Unveiling BYOVD Threats: Malware's Use and Abuse of Kernel Drivers

Andrea Monzani*, Antonio Parata*, Andrea Oliveri†, Simone Aonzo†, Davide Balzarotti†, Andrea Lanzi*
*University of Milan, Italy
{andrea.monzani, antonio.parata, andrea.lanzi}@unimi.it
†EURECOM, France
{andrea.oliveri, simone.aonzo, davide.balzarotti}@eurecom.fr

***Abstract*—Bring Your Own Vulnerable Driver (BYOVD) attacks abuse legitimate, digitally signed Windows drivers that contain hidden flaws, allowing adversaries to slip into kernel space, disable security controls, and sustain stealthy campaigns ranging from ransomware to state-sponsored espionage. Because most public sandboxes inspect only user-mode activity, this kernel-level abuse typically flies under the radar. In this work, we first introduce the first dynamic taxonomy of BYOVD behavior. Synthesized from manual investigation of real-world incidents and fine-grained kernel-trace analysis, it maps every attack to sequential stages and enumerates the key APIs abused at each step. Then, we propose a virtualization-based sandbox that follows every step of a driver's execution path, from the originating user-mode request down to the lowest-level kernel instructions, without requiring driver re-signing or host mod-ifications. Finally, the sandbox automatically annotates every observed action with its corresponding taxonomy, producing a stage-by-stage report that highlights where and how a sample exhibits suspicious behavior. Tested against the current landscape of BYOVD techniques, we analyzed 8,779 malware samples that load 773 distinct signed drivers. It flagged suspicious behavior in 48 drivers, and subsequent manual verification led to the responsible disclosure of seven previously unknown vulnerable drivers to Microsoft, their vendors, and public threat-intelligence platforms. Our results demonstrate that deep, transparent tracing of kernel control flow can expose BYOVD abuse that eludes traditional analysis pipelines, enriching the community's knowledge of driver exploitation and enabling proactive hardening of Windows defenses.**

I. Introduction

Bring Your Own Vulnerable Driver (BYOVD) is a sophisticated malware attack technique that leverages legitimate but vulnerable kernel drivers to compromise computer systems. While drivers were originally designed to facilitate communication between the operating system (OS) and physical hardware, some drivers exist solely to implement system-level features or to expose privileged operations to user-mode applications. If these drivers contain vulnerabilities, malicious actors can exploit them to bypass security mechanisms, gain elevated privileges, and execute undetected malicious code in the OS kernel [1]. However, vulnerable drivers are routinely patched and replaced with new, more secure versions.

This is what makes a BYOVD scenario so dangerous and effective against modern Windows systems. In these attacks, malicious software brings along *its own copy* of a vulnerable third-party driver and loads it into the operating system – effectively reintroducing into the system an old vulnerability it already knows how to exploit.

In recent years, BYOVD attacks have been linked to state-sponsored cyber espionage campaigns and sophisticated ransomware operations [2], in which attackers use this technique to disable detection and ensure the success of their malicious activities [3]. Moreover, the fact that these attacks rely on trusted and signed drivers means that BYOVD attacks can often remain undetected for extended periods of time, allowing attackers to conduct prolonged, stealthy campaigns.

While BYOVD is becoming increasingly popular among malware authors, defense techniques are lagging behind. For instance, modern Endpoint Detection and Response (EDR) systems, although effective at monitoring user-mode activity, are often blind to actions that take place in the kernel. Even existing dynamic analysis systems and malware analysis sandboxes fall short in detecting these threats. For instance, when a malware sample loads and abuses the Zemana anti-malware driver to terminate Windows Defender, public sandboxes on VirusTotal detect only the driver load event, but fail to capture how it is used once loaded into the system. This limitation stems from a fundamental lack of visibility into internal kernel-level communications.

While a significant body of research has been focused on detecting malicious activity within the kernel [4], [5], [6], [7]–especially in the context of rootkits and driver-based threats–existing approaches remain limited in their ability to reconstruct the kernel drivers' control flows. Hardware-assisted runtime monitoring approaches [8] primarily focused on kernel memory integrity, but lacked insight into behavioral context or user-mode interactions. *HookScout* [9], instead, introduced a combination of static and dynamic analysis to detect kernel rootkits via anomalous control flow, yet struggled with obfuscation and dynamic resolution–techniques now common in modern BYOVD attacks. Other dynamic systems have applied taint tracking and control-flow tracing to detect kernel interface

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231491
www.ndss-symposium.org

abuse [10], [11], but typically lack the semantic depth needed to attribute actions to specific user-mode initiators.

