SPVExec and SPVLUExec - A Novel Realtime Defensive Tool for Stealthy Malware Infection

Nicholas Phillips
Department of Computer and Information Sciences
Towson University, Towson, MD, USA
nphill5@students.towson.edu

Aisha Ali-Gombe
Division of Computer Science and Engineering
Louisiana State University, Baton Rouge, LA, USA
aaligombe@lsu.edu

***Abstract*—The vicious cycle of malware attacks on infrastructures and systems has continued to escalate despite organizations' tremendous efforts and resources in preventing and detecting known threats. One reason is that standard reactionary practices such as defense-in-depth are not as adaptive as malware development. By utilizing zero-day system vulnerabilities, malware can successfully subvert preventive measures, infect its targets, establish a persistence strategy, and continue to propagate, thus rendering defensive mechanisms ineffective. In this paper, we propose sterilized persistence vectors (SPVs) - a proactive *Defense by Deception* strategy for mitigating malware infections that leverages a benign rootkit to detect changes in persistence areas. Our approach generates SPVs from infection-stripped malware code and utilizes them as persistent channel blockers for new malware infections. We performed an in-depth evaluation of our approach on Windows systems, versions 7 and 10, and Ubuntu Linux, Desktop, Server, and Core 22.0.04, by infecting them with 2000 different malware samples, 1000 per OS typing, after training the system with 2000 additional samples to fine-tune the hashing. Based on the memory analysis of pre-and post-SPV infections, our results indicate that the proposed approach can successfully defend systems against new infections by rendering the malicious code ineffective and inactive without persistence.**

Keywords*— *Malware; Rootkit; Reverse Engineering; Persistence; Defence by Deception.

I. Introduction

Malware is a continued threat against cyber systems. Characterized by stealthiness, persistence, and mutation, new-generation malware often utilizes various system vulnerabilities for infection and then leverages standard system functionality to maintain persistence. With a suitable persistence strategy, malware can remain active and prolong its existence on a host system. One of the strengths of modern malware development is its adaptability: methodologies mutate rapidly, targeting areas where security measures are weaker or nonexistent. This is true across all systems, but specifically against Windows and Linux platforms. Windows continues to hold the majority of new and unique malware samples due to its position as the most distributed OS in the marketplace, while Linux has been seeing exponential growth, growing 646 percent in samples from 2021 to 2022 [6], as shown in Figure 1. In both related literature and practice, many malware defensive techniques have been proposed - (1) antiviruses and host-based intrusion detection [33], [82], (2) integrity checking [49], [51], (2) integrity checking [31], [42], detection [10], [28], [40], [41], [49], [51], and (3) after-effect or post-mortem analysis [12], [14], [34], [44], [45], [80] of modern malware. However, as evidenced by the continued rise in stealthier attack scenarios, new samples, and variant development [19], these defensive approaches fall short of addressing a growing threat.

The common theme of these techniques is identifying the problem either before infection through signature or anomaly detection or after infection through system scans. Neither provides a general means to stop malware due to its adaptability. These ideas of a responsive or reactionary approach to detecting and preventing malware infections, in many respects, play to malware's strengths. Because of the above mentioned limitations, we propose SPVs - a Defense by Deception approach. Our methodology aims to drastically reduce malware infections by reducing the available areas of persistence for a malicious actor's exploits, including zero-day attacks. Our approach employs malware code segments to defend a target system against future infection, thus serving as a defensive mechanism. This novel technique is a drastic shift from the conventional utilization of malware code for signature detection and fingerprinting. In our proposed approach, we place blockers called SPVs in critical areas of persistence on target systems. These SPVs are persistence and deployment elements stripped from the various malware samples analyzed. Essentially, SPVs prevent a new malware infection by blocking it from writing its own vector or overwriting the persistence vector associated with already established malware. With this approach, malware loses its ability to persist and is prevented from executing its payloads and consequently propagating further. Thus, in this extended version of our prior conference paper [1], we implemented the prototype of our SPV by manually building a library of 200 payload-stripped SPVs into the Defense by Deception code base, which is then compiled into a target system and deploying at system startup. The Defense by Deception code called the *SPVExec* on Windows

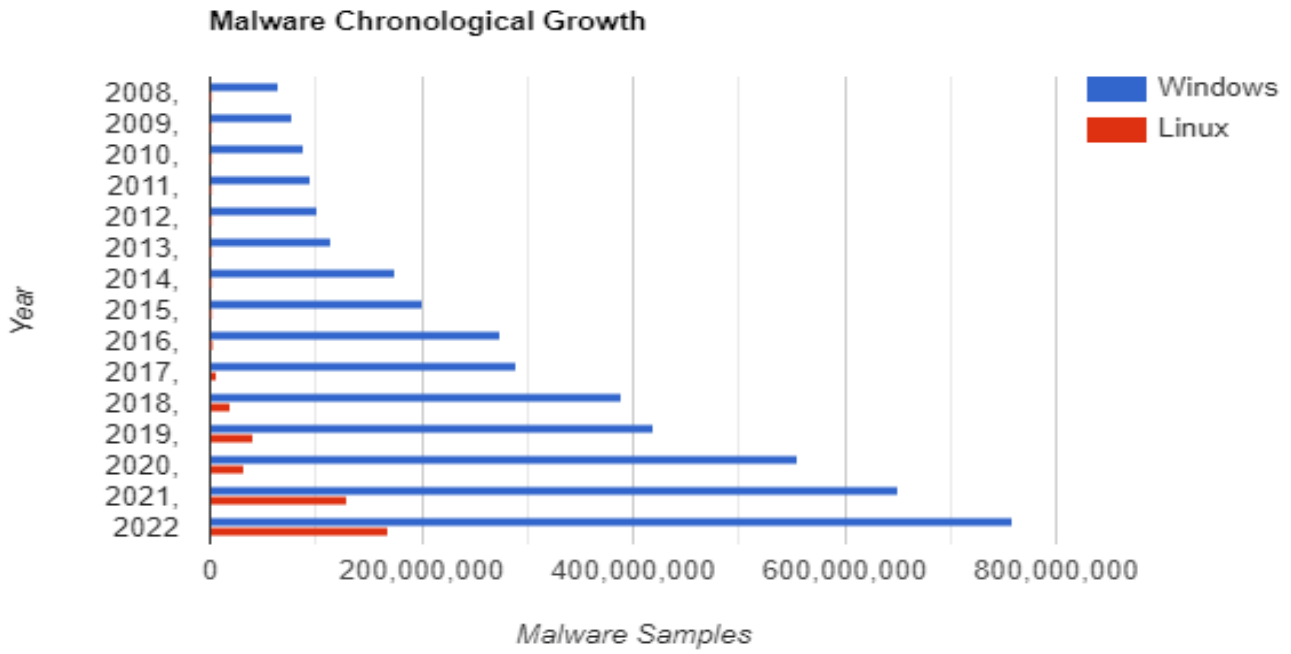

Fig. 1. Malware Growth By Year [6]

and *SPVLUEXEC* on Linux, then administered as a malware defensive apparatus on a need basis automatically at system runtime without user intervention. The empirical results of the evaluation on Windows 7 and 10, as well as Ubuntu Desktop, Server, and Core 22.0.04, for pre- and post-SPV deployment infected with 2000 malware samples, showed that the use of SPVs is a very effective strategy for malware defense. For 99% of the samples in the data set, the SPV Defense by Deception process rendered them inert - the malware sets could not execute their payloads, persist, or propagate. These blocked malware executables are also saved in a "quarantine" zone, allowing for collection and utilization in additional security tool development.

Contributions - Our proposed novel SPV strategy provides the following salient features:

- **Defense Against Malware**: Developing a practical approach to preventing new malware infections by simulating and inventing the perception that the system is already infected.
- **Fully Automated Deployment Process**: The deployment and rendering of the SPVs at runtime are done without human intervention.
- **Efficiency**: The SPV code incurs very minimal overhead on runtime system resources.
- **Usability**: The generated SPVs are reliable and seldom flagged as malware by system defense and antiviral tools. Furthermore, the proposed system allows legitimate programs to be installed without hindrance based on internal whitelisting.
- **Aided Defense Development**: Identified samples are saved and can be further analyzed for other security tool deployments.

The rest of the paper is organized as follows: Section 2 reviews the related literature; Section 3 presents the problem statement and an overview of rootkit infection; Sections 4 and 5 present the implementation of the SPV process and evaluation of our research, respectively; Section 6 details the future work; and Section 7 concludes the paper.

II. Related Work

With the rising threat of malware, the current field of work is constantly evolving, attempting to stem the problem and offer an effective form of Analysis and Defense against it. However, means of malware detection and analysis have grown more stagnant in the last ten years. Literature and current works can be divided into two main categories: Analysis and Detection/Defensive Measures.

