tools for Submitting Code. at the main branch of DFRWS-USA-2019 on GitHub f-block/DFRWS-USA-2019 (30)
3	Hollow process injection	● Tools for Submitting Code. at the main branch of DFRWS-USA-2019 on GitHub f-block/DFRWS-USA-2019 (30) ● Process hollowing using several methods using KSLSample.vmem (34)

Table 6: Process Injection Detection Methods are compared to the Current Methods

Sr	Process Injection Techniques	Compared With
1	Remote DLL injection	FindDLLInj (35)
2	Hollow process injection	Malfind (36), Hollowfind (37), Threadmap (38), Malfofind (39)
3	Thread execution hijacking, remote thread injection, malicious code injection, atomic bombing, and reflective DLL injection	Malfind (36)

Table 6 presents a comparison of various process injection detection methodologies with the proposed approach. This study utilized several Volatility commands to detect malware in Windows memory images, including Malfind (36), Hollowfind (37), Threadmap (38), Malfofind (39), Vadinfo (40), Impscan (41), and Volshell (42).

Experimental Findings

This section presents the experimental findings of the proposed *VMIPID* model. The performance of the framework was evaluated using multiple assessment metrics, including Accuracy, Detection Rate, F1-Score, False Positive Rate (FPR), Precision, and Recall. A series of experiments were conducted to assess the model's effectiveness, followed by a comparison with existing methodologies from the literature. The results demonstrate the model's ability to detect process injection techniques effectively, highlighting its advantages over traditional approaches. Detailed results and analysis are provided in the following subsections.

Figure 3: A List of the Host's Active VMs

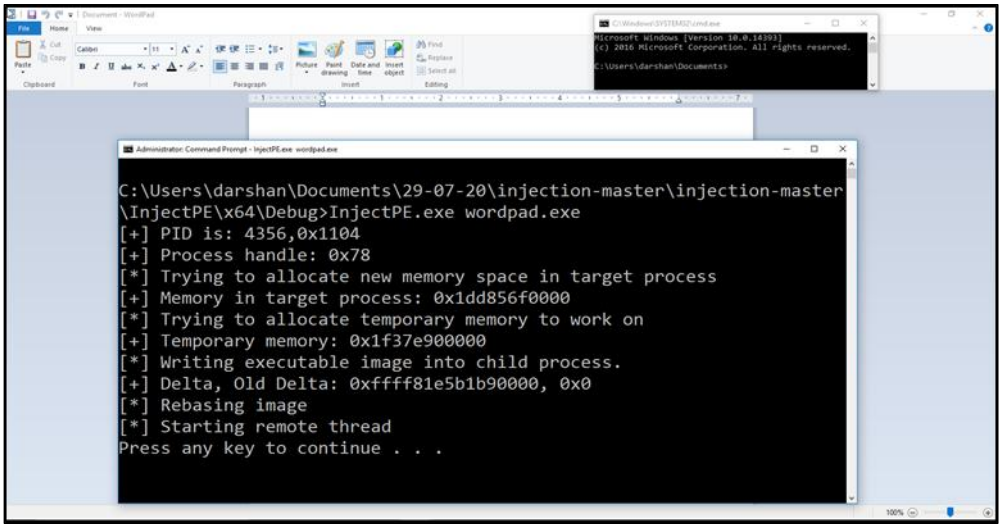

Figure 4: Injecting PE into the Win10_VM

One can see all of the host's currently running virtual machines in Figure 3. Figure 4 shows how PE injection is done on the Windows 10 virtual system using the Proofs of Concept (PoCs) from Table 5. The injection instance is initialized by launching the target process, wordpad.exe. Figure 4 shows the injection command.

Figure 5 displays the results of taking a memory snapshot of win10_VM using the 'virsh dump' command after PE injection. This document explains how to use the *ProcInjectionsFind* Volatility module to automatically identify the different process injection mechanisms. Both the memory of a virtual machine and a memory image infected with malware (PE injection) are subjected to the *ProcInjectionsFind* plugin (Figure 6 and Figure 7).

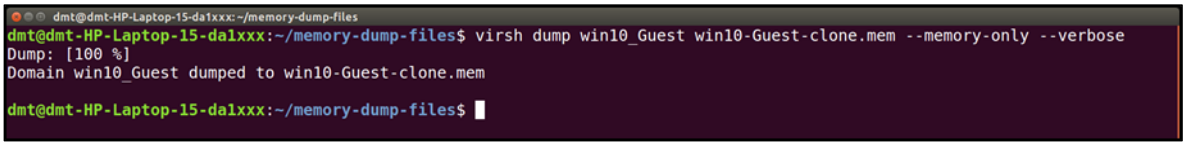

Figure 5: Acquiring the Live Win10_VM Memory Image

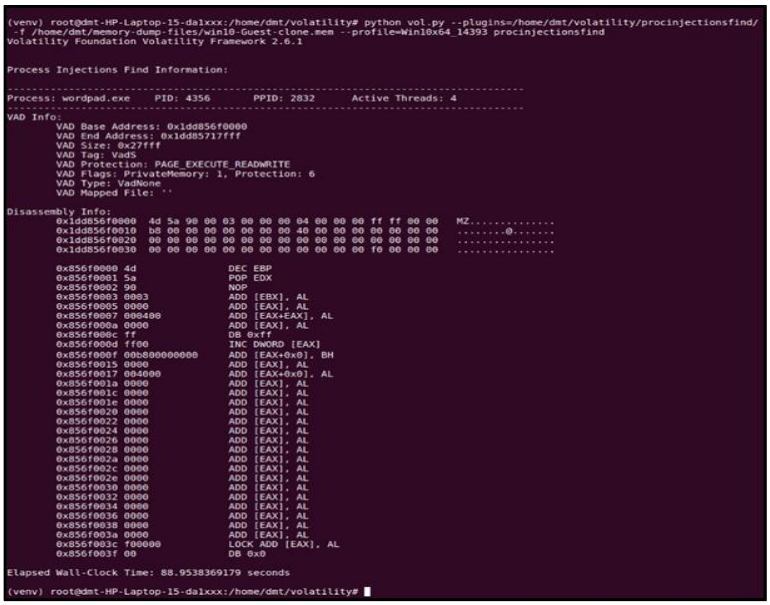

Figure 6: *ProcinjectionsFind* Plugin Execution on a Memory Image (Win10_VM) With Malware (PE Injection)

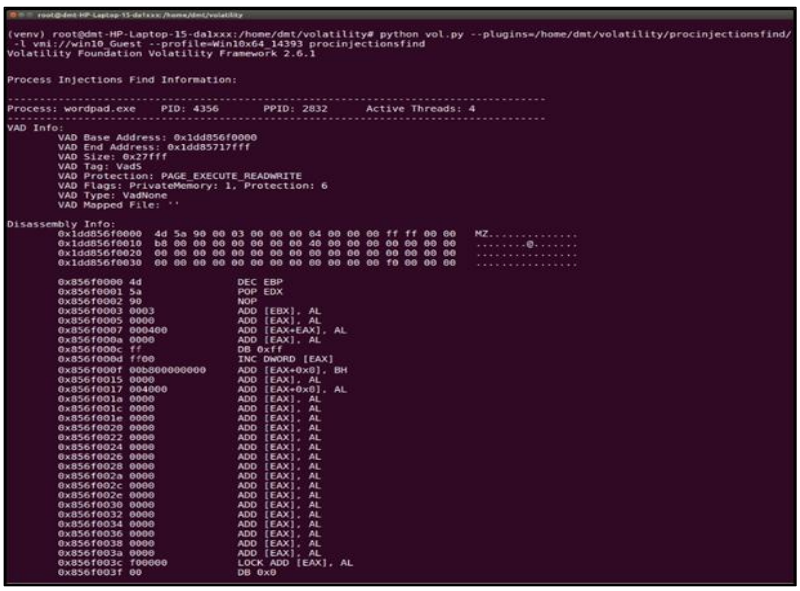

Figure 7: *ProcinjectionsFind* Plugin Execution on the Memory of a Running Win10_VM

Figure 6 illustrates the execution of the *ProcinjectionsFind* plugin on a memory image (win10_VM) containing malware through PE injection. Figure 7 demonstrates the execution of the *ProcinjectionsFind* plugin on the live memory of a running win10_VM. Details on the injected/victim process's VADs, disassembly, and hex-dump are published at the base address of the VAD by the *ProcInjectionsFind* plugin. We may further check the findings or do additional research by dumping an injected memory area to disk, which is made possible by the plugin.