The complexity of tracking drivers' control-flow arises from the fact that these interactions span multiple abstraction layers within the Windows operating system. A single command in user-space can trigger a cascade of driver dispatch routines, indirect function calls, and deferred callbacks deep within the kernel. Drivers may register multiple entry points–ranging from standard `IRP` handlers to minifilter message ports or even custom callback routines for system events–all of which attackers can leverage to achieve malicious goals. Accurately monitoring and attributing these behaviors requires end-to-end visibility: from the moment a user-space process initiates communication with a driver, through the internal execution paths taken within the kernel, and all the way until the final effect on the system.

In addition, the dynamic nature of driver execution, which often involves pointer indirection, runtime memory allocation, and interactions with poorly documented kernel APIs, makes the task even more complex. Furthermore, there is a need to correlate these kernel events back to the original user-mode trigger, which may be separated by layers of abstraction or deferred execution. In essence, the challenge of studying BYOVD attacks is not just the ability to capture isolated function calls in the kernel, but also reconstructing the complete behavioral chain that links a user-mode command with a sequence of low-level kernel actions. This requires fine-grained tracing, semantic interpretation, and robust techniques to resolve ambiguous execution flows, making it a fundamentally hard problem for dynamic analysis platforms.

To address these limitations, our paper brings three key contributions to the area of malware analysis.

First, we propose a taxonomy of observed BYOVD behaviors, capturing the common stages and APIs used in such attacks. It fuses manual investigation of real-world cases with dynamic kernel-trace analysis, moving beyond purely static enumerations to give method-level insight into how malware abuses drivers. By distilling attacks into five sequential stages–driver dropping, loading, communication, execution complexity, and observable suspicious behaviors–it supplies a "kill chain" model that guides defenders in detection and response.

Second, we develop a novel virtualization-based sandbox, based on DRAKVUF [12], that enables deep inspection of kernel-mode activities triggered by user-space programs. Our sandbox–in the spirit of open science, released as open-source [13]–functions as an advanced offline analyzer: an analyst feeds a suspected BYOVD sample into the virtualized environment, where the system runs it and intercepts kernel-mode API calls, driver dispatch routines, and event chains. Finally, it produces a report of the sample's behavior with respect to our taxonomy. However, rather than relying solely on a fixed, empirically derived rule set, the sandbox captures all interactions, including those not yet covered in our taxonomy so that when an unfamiliar sequence arises, even samples with novel or atypical BYOVD tactics are fully exposed, enabling researchers to enrich the rule database and guarantee broader, more accurate malware inspection over time. To the best of our knowledge, no existing sandbox in the literature is capable of correlating kernel-level traces to capture BYOVD behavior.

Third, to test and prove the validity of our system, we examined 8,779 samples containing both recognized and potentially vulnerable driver datasets. Of the 773 drivers in total, our system identified BYOVD activities in approximately 44% of the highly suspicious driver groups and discovered unusual operations in 48 of the newly revealed drivers. We manually validated seven of these drivers and the malware sample that abuses them. We then responsibly disclosed this information to the interested parties. These results highlight the practical impact and real-world effectiveness of our taxonomy-driven approach integrated into our sandbox.

II. Background

Starting from Windows Vista 64-bit, Microsoft introduced Driver Signature Enforcement (DSE) [14], [15] to ensure that each kernel driver loaded in the system is signed. To sign a driver, developers submit the binary to Microsoft's Windows Hardware Dev Center, where it undergoes a suite of security and compatibility checks before Microsoft applies its digital signature [14]. While there have been cases of malicious drivers signed by Microsoft [16], this is a risky approach for malware developers. Instead, attackers often exploit legitimate signed drivers with security flaws that expose kernel functions to user-mode programs. In particular, malware authors targets drivers that allow arbitrary memory access or dangerous actions like process termination, kernel memory mapping, privileged file access, and I/O monitoring.

If not adequately protected [17], these functionalities can be exploited by an attacker to subvert the system's security, e.g., to kill a protected process, load an unsigned driver, overwrite protected files, or to extract security artifacts from memory. Although Microsoft provides guidelines to minimize unsafe development practices [18], vulnerable drivers continue to be discovered on a regular basis.