A. Malware Analysis

Means of malware analysis have grown more stagnant in the last ten years. Windows malware analysis, in particular, has followed the main analysis structure since the early 2000s. As shown in Tahir, Alsmadi, and El Merabet [46], [47], [48], most of the improvements have been focused on implementing machine learning. This implementation is worked by classifying individual features within malware samples and rejecting

non-specific elements found within a large number of malware samples. While this is an improvement upon the standard malware detection means, there is the limitation that they are process intensive, both in the means of learning algorithms for detection and in scanning the multitude of files presented to the system. The remainder of the analysis techniques has dealt more with means of automation of the malware analysis. These are divided into two distinct areas of study, either generation of automated scripting to automate the analysis and the second is through continued utilization of machine learning to detect similar features within the malware samples.

Current literature in the analysis of malware adheres to a standard approach of malware analysis of a two-phased static and dynamic analysis approach [3]. However, others, such as Lee et al. [56], Dietz et al. [55], and Hwang et al. [57], have developed modifications to this by creating unique and independent analytical platforms. Mehdi et al. proposed Imad, an in-execution analysis platform for malware analysis. Another imported related technique proposed for malware detection is code or system-level instrumentation [4], [5]. Similar to the instrumentation is the development of Sandboxing environment. Mohino et al., in their work proposed MMALE - a Sandbox-like environment for automated execution and analysis of malware. Sandbox environments are also the focal point of Monnappa for attempts to automate malware analysis [49], [58], [59]. As with other Sandbox-based analysis techniques, there is the potential that malware authors can add code elements to detect these environments and not execute their code base. Additionally, malware authors have segmented their code bases and only contained some malicious elements in one download, as found in the research study across multiple malware samples presented in Kiachidis and Baltatzis [53]. This can cause the files in automated environments to flag them as non-malicious, even though they are the start of a malicious campaign.

In the area of machine-learning-based malware detection, the work of Jeon et al. and further developed by Kim et al., proposed deep learning for identifying similar elements of malware structure, such as similar function calls, domain addresses, string structures, etc., to try to determine the possibility of a newer sample being malicious [60], [54]. This is a tremendous step forward and can potentially speed up automated analysis. However, there is a shortcoming. These algorithms take time and a large sample base to learn the patterns in the malicious code. By the time they can identify the current trend of malicious code, malware authors can find new means to bypass these defensive measures, as shown in the evolution study presented by Cozzi et al. [61].

The SPV code does not have several limitations in the current analysis research elements. Instead of parsing through many code elements to determine maliciousness, the SPVs can target the smaller area of the malware persistence. This reduces thousands, if not millions, of code lines to a few major persistence areas. Additionally, as the SPV code is not attempting to stop the execution of the code through the current means, such as through the Sandbox analysis or process identification, several malware defensive measures are not utilized. The SPVs only need to worry about the raw malware code, not the identification of packers or encryption; it does not need to worry about antiVM and anti-RE capabilities. This leads to an extensive reduction in time and resources, which would be wasted in the analysis process. This paper presents a new SPV Defense by Deception strategy that leverages sterilized persistence vectors extracted from a real malware corpus to block potential malware infections. Our system utilizes code from malware samples, not as signatures but as defensive strategies that stop new infections from attempting to write into persistence regions. Compared to existing COTs and techniques described in the literature for malware detection and prevention, our approach is designed to be more robust and versatile, with the ability to block malware both on bare hardware and in virtualized environments. Additionally, our methodology does not require a signature or agnostic of the target malware behavior. Through an in-depth evaluation of 2000 malware samples with pre- and post-SPV infection, we demonstrate that our proposed SPV Defense by Deception mechanism can effectively defend systems against malware infections with 1-3 percent CPU and memory overhead while not limiting the ability to install legitimate programs properly.

B. Malware Detection

Malware detection methodologies can be broken down into Host-based, Hypervisor-based, and Post-mortem analysis.

1) Host-based Detection: The more traditional technique for rootkit detection is a host-based intrusion detection system that checks for anomalies or footprints of known malware. For example, the System Virginity Verifier verifies the validity of in-memory code for critical system DLLs and kernel modules; [39] checks the legitimacy of every kernel driver before it is loaded into the operating system; Panorama [40] is designed to perform behavioral runtime tracking, and SBCFI [26] detects threats by examining the control flow integrity of the kernel code. A smaller subset of methods, such as Autovac, utilizes forensics snapshot comparison engines to detect the execution of malware on the system to prevent it [38]. Other host-based rootkit detection systems include HookFinder [40] and HookMap [36]. These techniques use systematic approaches to detect and remove malware hooks in target operating systems. One offshoot of the pre-infection defensive measures proposed by Das et al. and further developed by Kedrowitsch et al. is the deployment of Docker containers as Honeypots and analyzing the behavior of the attacks to fine-tune defensive measures. By presenting these areas as more appealing targets for network attacks, security professionals can fine-tune the defensive measures on their main network components to avoid compromise [73], [75]. Shahzad et al. presented a means of detecting running malware on a Linux system by comparing the task structure of the Linux processes. By loading the kernel structures of a process, they have identified whether it is malicious or not with a minimal

impact on the system's overhead [64]. The shortcoming of this research is that once the specific modules have been removed, as shown in their evaluation, the accuracy begins to fail. Malware code has proven as it evolves to have the ability to start targeting elements that are preventing its infection, such as those proven in the study by Ngo et al. [81].

One major drawback of traditional host-based detection methodologies is the ability of the malicious entity to evade detection since it is running with the same level of privilege as the detection systems. Given that most of these tools are designed to probe for the rootkit signature and/or behavior, malware can easily subvert this effort as it evolves to have the ability to start targeting elements that are preventing its infection, such as those proven in the study by Ngo et al. [81]. The SPV code does not scan for malware footprint or traits; instead, it takes the more aggressive approach of hijacking the persistence area of a potential rootkit, leaving the malware with no place to hide. Furthermore, the SPV code is built so the malware cannot eject or terminate its process.

2) Hypervisor-based Detection: Integrity checking is a technique that requires continuous monitoring of the kernel code for changes to signatures, control flow, and kernel data structures. For kernel-level rootkits, the most practical approach for maintaining kernel integrity is hypervisor-based systems that leverage virtual machine introspection (VMI) [2], [17], [18], [28], [30], [31], [42], [43]. VMI systems and tools are built to introspect the virtual environment through the hypervisor. Since the hypervisor runs at a much lower level than the virtual OS, these mechanisms are often seen as effective for detecting rootkits and monitoring their behavior. However, their major limitation is that they target only virtualized environments and cloud infrastructures and cannot be applied to introspect real hardware-based systems. Moreover, most kernel integrity-check-based systems are susceptible to return-oriented rootkit attacks [17]. Asmitha and Vinod presented a means of malware classification based on eXtended-symmetric uncertainty. The work of [68] utilize entropy to rank features of different classes of malware and compare them against known features for malware identification. While promising research with the ability to detect nearly all cases with 99% accuracy, however, this work is limited to leveraging only the entropy. As shown in the cases of higher-level rootkits, such as those as part of the research in Raju et al. and Wang, newer malware samples can corrupt these algorithms that depend on static features [70]. Xu et al. proposed MIDAS, a real-time behavior auditing to detect malware across IoT devices. In their research, they developed a framework that analyzes the individual elements of executables as they are on the system. Once compared to the baseline, those that match the malware are flagged as malicious [71]. This research is vital because it checks for malicious elements as the sample run. However, this brings about its shortcoming: against elements of malicious code that do not match the pattern and benign software that does match the auditing requirements. Gomez et al., in their forensic analysis of IoT malware, found that several samples have become adapted to masquerading as benign programs, allowing the bypass of defensive measures [80]. The other central research element for pre-infection is the measurement of secure system installations. Sun et al. propose monitoring and protection elements during the installation phase of software deployment to minimize infection during the software deployments [76]. While this is an excellent idea, it comes to some of the same problems as other installation protection items, such as Antiviruses. Malware can compromise these checks and gain the access needed to complete their installation. Even some more robust defensive engines meant to stop improper software loading, such as SecureBoot, have been compromised, as proven in Alrawi et al. [85]. Methods used to detect the integrity of a system have been proven to be limited based on the existence of UEFI bootkits. These malicious code elements work by making the operating system accept that malicious code pieces are a legitimate portion of the system's code [11], [16], [27], [71]. With our proposed SPV Defense by Deception process, the system is designed to execute on both hardware and virtual systems, thus circumventing this limitation.