The *ProcInjectionsFind* Volatility plugin was tested on 75 different malware-infected memory images (25 images obtained from each of the three virtual machines), as shown in Figure 8. The results confirm that the plugin operates as expected.

Virtual Machine	Average Performance												Average Sample Size
	Precision (P)	Recall (R)	False Positive Rate (%)	Detection Rate (%)	F1-Score (%)	Accuracy (%)	Precision (P)	Recall (R)	False Positive Rate (%)	Detection Rate (%)	F1-Score (%)	Accuracy (%)	
	Malfind						ProcInjectionsFind						
win7_Guest	0.08	0.91	0.29%	0.01%	12.88%	99.70%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	8971
win8.1_Guest	0.06	0.83	0.24%	0.01%	12.80%	99.76%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6474
win10_Guest	0.10	0.75	0.12%	0.01%	22.61%	99.87%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	11439
	Hollowfind						ProcInjectionsFind						
win7_Guest	0.19	1.00	0.09%	0.02%	30.26%	99.91%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	8119
win8.1_Guest	0.08	1.00	0.19%	0.02%	14.29%	99.81%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6172
win10_Guest	0.02	1.00	0.29%	0.01%	4.65%	99.71%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	14058
	Malfind						ProcInjectionsFind						
win7_Guest	0.13	1.00	0.25%	0.02%	21.92%	99.75%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	8119
win8.1_Guest	0.08	1.00	0.19%	0.02%	14.29%	99.81%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6172
win10_Guest	0.02	1.00	0.28%	0.01%	4.76%	99.72%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	14058
	Threadmap						ProcInjectionsFind						
win7_Guest	0.05	1.00	0.28%	0.02%	10.00%	99.72%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	8119
win8.1_Guest	0.06	1.00	0.28%	0.02%	10.53%	99.72%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6172
win10_Guest	0.03	1.00	0.21%	0.01%	6.25%	99.79%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	14058
	Malfofind						ProcInjectionsFind						
win7_Guest	1.00	1.00	0.00%	0.02%	100.00%	100.00%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	8119
win8.1_Guest	1.00	1.00	0.00%	0.02%	100.00%	100.00%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6172
win10_Guest	1.00	1.00	0.00%	0.01%	100.00%	100.00%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	14058
	FindDLLInj						ProcInjectionsFind						
win7_Guest	##.##	0.00	0.00%	0.00%	##.##	99.99%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	10582
win8.1_Guest	##.##	0.00	0.00%	0.00%	##.##	99.98%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	5919
win10_Guest	1.00	0.83	0.00%	0.01%	88.89%	100.00%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	13477
	With Malware Hiding Technique												
	Malfind						ProcInjectionsFind						
win7_Guest	0.00	0.00	0.41%	0.00%	##.##	99.58%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	8631
win8.1_Guest	0.00	0.00	0.28%	0.00%	##.##	99.71%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	12787
win10_Guest	0.00	0.00	0.20%	0.00%	##.##	99.80%	1.00	1.00	0.00%	0.01%	100.00%	100.00%	14059
	Hollowfind						ProcInjectionsFind						
win7_Guest	0.00	0.00	0.87%	0.00%	##.##	99.11%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	4152
win8.1_Guest	0.00	0.00	0.59%	0.00%	##.##	99.39%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6391
win10_Guest	0.00	0.00	0.07%	0.00%	##.##	99.93%	1.00	1.00	0.00%	0.00%	100.00%	100.00%	34103
	Malfind						ProcInjectionsFind						
win7_Guest	0.00	0.00	0.10%	0.00%	##.##	99.88%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	4152
win8.1_Guest	0.00	0.00	0.19%	0.00%	##.##	99.80%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6391
win10_Guest	0.00	0.00	0.06%	0.00%	##.##	99.94%	1.00	1.00	0.00%	0.00%	100.00%	100.00%	34103
	Threadmap						ProcInjectionsFind						
win7_Guest	0.06	1.00	0.39%	0.02%	11.11%	99.61%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	4152
win8.1_Guest	0.06	1.00	0.27%	0.02%	10.53%	99.73%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6391
win10_Guest	0.00	1.00	0.71%	0.00%	0.82%	99.29%	1.00	1.00	0.00%	0.00%	100.00%	100.00%	34103
	Malfofind						ProcInjectionsFind						
win7_Guest	1.00	1.00	0.00%	0.02%	100.00%	100.00%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	4152
win8.1_Guest	1.00	1.00	0.00%	0.02%	100.00%	100.00%	1.00	1.00	0.00%	0.02%	100.00%	100.00%	6391
win10_Guest	0.50	1.00	0.00%	0.00%	66.67%	100.00%	1.00	1.00	0.00%	0.00%	100.00%	100.00%	34103
	Note: ##.## indicates undefined (Division by zero)												

Figure 8: Evaluation Metrics and Detection Methods are compared

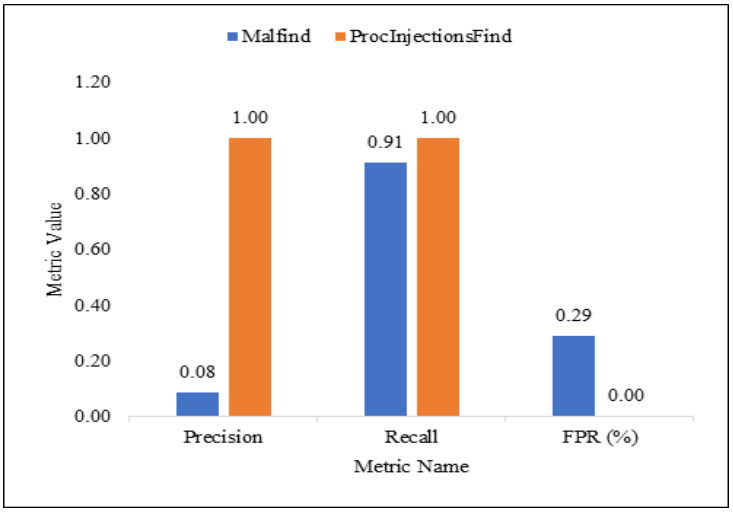

Figure 9A: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Malfind & Procinjectionsfind) on the Win7_VM Environment

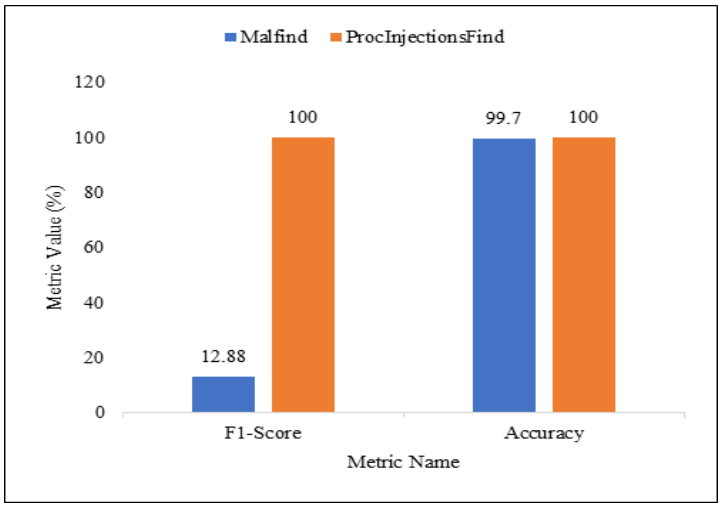

Figure 9B: Comparative Evaluation of F1-Score & Accuracy for Memory Forensics Tools (Malfind & Procinjectionsfind) on the Win7_VM Environment