A common way to trigger a vulnerability in a driver is to send a properly crafted request. Windows supports two different types of communications with kernel drivers: Minifilter APIs and I/O Request Packet (IRP). *Minifilter* drivers are kernel-mode modules that plug into the Windows Filter Manager (`FltMgr.sys`) to inspect, modify, or block, for example, file-system I/O through pre- and post-operation callbacks. Communication between user mode and a Minifilter driver uses a Filter Communication Port, a secure, bidirectional channel provided by the Windows Filter Manager. An IRP (I/O Request Packet) instead is a data structure used by Windows to facilitate communication between user-mode applications and generic kernel-mode drivers. Within an IRP, a specific command known as an IOCTL (I/O Control Code) instructs the driver to perform a particular operation. In this context, the IRP serves as the delivery mechanism, while the IOCTL represents the specific instruction it carries. If a driver fails to properly validate the input of an IRP, attackers may exploit this weakness by crafting malicious IOCTLs, potentially triggering vulnerabilities in the driver's handler routines. For instance, a Process ID (PID) embedded into an IOCTL buffer can be used to obtain a privileged handle to a process with the function `ZwOpenProcess` and then to terminate it through a call to `ZwTerminateProcess`. If the PID is linked to an anti-malware protected service, it can be killed by an unprivileged application just by abusing a vulnerable driver.

Once a driver with the desired characteristics is identified, the malware must drop it on disk and then load it after successfully infecting the system. This procedure is due to the fact that typical methods used to load drivers (e.g., `sc.exe`, `NtLoadDriver`, or `CreateService`) require a file path pointing to a location on disk and cannot be performed from memory. However, loading a driver in Windows requires the process to already run with administrative privileges. This raises a crucial question: why would malware bother loading a driver if it already has administrative privileges? The answer lies in the design of modern Windows operating systems. Some processes and system components are so essential to maintaining the security and integrity of the system that even administrators are prohibited from modifying them. Only the kernel has the necessary privileges to modify these highly protected system parts. In fact, the two most notable examples of the usage of the BYOVD technique to tamper with highly protected system components are the termination of protected processes and the loading of unsigned kernel drivers.

Protected Processes Termination. *Protected Processes Light* (PPL), introduced in Windows 8.1, provides a flexible protection model designed to isolate processes. They are extensively used to isolate different operating system components, such as process management, the Local Security Authority service [19], and Microsoft Defender. In addition, modern EDRs are implemented by using a combination of user-mode PPL processes (that contain the detection logic), and kernel-mode drivers (that provide access to privileged resources). Nowadays, malware can no longer terminate user-space EDR components (even when running with administrative privileges), thanks to their protection under the PPL mechanism. In these cases, an attacker could find a way to limit the EDR functionalities through other kinds of vulnerabilities [20], [21], [22], exploit flaws in the PPL implementation [23], or perform a BYOVD attack to abuse an existing driver that allows a user-mode process to kill even PPL processes.

Dynamic Code Execution in Kernel Another common reason for abusing vulnerable drivers is the exploitation of vulnerabilities that allow a user-mode process to execute code in kernel.

An example of this vulnerability can be found in the `capcom .sys` driver [24]. This driver was created by the video game producer company CAPCOM to fight game cheaters and contains a vulnerability that allows an attacker to execute code at kernel level. The driver communication is performed by sending an IOCTL packet that permits to disable the Supervisor Mode Execution Prevention (SMEP), executes the code residing at an address specified in the IOCTL packet by the user mode process, and enables again SMEP. With this code execution capability, a shellcode could be crafted to obtain full kernel code execution.

III. Taxonomy

To systematically characterize the BYOVD threat, we devised a bottom-up taxonomy that incorporates dynamic analysis by capturing kernel traces generated in our sandbox environment (later described in Section 1). We began with a manual review of publicly documented BYOVD cases, drawing on reports from security firms and independent researchers that detail how modern malware abuses this technique [2], and extended our work beyond prior efforts. Unlike a recent CheckPoint blog post [25], which statically enumerates API calls that can be potentially abused, our taxonomy enriches each category with method-level trace insights and explicates the behavioral phases of an attack with dynamic information. Central to our framework is the validation of the APIs list originally identified in CheckPoint's blogpost. We refer to these as *Abusable Imports*, and encompass common features exploited by known BYOVD malware. The complete list appears in Table VI in the Appendix.

We then created a YARA rule [26], that covered each entry of the list, and we executed it in the VirusTotal Retrohunt service. This allowed us to automatically retrieve many 64-bit signed drivers, which compose our dataset of *Potentially Vulnerable Drivers* (PVD) described later in Section V. The dataset is designed to uncover drivers that, although not previously reported as vulnerable, may exhibit characteristics commonly associated with BYOVD exploitation.