3) Post-mortem Analysis: The last category of rootkit detection methods is post-mortem analysis systems, designed to analyze the after-effects of rootkit execution. These forms of analysis are often passive and involve examining kernel memory snapshots looking for evidence of rootkit infection, persistence, and stealth. Disk forensics tools, such as [12], [14], [34], [44] are used for general system incident response. These tools can examine a target system for file modifications, running processes, network activities, and more. In much the same way as integrity checkers, disk forensics tools are limited by their coverage. If malicious code hides its elements in specific system files or structures, these will generally be missed by the post-mortem analysis [9]. With memory forensics, post-mortem analysis is carried out on a snapshot of volatile memory. The most widely used memory analysis framework is the volatility framework [45]. This methodology is restricted to current events and processes. Terminated malware behaviors cannot be retrieved. Furthermore, modern rootkits can evade detection from memory forensics tools by performing direct kernel object manipulations that hide their presence from registering in major kernel structures or by altering the memory collection or imaging process as a whole [21]. The SPV code does not scan for malware footprint or traits; instead, it takes the more aggressive approach of hijacking the persistence area of a potential rootkit, leaving the malware with no place to hide. Furthermore, the SPV code is built so the malware cannot eject or terminate its process. Compared to a more passive malware detection approach, our SPV process is an offensive approach that prevents malware infections in real-time. The SPVs are designed to block malware from executing, thus forcing the malware to terminate its process.

III. Problem Statement

Malware has always had the strength of its adaptability, which enables it to use multiple mechanisms to infect and

evade detection or bypass many of the elements of system defense [15]. Either using out-of-date signatures, exploiting unknown vulnerabilities, or targeting the weakest link - the human - malware will cause the defense to fail, even if only one falls short. Current detection and prevention tools are significantly disadvantaged because malware evolves faster than defense tools. Stealthy zero-day attacks are becoming increasingly common, and it takes only a single unknown offense or human error to bring down the whole gauntlet of defenses [20].

Thus, we present the SPV Defense by Deception process - a novel technique that attempts to hijack the areas in which malware, in general, and rootkits, in particular, can land their persistence vectors. Rootkit persistence vectors are specifically selected in this research because they are the most common persistence mechanism used by malware of all families [15].

The motivation to use persistence vectors stems from the fact that, in practice, infection vectors are unpredictable, meaning that exploits, especially zero-day exploits used to launch malware attacks, evolve with newly found vulnerabilities. However, the persistence vectors with which the malware maintains a presence on a victim's machine are often deterministic. As such, the most effective way to curtail rootkit infections and ultimately render them ineffective is to place blockers in the potentially persistent channels in the system. Long-term malware campaigns, specifically those utilized by Advanced Persistent Threats (APTs), do not wish to bring a targeted system down immediately. Instead, they want to complete target profiling against the network, exfiltrate sensitive data, and work further into the system. It can sometimes be months before the threat actors launch their final attack target. For this, they require a means to remain in the system. They require persistence. One of the longest of these types of campaigns was the Harkonnen Operation. Malicious actors could utilize their malware persistence and operate on a network for twelve years before they were finally detected. During this time, the malware implanted could assist with further target development, stealing essential data, such as corporate financial documentation, and pilfering money for the attackers [23]. Our approach injects the SPV code into the system startup process and can be rendered on bare hardware and virtualized environments. The SPV process blocks all malware by first detecting in real-time when the malware deploys its persistence vector. It then hijacks the malware area of persistence by automatically selecting and overwriting the malware code with certain SPVs. This process consistently blocks target malware from maintaining a presence on a defended system. Although our approach is currently limited to the categories of malware containing persistence vectors, "fileless" malware has only existed substantially since 2002. It is still not utilized as substantially as persistent malware [87], thus making this limitation minimal.

IV. The SPV - Defense by Deception Process

The SPV process is a code implementation of "sterilized" malware or malware with malicious content removed and injected via a common infection mechanism. It is a technique designed to prevent malware persistence on a system. SPV process involves injecting a malware persistence vector into a clean system to block potential malware from maintaining access. This process combines standing entries consisting of stripped malware persistence vectors and infection code fragments with filler code. With SPVs, the malicious payload code fragments are entirely stripped off while retaining the core elements of malware, such as API hooking, process manipulation, and service control in the SPV. Our proposed approach's workflow comprises the SPV development phase and SPVExec code deployment and integration.

A. Development Phase

This phase begins with identifying and extracting malware persistence vectors and then reprogramming the extracted persistence code fragments into one executable module.

1) Persistence Extraction: The mechanism in this stage requires manual extraction through detailed reverse engineering. We completed our reverse engineering via both static and dynamic malware analysis techniques. Malicious samples were collected from virus repositories: VirusShare [1] and Malshare [25]. One thousand samples per main OS platform were run through the two phases of reverse engineering. This was completed in a series of virtualized Windows and Linux environments. Two copies each were utilized, one for dynamic analysis and one for static analysis. These systems were identified as Testbed-1 for dynamic analysis and Testbed-2 for static analysis. Each machine had two 2.4 GHz cores and 4 GB RAM. For each target malware, we ran the sample against an unpacker for each target malware to remove any possible common packers and cryptors, leaving behind the bare-bones malware code that the analysis tools would evaluate. In this initial phase, the stripped malware code was executed in a custom-built dynamic analysis sandbox running ProcMon, CaptureBat, CFF Explorer, API Monitor, and RegShot for the Windows-based samples. The Linux samples were analyzed with X tools.

This static analysis identifies a specific part of the executable targeted during the dynamic analysis phase. Such code constructs include specific API invocation, non-normal network traffic, registry modification, and file creation. We executed the samples through a debugger and disassembler for the dynamic analysis, specifically IDAPro and OllyDbg (GDB for the Linux-based samples), targeting the identified elements in static analysis. Then, utilizing the HexRay program within IDAPro, the code section was removed and converted to a C program snippet.

2) SPV Generation: With the elements of persistence and infection identified and removed from the base malware code, we developed the SPVs. Since the identified persistence code was disassembled, we began this stage by converting the assembly code into C programming language. Upon extraction, PVs reflect specifically that individual sample of the malware, but additionally can be utilized against the majority of the

```
      // Installing the boot loader
      Status = BkSetupWithPayload(BootLoader, BootSize, Payload, PayloadSize);
      vFree(BootLoader);

      if (Status != NO_ERROR)
      {
           DbgPrint("BKSETUP: Installation failed because of unknown reason.\n");
           break;
      }

      // Creating program key to mark that we were installed
      if (RegCreateKey(HKEY_LOCAL_MACHINE, KeyName, &hKey) == NO_ERROR)
           RegCloseKey(hKey);

      Status = NO_ERROR;
      DbgPrint("BKSETUP: Successfully installed.\n");
} while(FALSE);

if (hMutex)
     CloseHandle(hMutex);

if (Payload)
     vFree(Payload);

if (KeyName)
     vFree(KeyName);

if (MutexName)
     vFree(MutexName);

if (IsExe)
     DoSelfDelete();
```

Fig. 2. Windows Extracted PV

samples of that specific malware family of that generation. For example, an extracted persistence vector from Zeus Botnet would identify that specific file and the different samples in that same generation of Zeus. Specific PVs could also be utilized against other families, dependent upon source code sampling used by the author upon its creation. Prior or future versions would require additional PV extractions depending on the evolution of the malware sample. Figure 2 show the PV extracted from Necurs Rootkit. The Necurs sample persists using multiple techniques, notably boot and registry modification implementation. These specific PVs were identified through our two-phased reverse engineering and exported for inclusion in the SPV library.

These 800 individual SPV extracted from the 1000 malware samples are loaded into the SPV Defense, including the deployment code elements. These were selected as they covered the range of persistence vectors and allowed for the broad defense of the SPVs when deployed on the system.

To build a more robust SPV defensive process, we developed an SPV library consisting of multiple SPVs.

B. SPVExec Implementation

The proposed SPV mechanism uses the extracted PVs to form a benign rootkit called SPVLUEXEC. Additional persistence scanning mechanisms were added to the code to overwrite non-whitelisted persistence modifications. Another functionality was implemented to deploy a FAT32 file system within the bootstrap code section of the system. This area was used for the SPV library, whitelisting, and the SPV Defense base code. The data remained encrypted, utilizing a 256-bit key to protect against registering on scans.

The SPVEXEC and SPVLUEXEC were implemented as single Windows EXE and Linux ELF executable programs loaded alongside the essential boot files at system startup. Each prototype is approximately 1800 lines of code in the C programming language. It is structured as follows:

- **SPV Database -** SPVs randomized for deployment across the system.
- **Defensive Measures -** Defensive elements to protect SPV code base from scans and identification of malicious code and legitimate defensive measures.
- **Dynamic White- and Blacklisting -** Included a listing of approved and disallowed changes that can be implemented on the system.
- **Analysis Mechanism -** Hash comparison against the deployed SPVs and values found in their areas.
- **SPV Launcher -** Mechanism that handles the deployment of the SPVs into their specific areas of persistence.
- **Quarantine Zone -** Area for tagged code samples for tool development.