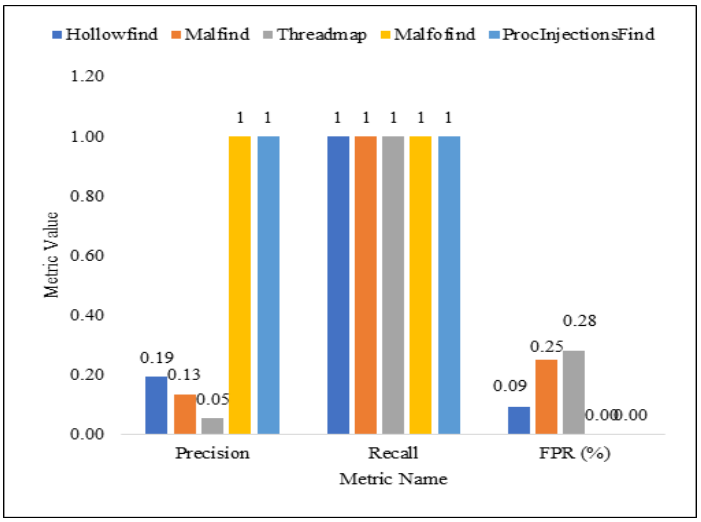

Figure 10A: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) on the Win7_VM Environment

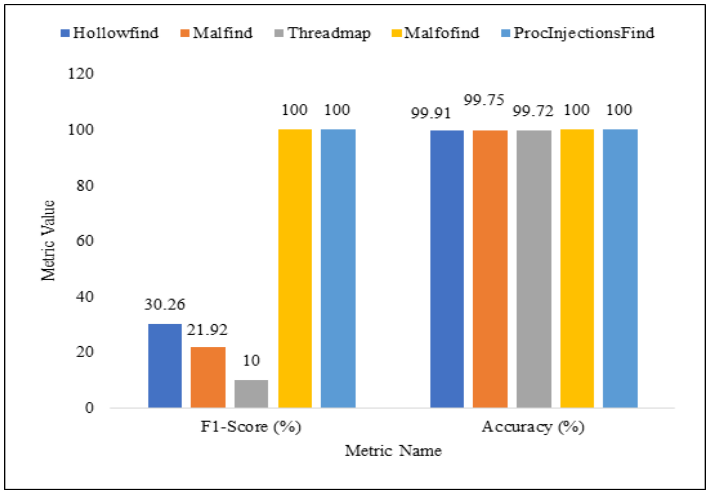

Figure 10B: Comparative Evaluation of F1-Score & Accuracy for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) on the Win7_VM Environment

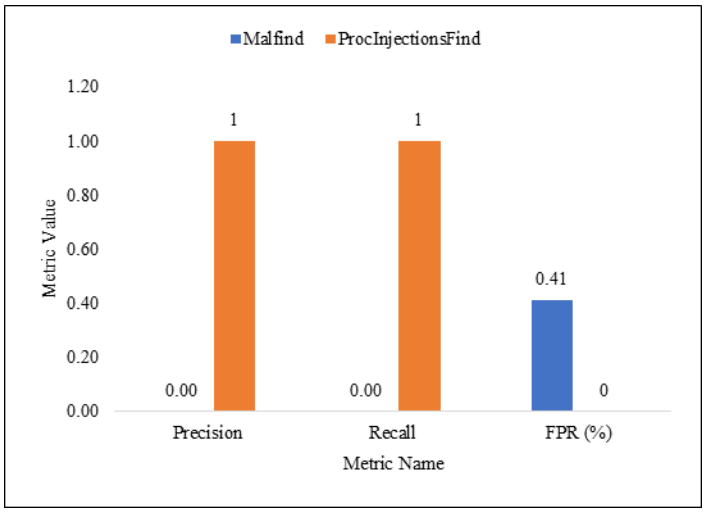

Figure 11A: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Malfind & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win7_VM Environment

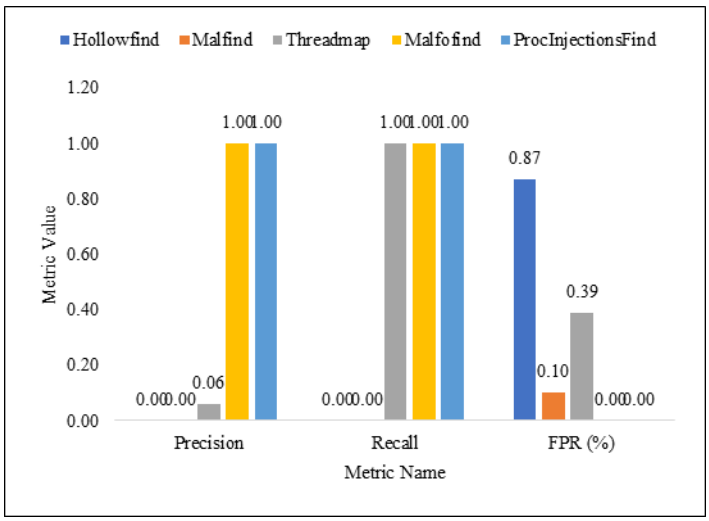

Figure 11B: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win7_VM Environment

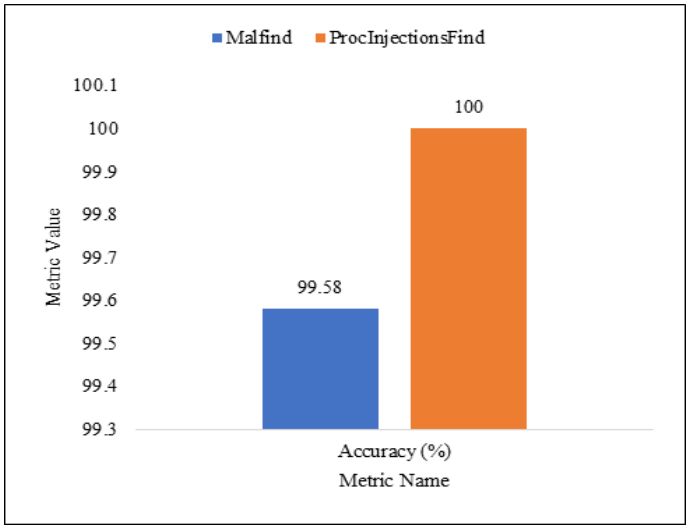

Figure 12A: Comparative Evaluation of Accuracy for Memory Forensics Tools (Malfind & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win7_VM Environment

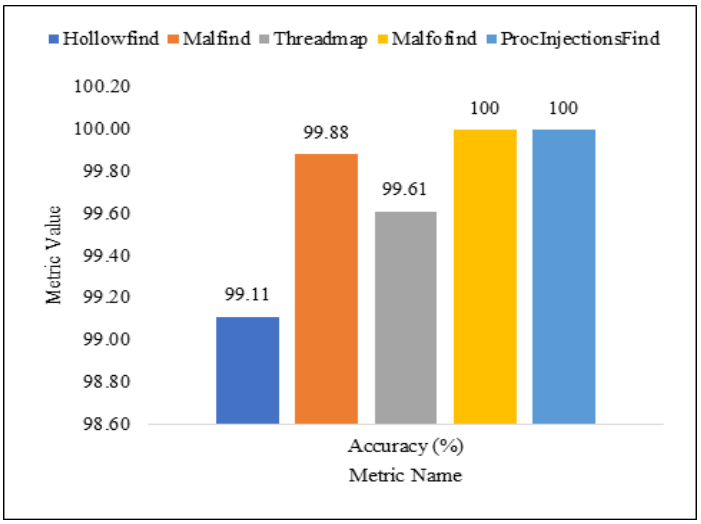

Figure 12B: Comparative Evaluation of Accuracy for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win7_VM Environment