We then identified **five common stages** that all BYOVD attacks have in common and that represent the essential steps a sample must take to exploit a vulnerable driver and trigger behavior that compromises system security. Section IV details the analysis system we built to dynamically execute samples and classify their behavior according to these stages:

I) Driver dropping. A sample can utilize a driver already present in the system or drop (e.g., by decrypting, decompressing, or downloading) a new file, which will later be loaded as kernel components.

II) Driver loading. Drivers are typically loaded by creating a Windows service. However, in some cases, a malware sample may load an unsigned driver by exploiting a vulnerability in a previously loaded module.

III) Driver communication. Communication with drivers is primarily carried out by sending I/O Request Packets to the device they expose. In some cases, if the driver is a Minifilter driver, communication can occur through Minifilter APIs.

IV) Execution. Triggering suspicious behaviors may require one or multiple interactions (IRP or Minifilter). This aspect helps determine whether execution is simple (one) or complex (multiple).

V) Observable Suspicious Behaviors. We have identified a list of suspicious behaviors through an iterative process detailed in more detail in Section IV. This process was guided by manual analysis of vulnerable drivers, to inform function hooking and behavior correlation, and further supported by examining CVEs associated with Windows drivers. The ordered sequence in which these behavior groups appear during the execution of the sample forms a *chain* of suspicious activity. We discuss these chains, with a detailed discussion on our detection approach, in IV-C1.

IV. Approach

As previously discussed, traditional malware analysis solutions focus on monitoring user-space components and thus fail to capture and identify BYOVD attacks. To address this limitation, we introduce a virtualization-based analysis sandbox specifically designed to detect suspicious driver interactions originating from user-mode processes while ensuring complete kernel-level monitoring of operating system actions. In particular, our framework offers (1) a systematic detection of BYOVD exploitation techniques, (2) an extensive kernel-level observation to capture subtle and elusive suspicious behaviors, and (3) automated analysis pipelines to pinpoint suspicious behavior indicative of driver abuse or exploitation.

A. System Architecture & Requirements

Our system produces a detailed behavioral report that highlights whether a vulnerable driver has been exploited by a malware sample according to our taxonomy. In line with standard sandbox-based malware analysis, we concentrate on capturing and presenting runtime behaviors, leaving detection and classification to downstream tools or human analysts.
To this end, our analysis system must satisfy three key requirements:

- **(R1)** Distinguish between safe kernel-driver communication and potential misuse by accurately identifying and extracting behaviors indicative of abuse.
- **(R2)** Carefully trace kernel execution to ensure no critical function calls are overlooked during event handling.
- **(R3)** Enrich each function call with sufficient dynamic information about its operands, to support effective analysis.

The analysis outputs a classification of the sample behaviors inside our taxonomy and a trace of relevant functions that can pinpoint a vulnerability or an abused functionality.

Our proposed system architecture comprises several interdependent components. The foundation is our virtualized sandbox, built on top of the Xen hypervisor technology [27], that provides an isolated, secure, and transparent execution environment that minimizes interference and allows precise monitoring of kernel activities. The sandbox management is provided by Drakvuf [28], [12], an advanced sandboxing framework that facilitates the systematic execution of malware samples. Importantly, no instrumentation is inserted into the guest operating system itself. Thanks to Drakvuf's hypervisor-based Virtual Machine Introspection (VMI) capabilities, there is no need to install kernel modules or other intrusive components within the guest. Instead, only a minimal user-mode application is required, solely to deploy the malware sample, prevent its memory from being paged out, and initiate its execution within the virtual machine.

As depicted in Figure 1, the core of our architecture is represented by the VM introspection capabilities provided by our custom-developed Drakvuf plugin, *kernelmon*. *Kernelmon* provides detailed introspection and logging capabilities of kernel-mode operations by placing breakpoints in crucial kernel functions. These functions include driver loading and unloading routines, critical memory allocation APIs, IOCTL packet handling, and kernel callback routines. The real-time monitoring capabilities of *kernelmon* enable it to capture complete parameter information, execution context, memory operations, and detailed interactions between kernel modules and user-mode processes, thus creating a rich and informative dataset essential for accurate analysis.