After successfully loading the SPVExec, the persistence vectors employ two scanning techniques to validate and ensure that an intruder has not altered the injected SPVs at runtime. The first check utilizes time-based scans, similar to those employed by current protective tools. In the current implementation, this check runs a scan every second. Our

secondary scanning technique leverages API hooking to check for malware intrusion. The SPV instances are injected into kernel-level processes. Any attempts to access the protected area of persistence are redirected to one of the SPV Defended DLLs. Both scanning techniques utilize hash lookups. During SPV code deployment, a hashmap of the injected SPVs and the region of persistence are stored. The rewriters dynamically replace code elements within the SPVExec codebase and are designed to look up any changes to the injected SPVs. The dynamically computed hashes of the injected vectors are then compared against the SPVs expected to be in those regions. If no match is returned, the code rewrites those SPVs as expected.

V. Evaluation of the SPV Defense by Deception Process

We evaluate the effectiveness of our proposed SPV defense mechanism by performing four major experiments that answered the following questions:

- **Persistence of the SPV Defense process -** Can the SPV Defense survive and persist through system restarts and power removal?
- **Defense against malware -** Can the SPVs be used as an effective strategy to block potential malware from writing to protected areas of persistence?
- **Defense Through Deception -** Does the SPV Defense identify as malware to other malware and legitimate to legitimate programs?
- **System Performance -** Can the SPV Defense process be used as an efficient apparatus for system defense without depleting system resources?
- **Whitlisting Capability -** Does the SPV Defense allow legitimate programs to install without being replaced with SPV code?
- **Defense Development -** Does the SPV Defense aid with other defensive tool development?

A. Test Environment

To test SPVs across operating systems, we generated Testbed-3 and Testbed-4, utilizing Windows testbeds using the same baseline operating systems as in the persistence extraction phase, i.e., Windows for Windows and Ubuntu for Linux. They both contain sets of virtual machines and bare metal with two 2.4 GHz cores and 4 GB RAM. Testbed-1 remained at the same level of security as that of the persistence extraction environment; this removes the chance of malware failing to infect because of patching or security tools. Unlike in persistence extraction, however, this testbed has most of its nonsecurity functionality restored. This allows the system to act similarly to a standard user system that would be part of a normal network. Testbed-2, Testbed-3, and Testbed-4 are equipped with system security monitoring tools, such as operating system inbuilt Defense, i.e., Host-based Security System, and other commercial off-the-shelf antivirus products appropriate to the respective OS. For all the testbeds, user programs were installed to simulate a working system on a network, and typical applications were often targeted for compromise. To provide better containment during our analysis and testing, we implemented FakeDNS to resolve any network traffic.

B. Post-Mortem Analysis Environment

We leverage an in-depth analysis of the target systems' extracted memory snapshots to evaluate the overall SPV Defense process's accuracy, resilience, and performance. To perform forensic examinations of the memory dumps, we created a separate system equipped with FTK (Linux Memory Extractor (LIME) for Linux) and Volatility. The collection tools were also loaded on a USB to protect the data from being compromised after a malware infection. This allowed the acquisition to have a limited impact on the system while keeping the tools from being impacted by any potential built-in anti-analysis approach.

C. Experiments

1) Experiment I: Persistence: Vital to the functionality of the SPVExec benign rootkit is its ability to maintain persistence. We took the Testbed-2 system post-SPV deployment to test this functionality and saved it as "X-Security-TestingPost." We then performed a power cycle. A start-up alert was entered into the code to present a popup if the SPV remained intact. This alert displays the first SPV value and a "Hello World" message. Upon powering the system, a memory collection was completed utilizing FTK Imager. Volatility Memory Framework processed the memory image with the following plugins: psxview, malfind, ldrmodules, apihooks, dlldump, procdump, and threads. Processes and Dynamic Link Libraries (DLLs) of the SPVExec proved that it could maintain its persistence, and a popup was displayed.

For the Linux-based systems, a start-up alert was entered into the code to present a terminal displaying the first SPV value and a "Hello World" message if the SPV remained intact. Upon powering the system on, a memory collection was completed utilizing LIME. Volatility Memory Framework processed the memory image with the following plugins: Linux_psaux, Linux_malfind, Linux_pstree, Linux_kernel_opened_files, Linux_hidden_modules, Linux_procdump, and Linux_bash. Processes and shared objects of the SPVLUExec were found that proved that it could maintain its persistence and a terminal with the defined items was displayed.

Presented below in Figures 3 and 4 are the outputs from the Malfind upon the memory collection of the respective system, showing the SPV code still operating.

2) Experiment II-A: Defense Against Malware: The primary functionality of the SPVExec is its ability to stop malware attacks against the system. To provide a sufficient test of the defensive capabilities of our approach, we conducted this experiment with 1000 malware samples with diverse infection and persistence vectors and varying degrees of

```
Time Stamp: Tue Mar  7 20:22:23 2023
"C:\Users\nicho\OneDrive\Desktop\PhD\VolatilityWorkbench\vol.exe" -f "C:\Users\nicho\OneDrive\Desktop\PhD\memdump.mem" windows.malfind.Malfind
Please wait, this may take a few minutes.

Volatility 3 Framework 2.0.1
PID Process Start VPN End VPN Tag Protection CommitCharge PrivateMemory File output Hexdump Disasm
1692 dwm.exe 0x264729f0000 0x264729f0fff VadS PAGE_EXECUTE_READWRITE 1 1 Disabled
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
60 04 00 00 40 00 00 00 `...@...
00 00 9f 72 64 02 00 00 ...rd...
1692 dwm.exe 0x26478a80000 0x26478a80fff VadS PAGE_EXECUTE_READWRITE 1 1 Disabled
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
a1 03 00 00 40 00 00 00 ....@...
00 00 a8 78 64 02 00 00 ...xd...
4748 XtuService.exe 0x27ad6e00000 0x27ad6e0ffff VadS PAGE_EXECUTE_READWRITE 2 1 Disabled
00 00 00 00 00 00 00 00 ........
27 9e dc 12 6e 3a 00 01 '...n:..
ee ff ee ff 02 00 00 00 ........
20 01 e0 d6 7a 02 00 00  ...z...
20 01 e0 d6 7a 02 00 00  ...z...
00 00 e0 d6 7a 02 00 00  ...z...
00 00 e0 d6 7a 02 00 00  ...z...
0f 00 00 00 00 00 00 00 ........ 00 00 00 00 00 00 00 00 27 9e dc 12 6e 3a 00 01 ee ff ee ff 02 00 00 00 20 01 e0 d6 7a 02 00 00 20 01 e0 d6 7a 02 00 00 00 00 e0 d6 7a 02 00 00 00 00 e0 d6 7a 02 00 00 0f 00 00 00 00 00 00 00
4748 XtuService.exe 0x27ad6db0000 0x27ad6dbffff VadS PAGE_EXECUTE_READWRITE 2 1 Disabled
00 00 00 00 00 00 00 00 ........
97 cf 1e b2 fe 3b 00 01 .....;..
ee ff ee ff 02 00 00 00 ........
20 01 db d6 7a 02 00 00  ...z...
20 01 db d6 7a 02 00 00  ...z...
00 00 db d6 7a 02 00 00  ...z...
00 00 db d6 7a 02 00 00  ...z...
0f 00 00 00 00 00 00 00 ........ 00 00 00 00 00 00 00 00 97 cf 1e b2 fe 3b 00 01 ee ff ee ff 02 00 00 00 20 01 db d6 7a 02 00 00 20 01 db d6 7a 02 00 00 00 00 db d6 7a 02 00 00 00 00 db d6 7a 02 00 00 0f 00 00 00 00 00 00 00
4748 XtuService.exe 0x7ff494fd0000 0x7ff49506ffff VadS PAGE_EXECUTE_READWRITE 2 1 Disabled
d8 ff ff ff ff ff ff ff ........
08 00 00 00 00 00 00 00 ........
01 00 00 00 00 00 00 00 ........
00 02 0e 03 38 00 00 00 ....8...
08 41 d7 07 0c 00 00 00 .A......
10 68 10 3b f8 7f 00 00 .h.;....
20 40 f7 3a f8 7f 00 00  @.:.....
d0 d3 1c 3b f8 7f 00 00 ...;.... d8 ff ff ff ff ff ff ff 08 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 02 0e 03 38 00 00 00 08 41 d7 07 0c 00 00 00 10 68 10 3b f8 7f 00 00 20 40 f7 3a f8 7f 00 00 d0 d3 1c 3b f8 7f 00 00
4748 XtuService.exe 0x7ff494fc0000 0x7ff494fcffff VadS PAGE_EXECUTE_READWRITE 1 1 Disabled
00 00 00 00 00 00 00 00 ........
78 0d 00 00 00 00 00 00 x.......
0c 00 00 00 49 c7 c2 00 ....I...
00 00 00 48 b8 a0 49 58 ...H..IX
9a f8 7f 00 00 ff e0 49 .......I
c7 c2 01 00 00 00 48 b8 ......H.
a0 49 58 9a f8 7f 00 00 .IX.....
```