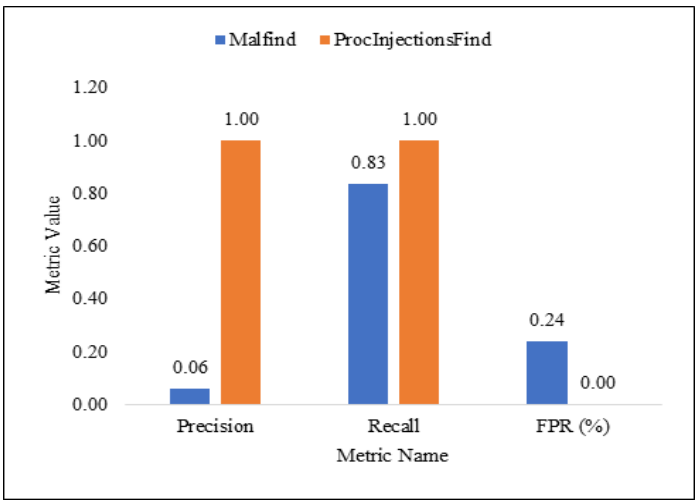

Figure 13A: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Malfind & Procinjectionsfind) on the Win8.1_VM Environment

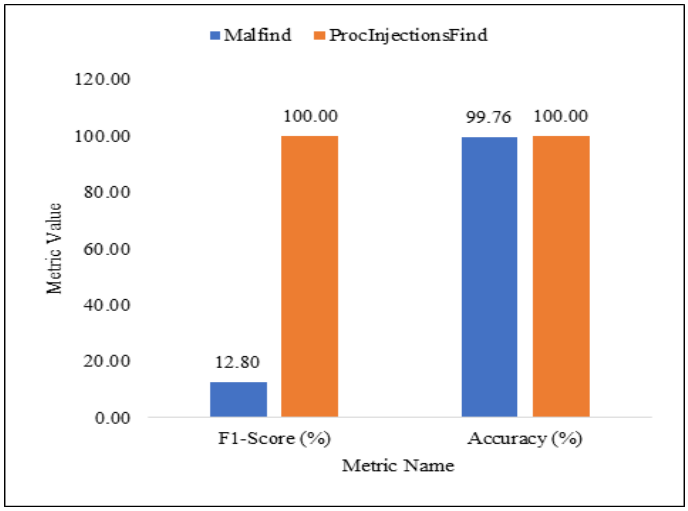

Figure 13B: Comparative Evaluation of F1-Score & Accuracy for Memory Forensics Tools (Malfind & Procinjectionsfind) on the Win8.1_VM Environment

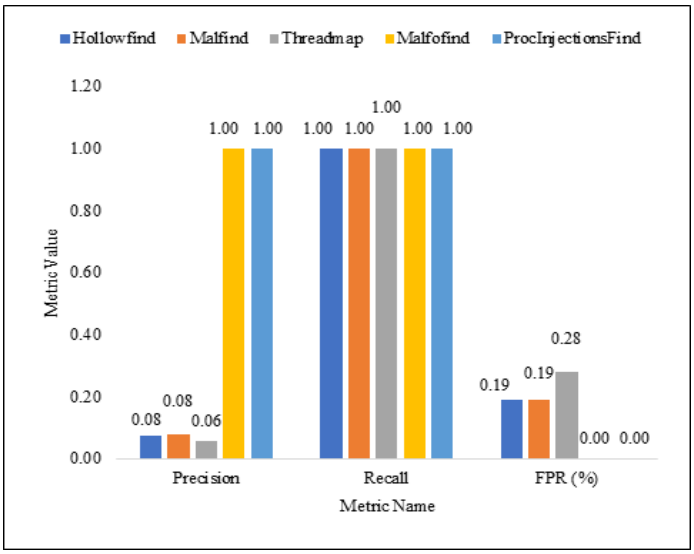

Figure 14A: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) on the Win8.1_VM Environment

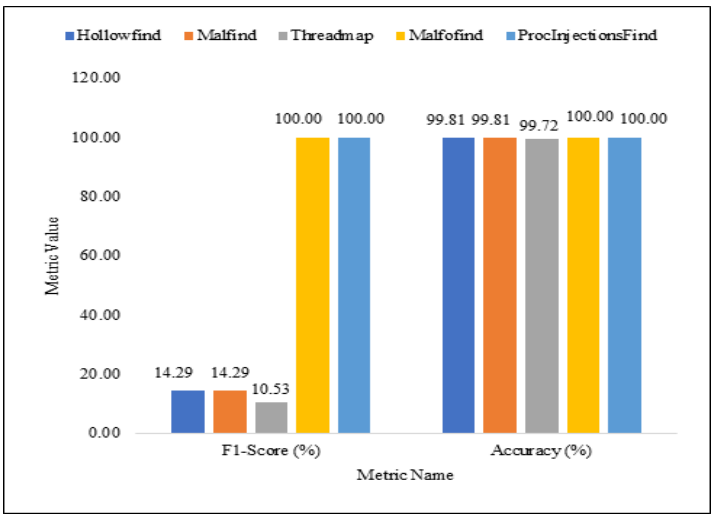

Figure 14B: Comparative Evaluation of F1-Score & Accuracy for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) on the Win8.1_VM Environment

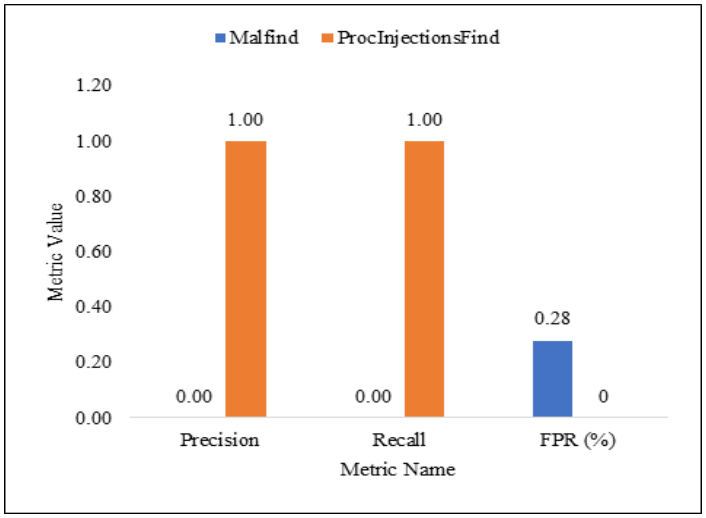

Figure 15A: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Malfind & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win8.1_VM Environment

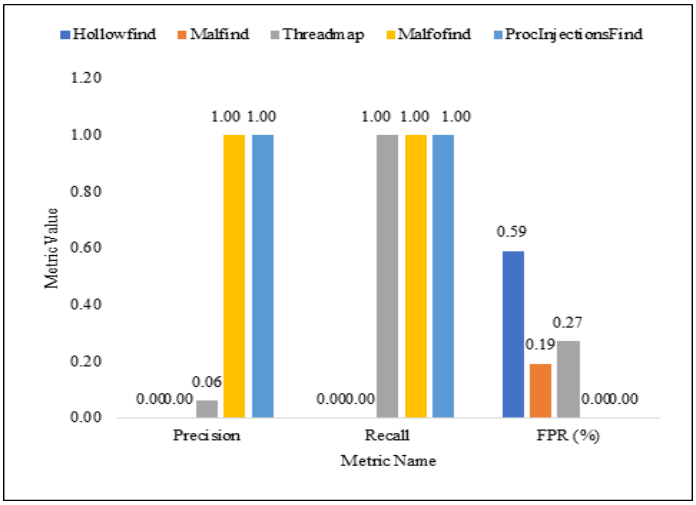

Figure 15B: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win8.1_VM Environment