Execution traces are recorded in JSON format, which is stored and indexed in a non-relational database to provide efficient retrieval and facilitate our post-execution analysis pipeline. After execution, the logged data is analyzed to enrich kernel traces with contextual information (e.g., mapping pointer addresses to function symbols). We also developed custom algorithms to reconstruct data flows, interaction sequences, and kernel events, enhancing analysis accuracy by capturing indirect or context-sensitive interactions not evident in the raw data.

For the interested reader, we provide a formalization of the analysis pipeline in Appendix A-A.

B. Threat Model

The threat model of our sandbox assumes that the sample runs with administrative privileges (we do not focus on specific user-mode privilege escalation techniques) sufficient to load a valid driver from disk because it is a mandatory prerequisite for BYOVD attacks. The guest Windows machine uses the default Windows Defender antivirus, configured to exclude all files and processes from scanning. This prevents Defender from blocking known malicious samples while maintaining a realistic AV deployment scenario. Moreover, given that BYOVD attacks often target specific AV products, we simulate them by spawning idle processes named after popular AV vendors and protecting them with `kernelmon` immediately before executing the sample. Crucially, evading our behavioral tracing is inherently difficult because the sandbox operates at the hypervisor layer, so we reasonably assume that code executing inside the VM cannot tamper with or bypass it. Finally, to defeat attempts to obscure malicious actions with interleaved benign API calls, our system employs data-flow analysis to reconstruct semantic call chains, allowing us to pinpoint genuine exploit behaviors and discard irrelevant noise.

C. Tracing Kernel Driver Behaviors

The *kernelmon* plugin provides fine-grained, real-time introspection into kernel-mode operations thanks to a number of hooks. We verified that all critical driver execution activities were captured by systematically comparing logged kernel events with ground-truth data from known drivers. We also

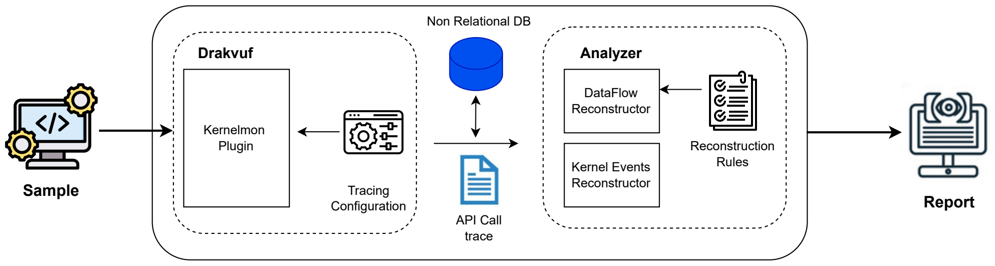

Figure 1: Architecture of the Analysis System

relied on the Windows Internals books [29], [30] and official kernel documentation to guide and validate our selection of hooking points and their coverage of relevant driver behaviors.

1) Observable Suspicious Behavior: These hooks are designed to capture specific high-impact security-related behaviors that represent the ultimate goals of BYOVD attacks. Each instrumentation point targets a suspicious action, such as memory tampering, protected process termination, and dynamic code execution within the kernel. To determine which hooks to implement, we ran malicious examples from different attack classes and analyzed the functions involved in each case. By observing these functions and their parameters, we could identify the specific behaviors tied to each malware goal (R1). This empirical approach allowed us to define the types of hooks that need to be monitored in order to capture eight critical behaviors associated with BYOVD attacks effectively.

[B1] Protected Service Termination Detection – *Kernelmon* detects unauthorized terminations of protected processes by setting hypervisor-based breakpoints on `ZwTerminateProcess` and monitoring the sequence of events initiated by user-mode drivers. Furthermore, it traces the origin of these terminations by observing handle acquisition APIs, such as `ZwOpenProcess` or `ObReferenceObjectByHandle`. It also reconstructs the full kill chain from handle access to termination, thus enabling a precise attribution of such activities to user-mode interactions.

[B2] Privileged User Handle Leakage Monitoring – To detect privileged handles opened by a kernel driver and used by user mode components, *kernelmon* hooks functions with prefixes like `ZwCreate` and `ZwOpen`, focusing on invocations where the `OBJ_KERNEL_HANDLE` flag is absent. The omission of this flag can unintentionally or maliciously grant user-mode applications unauthorized access to otherwise inaccessible resources. *Kernelmon* inspects the structure passed to these kernel calls and logs all improperly created handles.