Fig. 3. Volatility Output Windows: Malfind

```
8852 PhoneExperienc 0x7ff81a170000 0x7ff81a18ffff Vad  PAGE_EXECUTE_READWRITE 0 0 Disabled
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........ 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8852 PhoneExperienc 0x7ff81a160000 0x7ff81a16ffff Vad  PAGE_EXECUTE_READWRITE 0 0 Disabled
ff 25 fa 0f 00 00 4c 8b .%....L.
15 fb 0f 00 00 ff 25 fd ......%.
0f 00 00 90 66 66 66 66 ....ffff
ff 25 fa 0f 00 00 4c 8b .%....L.
15 fb 0f 00 00 ff 25 fd ......%.
0f 00 00 90 66 66 66 66 ....ffff
ff 25 fa 0f 00 00 4c 8b .%....L.
15 fb 0f 00 00 ff 25 fd ......%. ff 25 fa 0f 00 00 4c 8b 15 fb 0f 00 00 ff 25 fd 0f 00 00 90 66 66 66 66 ff 25 fa 0f 00 00 4c 8b 15 fb 0f 00 00 ff 25 fd 0f 00 00 90 66 66 66 66 ff 25 fa 0f 00 00 4c 8b 15 fb 0f 00 00 ff 25 fd
8852 PhoneExperienc 0x7ff81a150000 0x7ff81a15ffff Vad  PAGE_EXECUTE_READWRITE 0 0 Disabled
10 00 0c 20 00 00 00 00 ...  ....
09 08 28 02 00 00 00 00 ..(.....
b0 93 70 19 f8 7f 00 00 ..p.....
00 40 70 19 f8 7f 00 00 .@p.....
a0 ff 13 1a f8 7f 00 00 ........
38 49 14 1a f8 7f 00 00 8I......
48 00 15 1a f8 7f 00 00 H.......
60 00 15 1a f8 7f 00 00 `....... 10 00 0c 20 00 00 00 00 09 08 28 02 00 00 00 00 b0 93 70 19 f8 7f 00 00 00 40 70 19 f8 7f 00 00 a0 ff 13 1a f8 7f 00 00 38 49 14 1a f8 7f 00 00 48 00 15 1a f8 7f 00 00 60 00 15 1a f8 7f 00 00
8852 PhoneExperienc 0x7ff81a190000 0x7ff81a19ffff Vad  PAGE_EXECUTE_READWRITE 0 0 Disabled
10 01 0c 20 00 00 00 00 ...  ....
01 00 45 01 01 00 00 00 ..E.....
b0 93 70 19 f8 7f 00 00 ..p.....
00 40 70 19 f8 7f 00 00 .@p.....
c0 ff 15 1a f8 7f 00 00 ........
58 16 18 1a f8 7f 00 00 X.......
50 00 19 1a f8 7f 00 00 P.......
00 00 00 00 00 00 00 00 ........ 10 01 0c 20 00 00 00 00 01 00 45 01 01 00 00 00 b0 93 70 19 f8 7f 00 00 00 40 70 19 f8 7f 00 00 c0 ff 15 1a f8 7f 00 00 58 16 18 1a f8 7f 00 00 50 00 19 1a f8 7f 00 00 00 00 00 00 00 00
8852 PhoneExperienc 0x7ff81a210000 0x7ff81a21ffff Vad  PAGE_EXECUTE_READWRITE 0 0 Disabled
80 00 00 00 00 00 00 00 ........
00 00 00 00 00 00 00 00 ........
00 03 08 18 18 1f 80 dd ........
10 18 00 00 00 00 00 00 ........
28 ae 1f 1a f8 7f 00 00 (.......
10 d2 ba 19 f8 7f 00 00 ........
70 f5 1e 1a f8 7f 00 00 p.......
```

Fig. 4. Volatility Output Linux: Malfind

TABLE I: Regular Testing: Windows

Defense	TP	TN	FP	FN	Accuracy
Symantec	987	0	1	12	98.7%
Kaspersky	986	0	0	14	98.6%
Avast	984	0	1	15	98.4%
McAfee	987	0	0	13	98.7%
ESET	985	0	1	14	98.5%
SPV	999	0	0	1	99.9%

TABLE II: Regular Testing: Linux

Defense	TP	TN	FP	FN	Accuracy
Kaspersky	987	0	1	12	98.7%
BitDefender	986	0	0	14	98.6%
Avast	984	0	1	15	98.4%
McAfee	987	0	0	13	98.7%
ESET	985	0	1	14	98.5%
SPV	999	0	0	1	99.9%

TABLE III: Regression Testing: Windows

Defense	TP	TN	FP	FN	Accuracy
Symatec	500	0	25	475	50.0%
Kaspersky	475	0	90	435	47.5%
Avast	485	0	75	440	48.5%
McAfee	495	0	105	400	49.5%
ESET	480	0	120	400	48.0%
SPV	999	0	0	1	99.9%

TABLE IV: Regression Testing: Linux

Defense	TP	TN	FP	FN	Accuracy
Kaspersky	500	0	25	475	50.0%
BitDefender	475	0	90	435	47.5%
Avast	485	0	75	440	48.5%
McAfee	495	0	105	400	49.5%
ESET	480	0	120	400	48.0%
SPV	999	0	0	1	99.9%

stealthiness. We utilized Testbed-2, Testbed-3, and Testbed-4 and executed the SPVExec; the image was saved as "X-Post-SPV," with X representing the OS. Each malware sample was executed, and a snapshot and memory collection were taken. The system was then reset with the "Post-SPV" images and infected with the next malware sample. As each memory dump was analyzed with Volatility with the plugins mentioned above, the persistence elements of the SPV were found without the markers of the malware surviving. This proves that the SPV Defense prevented the malware from taking effect and rendered it inert, on the same level as other security tools. Comparisons of our process to standard antivirus software indicated that our proposed approach achieves the same level of accuracy as other COTs antiviruses as shown in Tables I and II.

D. Experiment II-B: Reversion Testing

An additional image of Testbed-2, Testbed-3, and Testbed-4 were generated for this experiment, titled "X-SecurityReversion-TestingPost." The commercial antivirus software signature libraries were downgraded by three versions lower, allowing newer malware to be tested as though it were a zero-day exploit. The sample repository listed above was run on both virtual machines. Compared to standard antivirus detection rates, SPV Defense was able to maintain consistent rates. However, during the zero-day detection experiment, it doubled the detection rates of standard antivirus software, as shown in Tables III and IV. This further proves that SPV Defense can perform far better than commercial malware detection tools against unknown threats because it only targets the persistence vectors.

1) Experiment III: Deceptive Capability: For this experiment, the SPVExec was run against two unique phases. One phase determined if malware identified SPVs as similar malware, avoiding infections. The second is if legitimate programs like Antivirus saw the SPVs as a benign code structure. The system was reverted to a save of the SPV-defended state presented in Testbed-1 for defense through deception testing. The Necurs malware sample was run against the Windows system, and The SpeakUp [84] malware sample was executed on this Linux OS. These particular samples were chosen because of a built-in function searching for already modified keys signaling an infected system. A total of ten instances of the malware were executed in attempts to infect the system; each time, memory collections were completed. Upon analysis of the memory samples via the Volatility analysis, no signs of the Necurs malware were present. Benign testing was conducted using a pool of fifteen antiviruses against the SPV code base. All tests returned negative, indicating that none of the antiviruses flagged the SPVs as malicious.

2) Experiment IV: System Performance: In this experiment, we evaluate the effectiveness of our approach on system resources, particularly the impact of the SPV Defense process on memory and CPU utilization.

(i) CPU Utilization: Utilization was recorded in two separate instances to obtain a baseline for the pre- and post-deployment system. Baseline scores for each of these system performances were recorded. Next, multiple applications were opened to simulate a typical user's desktop, including two Microsoft Word documents (LibreOffice Word documents on the Linux platforms), a single instance of Google Chrome, and one instance of the Windows or Linux file structure, depending on the system. The system was then left under these conditions for 10 minutes. In the same way, as most effective rootkits perform malicious activities without overloading the system, SPVs run in the background without exhausting CPU resources. The CPU usage overhead is on par with that of average antivirus software, or an IDS/IPS, which is approximately 2 percent on average [33].