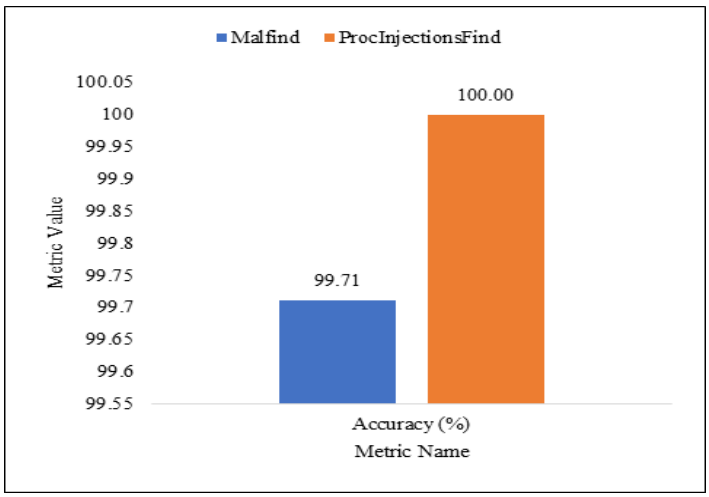

Figure 16A: Comparative Evaluation of Accuracy for Memory Forensics Tools (Malfind & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win8.1_VM Environment

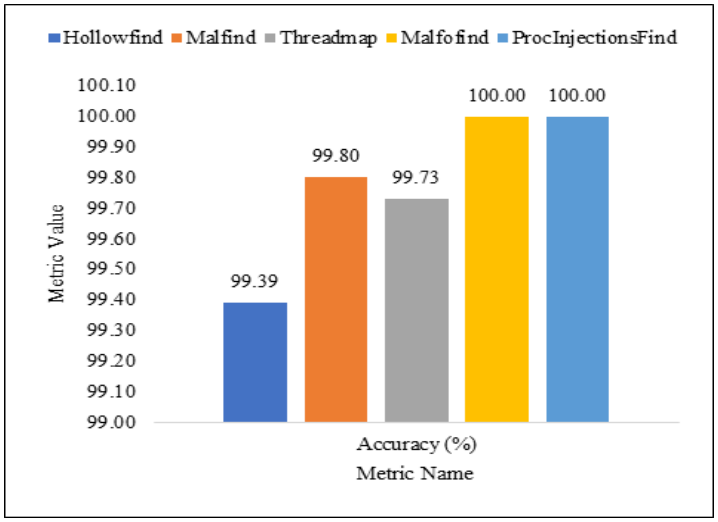

Figure 16B: Comparative Evaluation of Accuracy for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win8.1_VM Environment

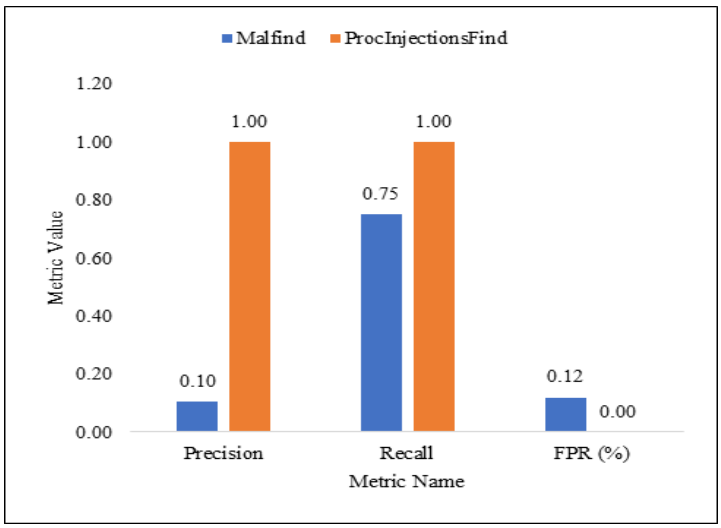

Figure 17A: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Malfind & Procinjectionsfind) on the Win10_VM Environment

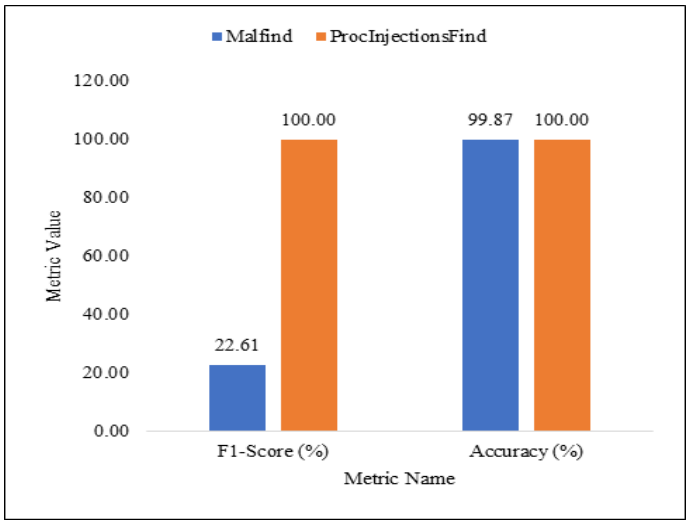

Figure 17B: Comparative Evaluation of F1-Score & Accuracy for Memory Forensics Tools (Malfind & Procinjectionsfind) on the Win10_VM Environment

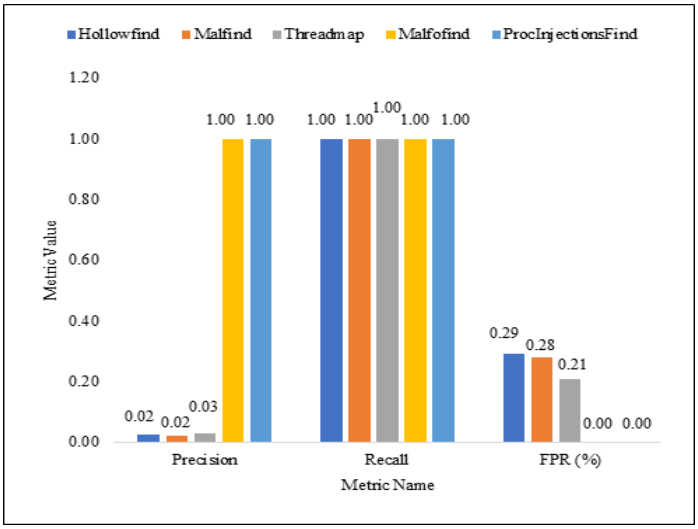

Figure 18A: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) on the Win10_VM Environment

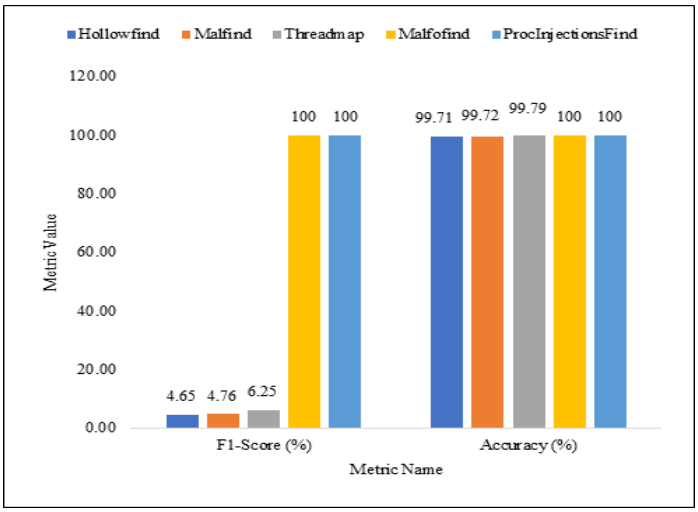

Figure 18B: Comparative Evaluation of F1-Score & Accuracy for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) on the Win10_VM Environment