[B3] Detection of Unsafe Pool Allocations – Drivers often allocate non-pageable kernel memory that is both writable and executable (`WX`), thus violating modern security practices such as the W⊕X policy. *Kernelmon* monitors allocations performed via `ExAllocatePoolWithTag` and checks the memory protection flags and target pool types. If a driver allocates memory with both `PAGE_EXECUTE_READWRITE` and `NonPagedPool` characteristics, *kernelmon* flags the allocation and records the responsible driver context.

[B4] Dynamic Kernel Code Execution Tracking – *Kernelmon* sets hypervisor-level execute breakpoints on newly allocated `WX` memory regions. This allows it to capture actual execution attempts within dynamically generated code segments. By correlating execution attempts with previous memory allocations, *kernelmon* can attribute runtime code generation to specific drivers or IOCTLs, thereby identifying potential execution of unsigned drivers' code in the kernel.

[B5] Mismatched Memory Mapping Detection – Drivers sometimes remap physical memory or locked pages with elevated permissions, such as mapping non-executable regions as executable or bypassing cache protections. *Kernelmon* intercepts remapping operations through functions like `MmMapIoSpace` and `MmMapLockedPagesSpecifyCache`, comparing the requested protection level with the attributes of the original memory region. Discrepancies are flagged and logged, providing evidence of possible unsafe memory modifications.

[B6] Real-Time Code Integrity Tampering Detection – To detect manipulation of kernel-level integrity checks, *kernelmon* places write-monitoring breakpoints on critical kernel structures involved in Windows Code Integrity enforcement, such as the `g_CiOptions` global variable [31] or callback tables[32]. When modifications to these structures are detected, *kernelmon* logs the offending instruction pointer and stack context. This captures attempts to disable driver signature enforcement or patch code integrity in memory—key steps for unsigned driver loading or stealthy rootkit deployment.

[B7] Detection of Remote Handle Closure – *Kernelmon* monitors kernel calls involving cross-process handle manipulations, specifically tracking scenarios where drivers leverage `KeStackAttachProcess` to gain remote execution contexts followed by handle closure via `ObCloseHandle`. This is commonly observed in attacks targeting security or monitoring processes, enabling the detection of stealthy handle closures intended to disrupt critical system operations.

[B8] Detection of Arbitrary Memory Reads and Writes in Kernel Space – Although these general vulnerabilities exist and are actively exploited by attackers, the detection of arbitrary read/write operations goes beyond the purpose of this paper. In fact, to precisely identify their presence and determine their actual impact on the system, a dynamic analysis system would need to monitor every single memory access that occurs in kernel space. This would impose a fundamental obstacle to the scalability of the system, and thus we decided to exclude the monitoring of this generic behavior from our dynamic approach. This is also justified by the fact that other approaches, typically based on static analysis, already exist to detect the presence of arbitrary read/write vulnerabilities in kernel modules [33], [34], [35]. These tools demonstrate the capability of static analysis to effectively uncover such issues without requiring exhaustive dynamic monitoring.

2) Comprehensive Driver Execution Tracing: To obtain complete visibility over the driver execution (requirement R2), *Kernelmon* implements a suite of strategically chosen hooks that cover the full operational lifecycle of kernel drivers. These hooks, grouped into five categories (H1–H5), are designed to capture control transfers into and out of driver code while preserving context for reconstructing behavior patterns and verifying sample provenance.

[H1] Driver Lifecycle Event Monitoring – *Kernelmon* hooks the `MmLoadSystemImageEx` and `MmUnloadSystemImage` kernel functions to intercept driver loading and unloading events. These hooks allow the system to detect when a new driver binary is mapped into memory and later removed, recording the base address, size, and associated module metadata. This traceability enables accurate attribution of behaviors to specific drivers and provides temporal anchoring for behavioral reconstruction. This monitoring is essential to discriminate whether suspicious actions originate from recently introduced drivers, and supports early termination of analysis in the absence of new driver activity (as defined in the analysis pipeline).

[H2] Driver Communication Entry Point Tracing – Understanding how a driver interfaces with user-mode processes is critical for vulnerability attribution. *Kernelmon* intercepts the execution of `DriverEntry`, the canonical initialization routine, and subsequently extracts the dispatch table embedded in the `DRIVER_OBJECT` structure. It specifically identifies handlers for `IRP_MJ_CREATE` and `IRP_MJ_DEVICE_CONTROL`, which govern device handle creation and IOCTL processing, respectively. Additionally, the system supports minifilter communication ports (through `FltCreateCommunicationPort`), further enhancing driver-to-user communication tracking. The plugin then sets breakpoints on these routines, enabling fine-grained tracing of user-to-kernel transitions and input-driven behaviors. This capability is particularly useful for identifying driver communication patterns, such as the triggering of vulnerable paths via IOCTL messages. These hooks are directly linked to both R2 and R3, providing operand-level data (e.g., IOCTL codes and buffer contents) used to classify communication as benign or exploitative.