(ii) Memory Utilization The amount of memory the SPVs

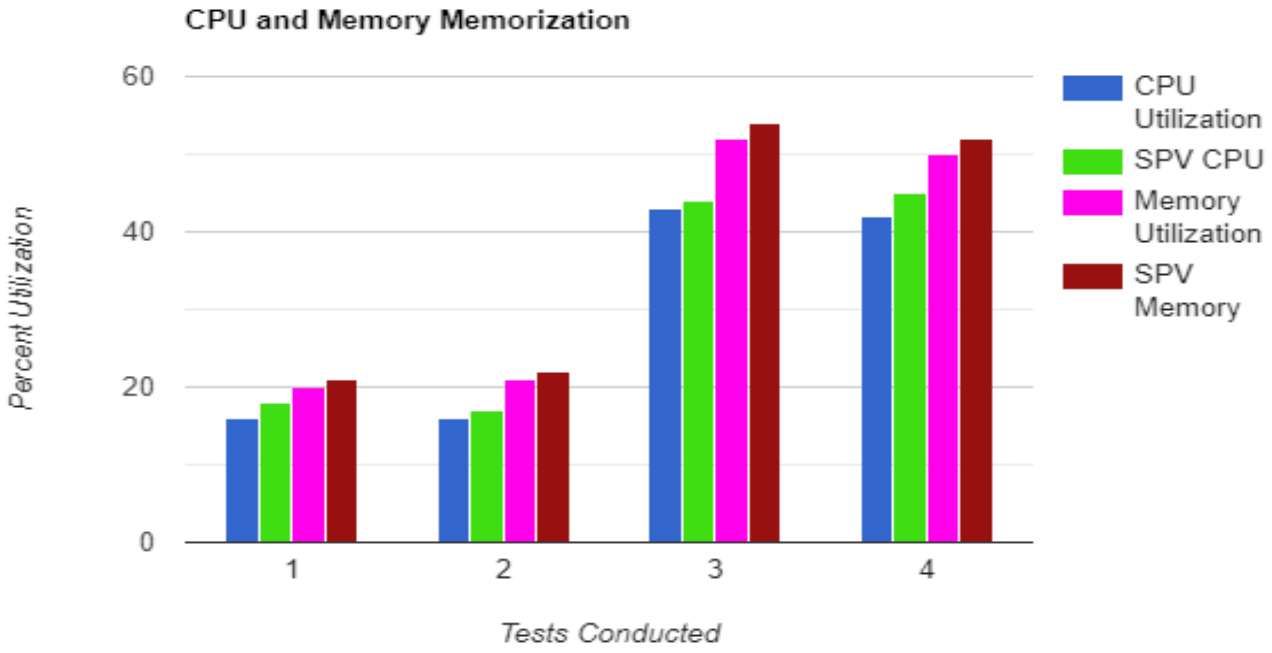

Fig. 5. Windows CPU and Memory System Comparison

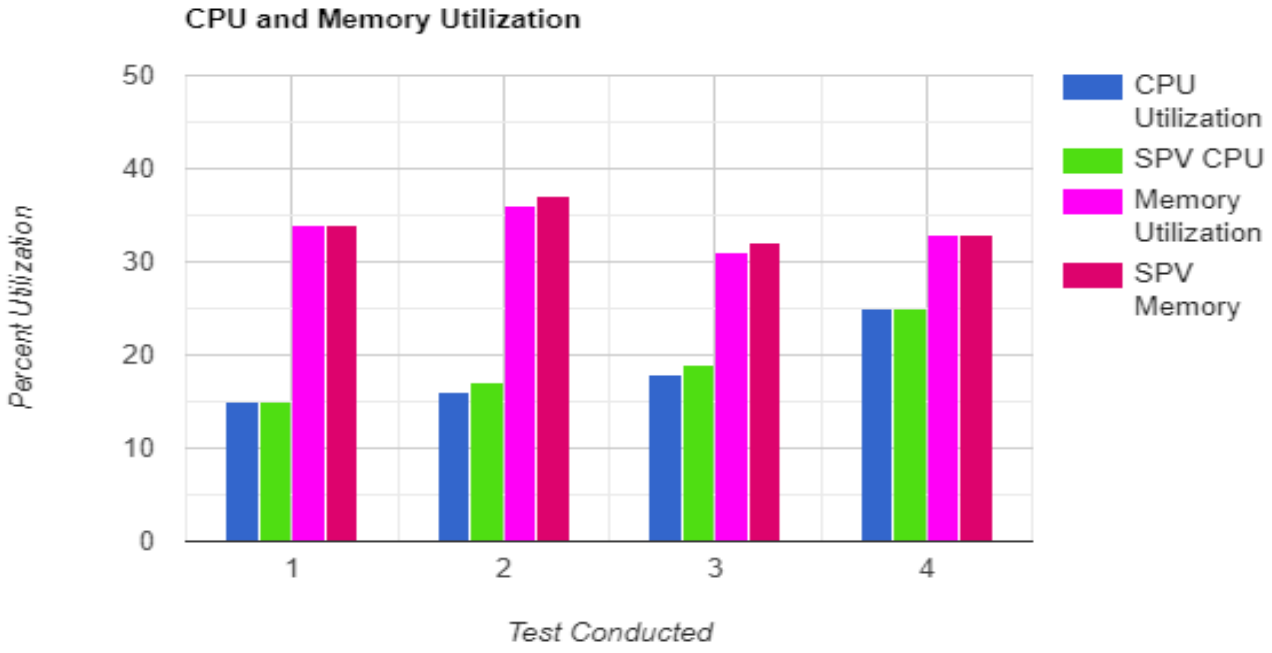

Fig. 6. Linux CPU and Memory System Comparison

utilize, specifically as they spawn processes, is also crucial. Too much memory utilization can cause an internal denial of service, making the method unusable. The baselines were again compared using the same parameters as in the CPU overhead test with the same software instances for the 10-minute implementation. This result also showed minimal impact on the system resources.

Figures 5 and 6 show a complete breakdown of these individual CPU and Memory utilization tests.

3) Experiment V: Whitelisting Capability: All the experiments conducted above proved the proposed method's ability to block future malware infections. However, this would be moot if regular benign programs could not make low-level system modifications and maintain their persistence. For this experiment, we attempted to install 10 "legitimate" programs on an SPV Defended system and determined that all were still installed after the system restart. These programs were PyCharm, Visual Studio, BitRise, Atom, BlueFish, CodePen, Crimson Editor, Eclipse, Komodo Edit, and NetBeans. The same methodology was leveraged to examine these software programs as malware to determine the system changes made to ensure their persistence. Individual snapshots from the "X-Post-SPV" series had one of the above ten programs installed. Memory collection was completed, and a snapshot was taken, titled "XPost-SPVTool," with X being the software installed. Upon powering on, a second memory collection was completed. Finally, the application was tested for functionality by launching the program. In all instances, both the SPV Defense and the program were operational and maintained persistence.

4) Experiment VI: Forensic Analysis of SPV Quarantine: Per the SPV code base, blocked malware becomes flagged and added to the equivalent of an antivirus "quarantine" zone. SPVs can collect attacks and convert information from these executables into regular defensive measures. To test this functionality, we attempted the infection with ten malware not utilized in creating the SPVs. Individually, each sample was executed against the SPV-loaded image. After each malware was launched, we took a forensics image of the test machine utilizing FTK Imager loaded on a separate drive. This process was repeated for each of the malware samples. Upon loading the evidence files into FTK Forensic Suite, all ten files were found inside the created quarantine zone.

VI. Limitations and Future Works

Our proposed SPV provides a more robust defense against malware than the existing research. However, the current implementation is limited to only core Linux and Windows OS. Additional work can be conducted into the persistence vectors that are different and unique to other OSes, which could prove beneficial. Other major operating systems, specifically Mobile OSs such as MacOS and Android, can benefit from the defense-by-deception strategy of SPVs. Thus, as part of future work, we plan to extend the current SPV to these platforms, along with improvements to the automation of the SPV generation. Additionally, research can be conducted into merging the SPVs into a universal executable, which is platform agnostic, and deploys SPVs based on an OS scan upon execution.

VII. Conclusion

This paper presents a new SPV Defense by Deception strategy that leverages sterilized persistence vectors extracted from a real malware corpus to block potential malware infections. Our system utilizes code from malware samples, not as signatures but as defensive strategies that stop new infections from attempting to write into persistence regions. Compared to existing COTs and techniques described in the literature for malware detection and prevention, our approach is designed to be more robust and versatile, with the ability to block malware both on bare hardware and in virtualized environments. Additionally, our methodology does not require a signature or agnostic of the target malware behavior. Through an in-depth evaluation of 2000 malware samples with pre- and post-SPV infection, we demonstrate that our proposed SPV Defense by Deception mechanism can effectively defend systems against malware infections with 1-3 percent CPU and memory overhead while not limiting the ability to install legitimate programs properly.