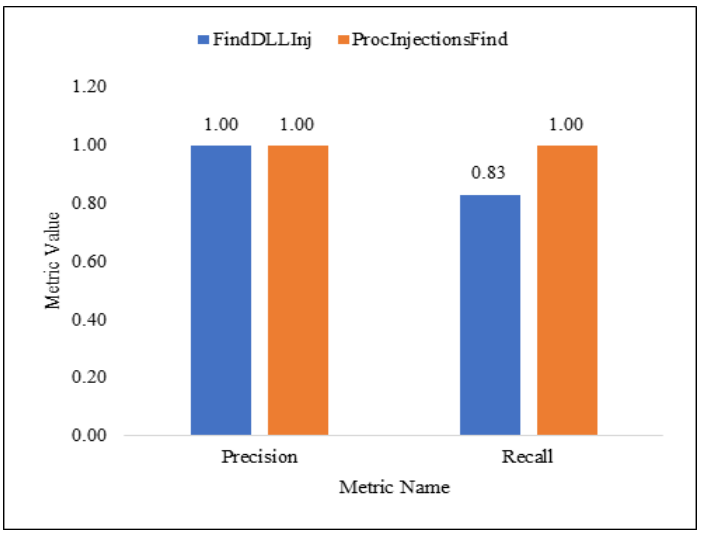

Figure 19A: Comparative Evaluation of Precision & Recall for Memory Forensics Tools (Finddllinj & Procinjectionsfind) on the Win10_VM Environment

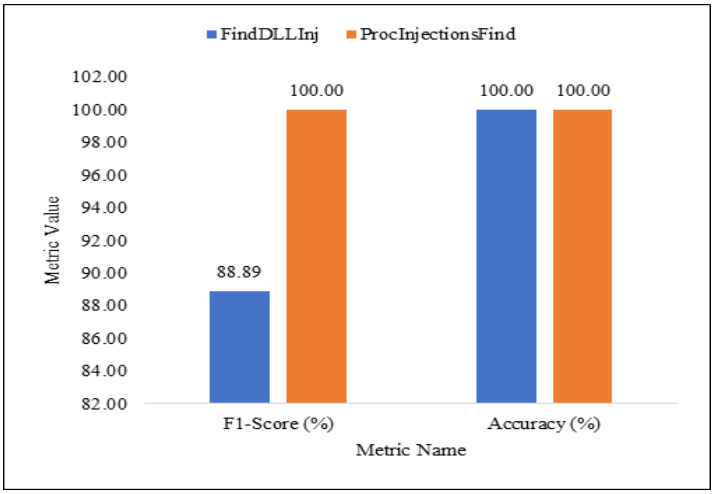

Figure 19B: Comparative Evaluation of F1-Score & Accuracy for Memory Forensics Tools (Finddllinj & Procinjectionsfind) on the Win10_VM Environment

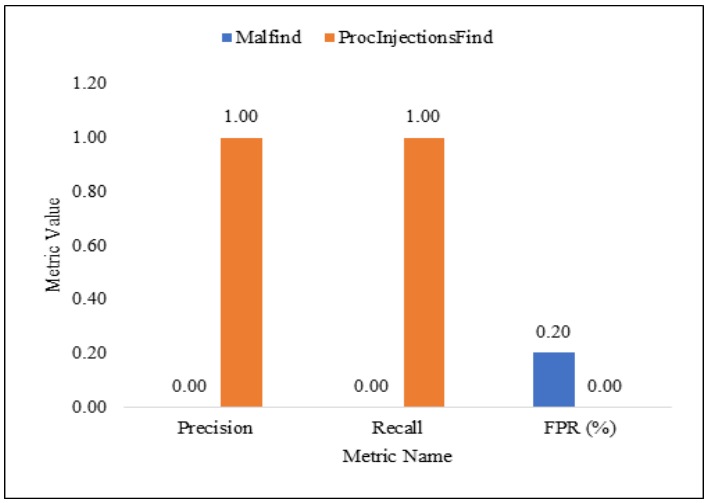

Figure 20A: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Malfind & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win10_VM Environment

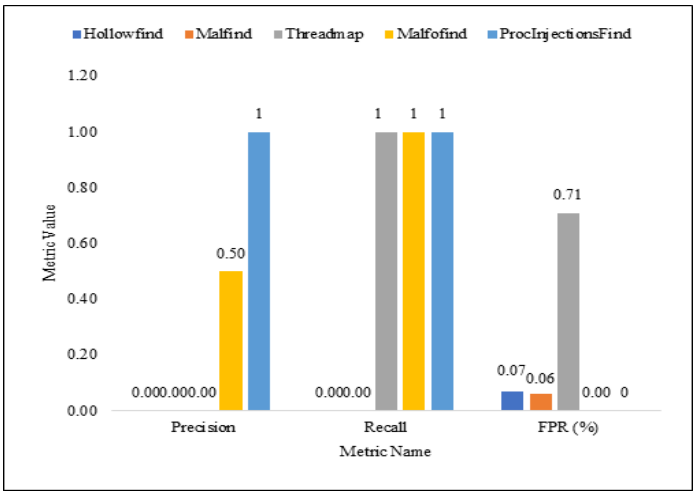

Figure 20B: Comparative Evaluation of Precision, Recall, & FPR for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win10_VM Environment

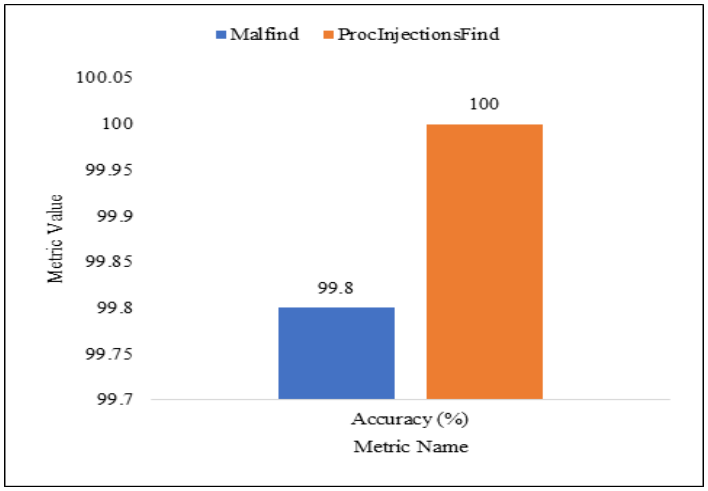

Figure 21A: Comparative Evaluation of Accuracy for Memory Forensics Tools (Malfind & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win10_VM Environment

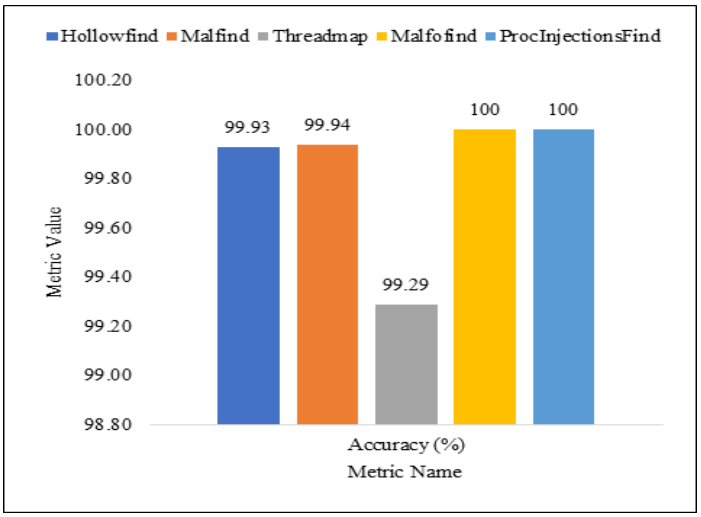

Figure 21B: Comparative Evaluation of Accuracy for Memory Forensics Tools (Hollowfind, Malfind, Threadmap, Malofind, & Procinjectionsfind) Utilizing Malware Hiding Techniques on the Win10_VM Environment

The calculated results, as shown in Figure 8, are graphically represented in Figures 9 to 21. To identify and dump an injected memory location, we utilize the *ProcInjectionsFind* plugin. VirusTotal, a powerful and freely available online malware scanner, can be used to determine whether an executable file is malicious or safe. It offers a free service that uses multiple antivirus engines to scan suspicious files (43). Another free tool, Hybrid Analysis, employs a unique methodology to identify and analyze unknown threats (44). To validate our findings, we can submit the file hash to VirusTotal or Hybrid Analysis (online malware scanners) and hash the memory area identified as injected, as shown in Figure 22.