[H3] Kernel Callback Routine Analysis – Many drivers register asynchronous callbacks to respond to kernel-level events. Examples include `PsSetCreateProcessNotifyRoutine`, `PsSetLoadImageNotifyRoutine`, and `ObRegisterCallbacks`. *Kernelmon* intercepts these registration calls and dynamically instruments the associated callback routines by resolving the function pointer arguments at runtime. Inserting breakpoints at these addresses captures deferred or event-driven logic. Callback analysis is instrumental in linking driver activity to high-level system events and complements control-flow tracing. It directly supports R2 and R3 by enriching the event log with context-aware, indirect execution paths.

[H4] Selective Tracing via Process Inclusion/Exclusion – To manage the trade-off between coverage and overhead, *Kernelmon* supports a configurable policy for process inclusion or exclusion based on executable names or full paths. This filtering mechanism ensures that only relevant user-mode processes (e.g., the sample under test) are monitored for driver interactions. Importantly, this capability helps maintain high performance in virtualized environments without sacrificing the completeness of critical trace data, aligning with the need for scalable analysis as required by R2. One potential concern is that malicious software could impersonate an excluded process to evade monitoring. To mitigate this risk, we rely on the kernel-level process path to filter out only a small set of processes, such as antivirus components or essential system background services. Commonly exploited processes like the Windows shell (`explorer.exe`) are not excluded by default. This design choice ensures that exclusion policies cannot be easily bypassed by malware imitating legitimate processes. To circumvent this countermeasure, malware would need to either replace a system binary, likely invalidating its digital signature, or perform process injection, both of which are known and detectable techniques that fall outside the scope of this paper. The exclusion list can be updated dynamically between runs if more relaxed filtering is needed.

[H5] Full Driver Import Resolution and Execution Tracing – *Kernelmon* is capable of tracing all functions imported by a given driver binary. This includes resolving static imports from `ntoskrnl.exe` and other modules by obtaining function addresses in the Import Address Table (IAT) or via dynamic resolution through `MmGetSystemRoutineAddress`. By instrumenting these function calls, the system reconstructs complete control flow within the driver, capturing transitions across both statically linked and dynamically resolved APIs. This is essential for detecting non-observable internal routines and indirect attack sequences. Crucially, it ensures that no logic path escapes analysis, fulfills R2's completeness requirement, and supports the semantic call-chain construction of R3.

D. Data Processing and Analysis

During the execution of each sample, *kernelmon* collects and logs all telemetry related to kernel driver execution. After the execution is completed, our system initiates a multistage analysis pipeline to classify the behavior of the sample and extract security-relevant insights. The goal of this process is to reconstruct semantically meaningful activity chains from raw kernel events and identify patterns that align with our behavioral taxonomy.

The first step involves loading the module extraction, where we query the trace to determine whether any new kernel drivers were mapped to memory during execution. If no such driver activity is detected, the analysis is aborted, as our focus is specifically on detecting abuses originating from possibly vulnerable or malicious drivers.

Next, we perform driver attribution and lifecycle reconstruction, which includes identifying driver drop events, locating driver load operations, and tracking invocations of key driver routines such as `DriverEntry`, `IoCreateDevice`, `IoCreateSymbolicLink`, and `DeviceControl`. By associating these events with file system activity and process identifiers, we reconstruct how and when the sample introduced and engaged each driver.

The core of the analysis revolves around the detection of observable behaviors as defined in our taxonomy. For instance, it searches for specific syscall patterns and kernel API usage (e.g., `ZwTerminateProcess`, `MmMapIoSpace`, or `ExAllocatePoolWithTag`) to identify actions indicative of process tampering, code injection, memory manipulation, or privilege escalation. Each suspicious event is correlated with the driver and user-mode process that triggered it. This detection step is rule-based and relies on a set of predefined sequences of kernel API invocations and parameter patterns indicative of suspicious behavior. These rules are derived both from the plugin's direct event tagging (e.g., dynamic code execution or integrity tampering) and from the contextual inspection of kernel call arguments, such as protected process IDs in termination requests.