References

[1] N. Phillips and A. Ali Gombe, "Sterilized Persistence Vectors (SPVs): Defense Through Deception on Windows Systems," in Proc. CYBERWARE, 2022, pp. 56-61.

[2] I. Ahmed, A. Zoranic, S. Javaid, and G.G. Richard III, "Modchecker: Kernel module integrity checking in the cloud environment," In 2012 41st International Conference on Parallel Processing Workshops, Sep. 2012, pp. 306-313, IEEE.

[3] A. Ali-Gombe, I. Ahmed, G.G. Richard III, and V. Roussev, "AspectDroid: Android app analysis system," In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, pp. 145-147, 2016.

[4] A. Ali-Gombe, B. Saltaformaggio, D. Xu, and G.G. Richard III, "Toward a more dependable hybrid analysis of Android malware using aspect-oriented programming," Computers & Security, vol. 73, pp. 235-248, 2018.

[5] A. Ali-Gombe, I. Ahmed, G.G. Richard III, and V. Roussev, "Opseq: Android malware fingerprinting." In Proceedings of the 5th Program Protection and Reverse Engineering Workshop, Dec. 2015, pp. 1-12.

[6] Z. Gittins and M. Soltys, "Malware persistence mechanisms," Procedia Computer Science, vol. 176, pp. 88-97. Jan. 2020.

[7] M.U. Rana, M.A. Shaha, and O. Ellahi, "Malware Persistence and Obfuscation: An Analysis on Concealed Strategies," In 2021 26th International Conference on Automation and Computing (ICAC), Sep. 2021, pp. 1-6, IEEE.

[8] B.V. Prasanthi, "Cyber forensic tools: a review," International Journal of Engineering Trends and Technology (IJETT), vol. 41(5), pp. 266-271, 2016.

[9] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," In Proceedings of the 16th ACM conference on Computer and communications security, Nov. 2009, pp. 555-565.

[10] E. Chan, S. Venkataraman, F. David, A Chaugule, and R. Campbell, "Forenscope: A framework for live forensics," In Proceedings of the 26th Annual Computer Security Applications Conference, Dec. 2010, pp. 307-316.

[11] B.N. Flatley, "Rootkit Detection Using a Cross-View Clean Boot Method," AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH GRADUATE SCHOOL OF ENGINEERING AND MANAGEMENT, Mar. 2013.

[12] S.L. Garfinkel, "Automating disk forensic processing with SleuthKit, XML, and Python," In 2009 Fourth International IEEE Workshop on Systematic Approaches to Digital Forensic Engineering, May 2009, pp. 73-84, IEEE.

[13] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu, "Face-change: Application-driven dynamic kernel view switching in a virtual machine," In 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Jun. 2014, pp. 491-502, IEEE.

[14] D. Byers and N. Shahmehri, "A systematic evaluation of disk imaging in EnCase® 6.8 and LinEn 6.1," Digital Investigation 6.1-2, 2009, pp. 61-70.

[15] I.U. Haq, S. Chica, J. Caballero, and S. Jha, "Malware lineage in the wild," Computers & Security, vol. 78. pp. 347-363, Sep. 2018.

[16] O.S. Hofmann, A.M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with OSck," ACM SIGARCH Computer Architecture News, vol. 39(1), pp. 279-290, 2021.

[17] R. Hund, T. Holz, and F.C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," InUSENIX security symposium, Aug. 2009, pp. 383-398.

[18] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through VMM-based 'out-of-the-box'semantic view," In 14th ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, Nov. 2007, pp. 128-138.

[19] A. Kapoor and R. Mathur, "Predicting the future of stealth attacks," In Virus Bulletin Conference, Oct. 2011, pp. 1-9.

[20] J.D. Kornblum and C.F. ManTech, "Exploiting the rootkit paradox with Windows memory analysis," International Journal of Digital Evidence, vol. 5(1), pp. 1-5, 2006.

[21] T.K. Lengyel, S. Maresca, B.D. Payne, G.D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity, and stealth in the DRAKVUF dynamic malware analysis system," In Proceedings of the 30th annual computer security applications conference, Dec. 2014, pp. 386-395.

[22] L. Litty, H.A. Lagar-Cavilla, and D. Lie, "Hypervisor Support for Identifying Covertly Executing Binaries," In USENIX Security Symposium, Jul. 2008, vol. 22, p. 70.

[23] R. Luh, S. Schrittwieser, and S. Marschalek, "TAON: An ontology-based approach to mitigating targeted attacks," In Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services, Nov. 2016, pp. 303-312.

[24] D. Patten, The evolution to fileless malware, 2017.

[25] Malshare, www.malshare.com, Oct. 2019.

[26] N.L. Petroni Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," In Proceedings of the 14th ACM conference on Computer and communications security, Oct. 2007, pp. 103-115.

[27] F. Raynal, Y. Berthier, P. Biondi, and D. Kaminsky, "Honeypot forensics," In Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, Jun. 2004, pp. 22-29. IEEE.

[28] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," In International Workshop on Recent Advances in Intrusion Detection, Sep. 2008, pp. 1-20, Springer, Berlin, Heidelberg.

[29] J. Rutkowska, "System virginity verifier: Defining the roadmap for malware detection on Windows systems," In Hack in the Box security conference, Sep. 2005.

[30] M. Schmidt, L. Baumgartner, P. Graubner, D. Bock, and B. Freisleben, "Malware detection and kernel rootkit prevention in cloud computing environments," In 2011 19th International Euromicro Conference on Parallel, Distributed, and Network-Based Processing, Feb. 2011, pp. 603-610. IEEE.

[31] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, Oct. 2007, pp. 335-350.

[32] M.I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," In Proceedings of the 16th ACM conference on Computer and communications security, Nov. 2009, pp. 477-487.

[33] O. Sukwong, H. Kim, and J. Hoe, "Commercial antivirus software effectiveness: an empirical study," Computer, vol. 44(03), pp. 63-70, Mar. 2011.

[34] S. Vömel and H. Lenz, "Visualizing indicators of Rootkit infections in memory forensics," In 2013 Seventh International Conference on IT Security Incident Management and IT Forensics, Mar. 2013, pp. 122-139, IEEE.

[35] J. Wang, A. Stavrou, and A. Ghosh, "Hypercheck: A hardware-assisted integrity monitor," In International Workshop on Recent Advances in Intrusion Detection, Sep. 2010, pp. 158-177, Springer, Berlin, Heidelberg.

[36] Z. Wang, X. Jiang, W Cui, and X. Wang, "Countering persistent kernel rootkits through systematic hook discovery," In International Workshop on Recent Advances in Intrusion Detection, Sep. 2008, pp. 21-38, Springer, Berlin, Heidelberg.

[37] M. Xu, X. Jiang, R. Sandhu, and X. Zhang, "Towards a VMM-based usage control framework for OS kernel integrity protection," In Proceedings of the 12th ACM symposium on Access control models and technologies, Jun. 2007, pp. 71-80.

[38] Z. Xu, J. Zhang, G. Gu, and Z. Lin, "Autovac: Automatically extracting system resource constraints and generating vaccines for malware immunization," In 2013 IEEE 33rd International Conference on Distributed Computing Systems, Jul. 2013, pp. 112-123, IEEE.

[39] J. Rutkowska, "System virginity verifier: Defining the roadmap for malware detection on Windows systems," In Hack in the Box security conference, Sep. 2005.

[40] H. Yin, Z. Liang, and D. Song, "HookFinder: Identifying and understanding malware hooking behaviors," Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, Feb. 2008

[41] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," In Proceedings of the 14th ACM conference on Computer and communications security, Oct. 2007, pp. 116-127.

[42] H.A. Lagar-Cavilla and L. Litty, "Patagonix: Dynamically Neutralizing Malware with a Hypervisor," 2008.

[43] Y. Oyama, T.T. Giang, Y. Chubachi, T. Shinagawa, and K. Kato, "Detecting malware signatures in a thin hypervisor," In Proceedings of the 27th Annual ACM Symposium on Applied Computing, Mar. 2012, pp. 1807-1814.

[44] O. Vermaas, J. Simons, and R. Meijer, "Open computer forensic architecture a way to process terabytes of forensic disk images,"

In Open Source Software for Digital Forensics, 2010, pp. 45-67, Springer, Boston, MA.

[45] A. Mohanta and A. Saldanha, "Memory Forensics with Volatility," In Malware Analysis and Detection Engineering, 2020, pp. 433-476, Apress, Berkeley, CA.

[46] R. Tahir, "A study on malware and malware detection techniques," International Journal of Education and Management Engineering, vol. 8(2). p. 20, Mar. 2018.

[47] N. Idika and A.P. Mathur, "A survey of malware detection techniques," Purdue University, vol. 48(2). pp. 32-46, Feb. 2007.