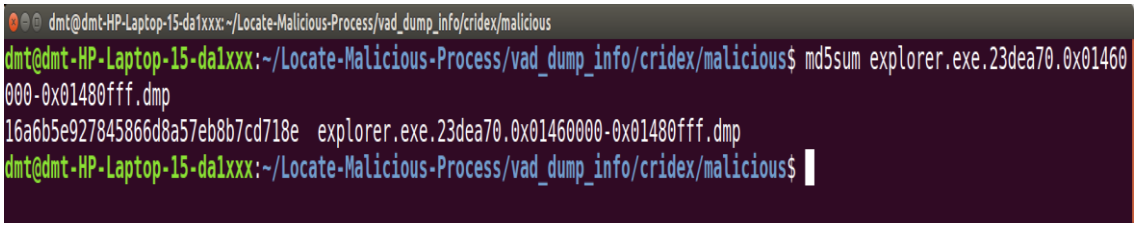

Figure 22: Calculating Suspected File Hash

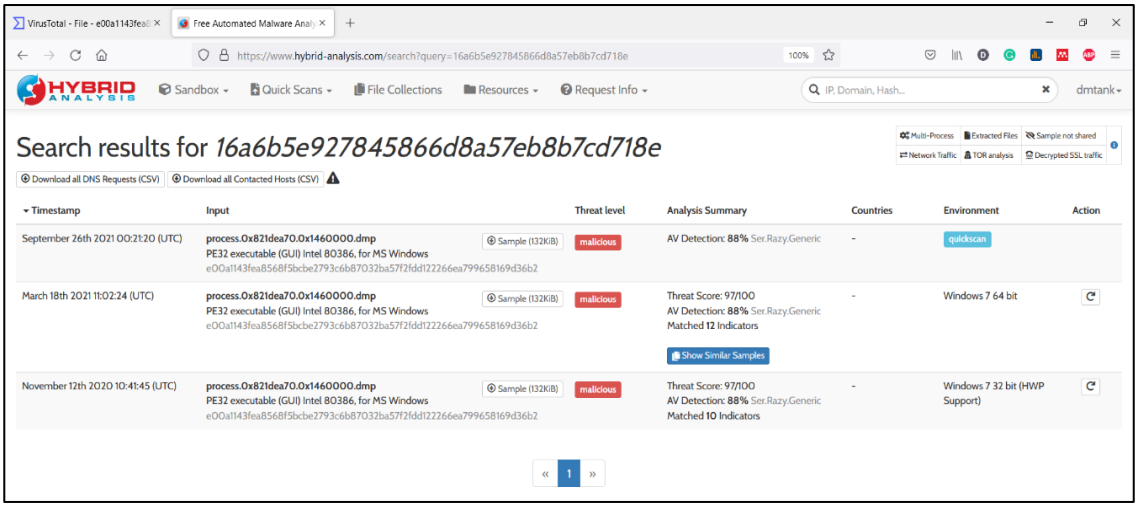

Figure 23: Using Hybrid Analysis to Analyse a Suspicious Memory Area (44)

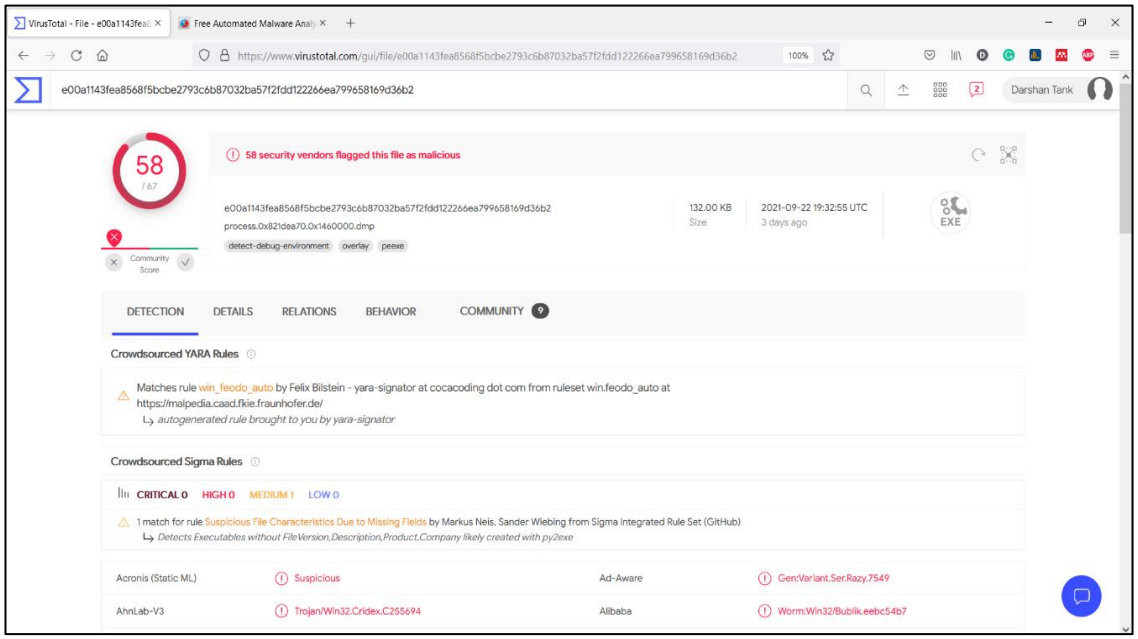

Figure 24: Using Virus Total to Analyze a Suspicious Memory Area (43)

In Figure 23, Hybrid Analysis is used to analyze the file hash of the injected memory areas, while Figure 24 shows the same process performed with VirusTotal. Both online malware scanners flagged the file as malicious.

This study focuses on the identification of eight distinct process injection techniques. In the future, additional process injection techniques could be incorporated, and malware hiding methods not covered in this study could be explored. A created plugin was tested against various Windows-based virtual machines, including Windows 7, Windows 8.1, and Windows 10. The plugin's compatibility and compliance with different Windows operating systems may be evaluated. The Volatility plugin *ProcInjectionsFind* may be integrated with existing process injection detection plugins and ported to Rekall, a widely used memory forensic framework (45). The proposed framework could also be applied to virtual machines running Linux or Mac OS.

Conclusion

In this article, we analyzed and evaluated eight different process injection techniques. The experimental results demonstrate the strong potential of the proposed methods, yielding several valuable insights. Across all three virtual machines (Win7, Win8.1, and Win10), the *VMIPID* model consistently achieves excellent performance across all evaluation metrics. It delivers a 100% positive predictive value,

completeness, F1-score, and accuracy, along with a 0% false-positive rate on each VM. Compared to existing solutions, the experimental data show that the proposed model outperforms others in accuracy, F1-score, and true positive rates while minimizing false positives. Overall, the framework identifies a broader range of malware types and surpasses previous models in every evaluation criterion discussed. Additionally, the *ProcInjectionsFind* module is designed to automatically detect the process injection techniques explored in this study, saving identified injected memory locations to disk for further analysis.