[48] H. El Merabet and A. Hajraoui, "A survey of malware detection techniques based on machine learning," International Journal of Advanced Computer Science and Applications, vol. 10(1), 2019.

[49] K. Monnappa, "Automating Linux malware analysis using limon sandbox," Black Hat Europe, 2015, IV-A.

[50] M. Alrammal, M. Naveed, S. Sallam, and G. Tsaramirsis, "Malware analysis: Reverse engineering tools using santuko Linux," Materials Today: Proceedings, vol. 60, pp. 1367-1378, 2022.

[51] A. Ravi and V. Chaturvedi, "Static Malware Analysis using ELF features for Linux-based IoT devices," In 2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID), Feb. 2022, pp. 114-119, IEEE.

[52] D. Serpanos, P. Michalopoulos, G. Xenos, and V. Ieronymakis, "Sisyfos: A modular and extendable open malware analysis platform," Applied Sciences, Vol. 11(7), p. 2980, 2021.

[53] I.G. Kiachidis and D.A. Baltatzis, "Comparative Review of Malware Analysis Methodologies," arXiv preprint arXiv:2112.04006, 2021.

[54] J. Kim, Y. Ban, G. Jeon, Y.G. Kim, and H. Cho, "LiDAR: A Light-Weight Deep Learning-Based Malware Classifier for Edge Devices," Wireless Communications and Mobile Computing, 2022.

[55] C. Dietz, M. Antzek, G. Dreo, A. Sperotto, and A. Pras, "DMEF: Dynamic Malware Evaluation Framework," In NOMS 2022 - IEEE/IFIP Network Operations and Management Symposium, Apr. 2022, pp. 1-7, IEEE.

[56] S. Lee, H. Jeon, G. Park, J. Kim, and J.M. Youn, "IoT Malware Static and Dynamic Analysis System," Journal of Human-centric Science and Technology Innovation, Vol. 1(1), pp. 43-48, 2021.

[57] C. Hwang, J. Hwang, J. Kwak, and T. Lee, "Platform-independent malware analysis applicable to Windows and Linux environments," Electronics, vol. 9(5), p. 793, 2020.

[58] J.J. De Vicente Mohino, J. Bermejo-Higuera, J.R. Bermejo Higuera, J.A. Sicilia, M. Sánchez Rubio, and J.J. Martínez Herraiz, "MMALE a methodology for malware analysis in Linux environments," 2021.

[59] S.B. Mehdi, A.K. Tanwani, and M. Farooq, "Imad: in-execution malware analysis and detection," In Proceedings of the 11th Annual Conference on Genetic and evolutionary computation, pp. 1553-1560, 2009.

[60] J. Jeon, J.H. Park, and Y.S. Jeong, "Dynamic analysis for IoT malware detection with convolution neural network model," IEEE Access, vol. 8, pp. 96899-96911, 2020.

[61] E. Cozzi, "Binary Analysis for Linux and IoT Malware," Doctoral dissertation, Sorbonne Université, 2020.

[62] K.A. Asmitha and P. Vinod, "A machine learning approach for Linux malware detection," In 2014 International conference on issues and challenges in intelligent computing techniques (ICICT), 2014, pp. 825-830, IEEE.

[63] M. Kumar, "Scalable malware detection system using big data and distributed machine learning approach," Soft Computing, vol. 26(8), pp. 3987-4003, 2022.

[64] F. Shahzad, S. Bhatti, M. Shahzad, and M. Farooq, "In-execution malware detection using task structures of Linux processes," In 2011 IEEE International Conference on Communications (ICC), 2011, pp. 1-6. IEEE.

[65] I. Vurdelja, I. Blažić, D. Drašković, and B. Nikolić, "Detection of Linux Malware Using System Tracers-An Overview of Solutions," IcEtran, 2020.

[66] S.M.P. Dinakarrao, H. Sayadi, H.M. Makrani, C. Nowzari, S. Rafatirad, and H. Homayoun, "Lightweight node-level malware detection and network-level malware confinement in IOT networks," In 2019 Design, Automation and Test in Europe Conference and Exhibition, 2019, pp. 776-781, IEEE.

[67] T. Landman, and N. Nissim, "Deep-Hook: A trusted deep learning-based framework for unknown malware detection and classification in Linux cloud environments," Neural Networks, vol. 144, pp. 648-685, 2021.

[68] K.A. Asmitha, and P. Vinod, "Linux malware detection using eXtended–symmetric uncertainty," In International Conference on Security, Privacy, and Applied Cryptography Engineering, 2014, pp. 319-332, Springer, Cham.

[69] K.A. Asmitha and P. Vinod, "Linux malware detection using non-parametric statistical methods," In 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2014, pp. 356-361, IEEE.

[70] A. D. Raju, I. Y. Abualhaol, R.S. Giagone, Y. Zhou, and S. Huang, "A survey on cross-architectural IOT malware threat hunting," IEEE Access, vol. 9, pp. 91686-91709, 2021.

[71] Y. Xu, Z. Yin, Y. Hou, J. Liu, and Y. Jiang, "MIDAS: Safeguarding IoT Devices Against Malware via Real-Time Behavior Auditing," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 41(11), pp. 4373-4384, 2022.

[72] A. Mishra, A. Roy, and M.K. Hanawal, "Evading Malware Analysis Using Reverse Execution," In 2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS), 2022, pp. 1-6. IEEE.

[73] S. Das, H. Xiao, Y. Liu, and W. Zhang, "Online malware defense using attack behavior model," In 2016 IEEE International Symposium on Circuits and Systems (ISCAS), 2016, pp. 1322-1325. IEEE.

[74] F. Shahzad, M. Shahzad, and M. Farooq, "In-execution dynamic malware analysis and detection by mining information in process control blocks of Linux OS," Information Sciences, vol. 231, pp. 45-63, 2013.

[75] A. Kedrowitsch, D. Yao, G. Wang, and K. Cameron, "A first look: Using Linux containers for deceptive honeypots," In Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense, 2017, pp. 15-22.

[76] W. Sun, R. Sekar, Z. Liang, and V.N. Venkatakrishnan, "Expanding malware defense by securing software installations," In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 164-185, Springer, Berlin, Heidelberg, 2008.

[77] S.M.P. Dinakarrao et al., "Adversarial attack on microarchitectural events based malware detectors," In Proceedings of the 56th Annual Design Automation Conference, 2019, pp. 1-6.

[78] M. Zhang, A. Raghunathan, and N.K. Jha, "A defense framework against malware and vulnerability exploits," International Journal of information security, vol. 13(5), pp. 439-452, 2014.

[79] V. Chierzi and F. Mercês, "Evolution of IoT Linux Malware: A MITRE ATTandCK TTP Based Approach," In 2021 APWG

Symposium on Electronic Crime Research (eCrime), 2021, pp. 1-11, IEEE.

[80] J.M.C. Gómez, J.R. Gómez, J.L.M. Martínez, and A. del Amo Mínguez, "Forensic Analysis of the IoT Operating System Ubuntu Core," In Journal of Physics: Conference Series, vol. 2224, no. 1, p. 012082, IOP Publishing, 2022.

[81] Q.D. Ngo, H.T. Nguyen, V.H. Le, and D.H. Nguyen, "A survey of IoT malware and detection methods based on static features," ICT Express, vol. 6(4), pp. 280-286, 2020.

[82] H. Wang, W. Zhang, H. He, P. Liu, D.X. Luo, Y. Liu, and X. Lan, "An evolutionary study of IoT malware," IEEE Internet of Things Journal, vol. 8(20), pp. 15422-15440, 2021.

[83] C, R. (2022, July 27), "Linux malware trends 2022 H1." Infogram, Retrieved January 5, 2023, from https://infogram.com/linux-malware-trends-2022-h1-1ho16vowkmwm84n

[84] S. Popoveniuc, "Speakup: remote unsupervised voting," Industrial Track ACNS, 2010.

[85] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K.Z. Snow, F. Monrose, and M. Antonakakis, "The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle," In USENIX Security Symposium, 2021, pp. 3505-3522.

[86] A.D. Raju, I.Y. Abualhaol, R.S. Giagone, Y. Zhou, and S. Huang, "A survey on cross-architectural IOT malware threat hunting," IEEE Access, vol. 9, pp. 91686-91709, 2021.

[87] D. Patten, "The evolution to fileless malware," 2017.

[88] T. Steffens, "Attribution of Advanced Persistent Threats," Springer Berlin Heidelberg, pp. 153-164, 2020.

[89] L.E.S. Jaramillo, "Malware detection and mitigation techniques: Lessons learned from Mirai DDOS attack," Journal of Information Systems Engineering & Management, vol. 3(3), p. 19, 2018.