Abbreviations

VM: Virtual Machine, VMM: Virtual Machine Monitor, API: Application Programming Interface, DLL: Dynamic Link Library, PE: Portable Executable, APC: Asynchronous Procedure Call, VAD: Virtual Address Descriptor, PoC: Proof of Concept, TP: True Positive, FP: False Positive, TN: True Negative, FN: False Negative, FPR: False Positive Rate, P: Precision, R: Recall, LSTM: Long Short-Term Memory, KVM: Kernel-based Virtual Machine, IaaS: Infrastructure as a Service.

Acknowledgement

I would like to express my sincere gratitude to Dr. Akshai Aggarwal and Dr. Nirbhay Kumar Chaubey for his constant guidance, continuous support, and valuable inputs.

Author Contributions

The author confirms sole responsibility for the manuscript preparation.

Conflict of Interest

The author declares that there is no conflict of interest.

Ethics Approval

Not applicable.

Funding

No specific grant received from any funding agencies.

References

1. Red Canary. Process Injection - Threat Detection Report. 2025. https://redcanary.com/threat-detection-report/techniques/process-injection/
2. White A. Hashtest Volatility Plugin. 2013. https://github.com/a-white/Hashtest
3. Zhang S, Meng X, Wang L, Xu L, Han X. Secure virtualization environment based on advanced memory introspection. Security and Communication Networks. 2018; 2018(Article ID 9410278):16. https://doi.org/10.1155/2018/9410278
4. Qiao Y, Yang Y, He J, Tang C, Liu Z. CBM: free, automatic malware analysis framework using API call sequences. InKnowledge Engineering and Management: Proceedings of the Seventh International Conference on Intelligent Systems and Knowledge Engineering, Beijing, China. (ISKE 2012). Springer Berlin Heidelberg. 2012 Dec: 225-236.
5. Li C, Xiang Y, Shi J. A model of dynamic malware analysis based on VMI. In Algorithms and Architectures for Parallel Processing: ICA3PP International Workshops and Symposiums, Zhangjiajie, China, November 18-20, 2015, Springer International Publishing. 2015; Proceedings 15:465-475.
6. Kumara MA, Jaidhar CD. Virtual machine introspection based spurious process detection in virtualized cloud computing environment. In 2015 international conference on futuristic trends on computational analysis and knowledge management (ABLAZE). IEEE. 2015:309-315.
7. Volatility Foundation. The Volatility Framework. 2016. http://www.volatilityfoundation.org
8. Monnappa KA. Detecting deceptive process hollowing techniques using hollowfind volatility plugin. 2016. https://cysinfo.com/detecting-deceptivehollowing-techniques/
9. Pék G, Lázár Z, Várnagy Z, Félegyházi M, Buttyán L. Membrane: A posteriori detection of malicious code loading by memory paging analysis. In: ESORICS. LNCS. 2016; 9878:199–216.
10. Hosseini A. Ten process injection techniques: A technical survey of common and trending process injection techniques. 2017. https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process
11. Atkinson J, Desimone J. Taking hunting to the next level: Hunting in memory. SANS Threat Hunting Summit 2017. Get-InjectedThread.ps1. https://gist.github.com/jaredcatkinson/23905d34537ce4b5b1818c3e6405c1d2
12. Barabosch T, Bergmann N, Dombeck A, and Padilla E. Quincy: Detecting host-based code injection attacks in memory dumps. In Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Springer International Publishing. 2017; Proceedings 14:209-229.
13. Block F, Andreas D. Windows memory forensics: Detecting (un) intentionally hidden injected code by examining page table entries. Digital Investigation. 2019:29: S3-S12.
14. Mathew J, Ajay Kumara MA. API call based malware detection approach using recurrent neural network—LSTM. In Intelligent Systems Design and Applications: 18th International Conference on Intelligent Systems Design and Applications (ISDA

2018) held in Vellore, India. Springer International Publishing. 2018 December 6-8; 1:87-99.
15. MITRE ATT&CK. Process Injection, Technique T1055. https://attack.mitre.org/techniques/T1055/
16. Microsoft. Microsoft Malware Classification Challenge (BIG 2015). 2015. https://www.kaggle.com/c/malware-classification
17. AVTEST. The AV-TEST Security Report 2016/17. 2017. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2015-2016.pdf
18. Garnaeva M, Sinitsyn F, Namestnikov Y, Makrushin D, Liskin A. Overall statistics for 2016.https://kasperskycontenthub.com/securelist/files/2016/12/Kaspersky_Security_Bulletin_2016_Statistics_ENG.pdf
19. Symantec. Internet Security Threat Report 21. https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf
20. Secrary. Process Injection Techniques. GitHub – secrary/InjectProc. https://github.com/secrary/InjectProc
21. Tank D. ProcInjectionsFind. GitHub – darshantank/ProcInjectionsFind. https://github.com/darshantank/ProcInjectionsFind
22. Volatility Foundation. Volatility. http://www.volatilityfoundation.org/
23. KVM, https://www.linux-kvm.org/page/Main_Page/
24. LibVMI, http://libvmi.com/
25. Michael HL, Andrew C, Jamie L, AAron W. The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory. Technical Book, Published by John Wiley & Sons, 2014.
26. Microsoft Docs. VirtualProtectEx function (memoryapi.h) Win32 apps. 2018. https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotectex
27. TheEvilBit. Injection techniques. GitHub – theevilbit/injection. https://github.com/theevilbit/injection
28. Fdiskyou. Seven different DLL injection techniques. GitHub – injectAllTheThings. https://github.com/fdiskyou/injectAllTheThings
29. CptGibbon. Windows Process Injection. GitHub – CptGibbon/Windows-Process-Injection. https://github.com/CptGibbon/Windows-Process-Injection
30. Block F. Code Injection Tools. DFRWS-USA-2019/tools at master. GitHub – f-block/DFRWS-USA-2019. https://github.com/f-block/DFRWS-USA-2019/tree/master/tools
31. Fewer S. Reflective DLL Injection. GitHub. https://github.com/stephenfewer/ReflectiveDLLInjection
32. M0n0ph1. Process Hollowing. GitHub. https://github.com/m0n0ph1/Process-Hollowing
33. BreakingMalwareResearch. AtomBombing. GitHub – atom-bombing. https://github.com/BreakingMalwareResearch/atom-bombing
34. KSLSample.vmem. Process hollowing in different approaches. https://www.mediafire.com/file/jlmtbbinanuh6jr/KSLSample.rar
35. Balaoura S. Process Injection Techniques and Detection using the Volatility Framework [Master's thesis]. University of Piraeus. 2018.
36. Volatility Foundation. Volatility's Malfind Plugin. 2017. https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/malfind.py
37. Monnappa KA. HollowFind Volatility Plugin. 2016. https://github.com/monnappa22/HollowFind/blob/master/hollowfind.py
38. KSL Group. Threadmap Volatility Plugin. 2017. https://github.com/kslgroup/threadmap
39. Pshoul D. Malfofind Volatility Plugin. 2017. https://github.com/volatilityfoundation/community/blob/master/DimaPshoul/malfofind.py
40. Volatility Foundation. Volatility's Vadinfo Plugin. 2017. https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/vadinfo.py
41. Volatility Foundation. Volatility's Impscan Plugin. 2017. https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/malware/impscan.py
42. Volatility Foundation. Volatility's Volshell Plugin. 2017. https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/volshell.py
43. VirusTotal. https://www.virustotal.com/
44. Hybrid Analysis. https://www.hybrid-analysis.com/
45. Google Inc. Rekall memory forensic framework. 2018. http://www.rekall-forensic.com