To increase the precision of detection, our system also incorporates a lightweight data-flow tracking mechanism. For each suspicious action, the system recursively traces the propagation of input values, such as process handles or memory addresses, across earlier function calls. This reverse data-flow analysis is configured through a function-to-parameter mapping file and results in a contextualized call chain (or behavioral frame) that offers a high-level semantic view of the attack sequence (R3).

The final output of the analysis stage includes:

- A classification of the driver-related activities of the sample (e.g., driver dropping, loaded via service, IOCTL communication).
- A timeline of triggered behaviors, along with their associated driver modules and process origins.
- A condensed behavioral signature composed of call frames that lead to each detected malicious action.

V. Dataset Design and Methodology

To effectively analyze the BYOVD phenomenon, it is essential to consider kernel drivers that exhibit exploitable behaviors. To this end, we construct and rely on two distinct datasets, each curated explicitly for this study.

The first dataset, called the KVD *(Known Vulnerable Drivers)* dataset, includes drivers already recognized by the security community as vulnerable. It serves as a control set to validate our sandbox's functionality, assess BYOVD detection performance, and characterize executables involved in BYOVD attacks.

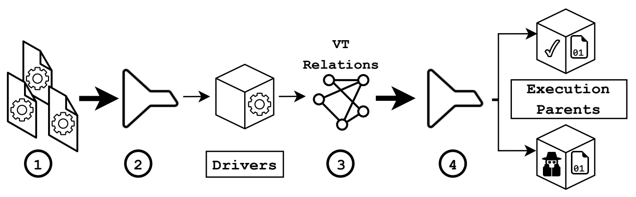

Figure 2: Datasets Collection and Filtering Pipeline.

The second dataset, designated as the PVD *(Potentially Vulnerable Drivers)* dataset, is composed of drivers collected from in-the-wild sources, for which no prior vulnerability information is available. We use the PVD dataset to evaluate the sandbox's ability to perform dynamic detection and to discover previously undocumented instances of BYOVD exploitation.

Finally, in order to thoroughly study BYOVD attacks, we also need the user-space executables that deploy these drivers and potentially leverage their weaknesses for malicious purposes. We therefore included in both datasets also their corresponding *execution parents*, following VirusTotal's terminology, which consists of those executables that have been observed to write the driver to disk.

A. Dataset Construction Pipeline

As illustrated in Figure 2, our pipeline begins at Step ①, where we gather driver samples linked to executable binaries observed in the wild for each dataset (see the next subsection for details). In Step ②, we remove drivers that are unsupported on modern Windows systems, i.e., 32-bit drivers, not digitally signed, or those with an invalid signature. In Step ③ we use VirusTotal's *Relations* feature to retrieve all the execution parents that wrote one of the considered drivers to disk during dynamic analysis. This is a necessary step to perform a BYOVD attack, since Windows only supports loading drivers from storage and not from memory. Finally, in Step ④, we cluster the execution parents by using VirusTotal `vhash` [36], a fuzzy hashing algorithm based on files' structure, and retain only a single sample from each cluster. This helps to reduce redundancy while preserving behavioral diversity.

Finally, we classify all execution parents based on their VirusTotal (VT) score, which represents the number of antivirus engines that flag a given binary as malicious. Following thresholds proposed in prior work [37], we separate the execution parents into two distinct subsets:

- Samples with a `VT score` < 6 are considered potentially benign. However, this set may also include stealthy BYOVD loaders that avoid widespread detection.
- Samples with a `VT score` > 49 are considered definitively malicious, as they are already flagged by the vast majority of antivirus engines.

This division allows us to analyze the behavioral characteristics of known malware but also to investigate samples that may exploit vulnerable drivers under the radar of traditional detection systems.

B. Datasets

Known Vulnerable Drivers (KVD) Dataset. To construct the KVD dataset, we rely on two authoritative sources that catalog drivers known to be vulnerable. The first is the *Living Off The Land Drivers (LOLDrivers)* project [38], a community-driven initiative that maintains a curated list of Windows drivers with publicly disclosed vulnerabilities or exploitable design flaws. Security researchers contribute to this list by submitting driver samples via pull requests to a dedicated GitHub repository. Most of these drivers are legitimate, digitally signed, and remain loadable on fully updated Windows systems. At the time of collection, the repository included 1805 drivers, with VT scores ranging from 0 to 63. The second source is the *Microsoft Vulnerable Driver Blocklist* [39], a security mechanism introduced to prevent the misuse of known-vulnerable drivers in modern Windows environments. This list, enforced by default on Windows 11, contains fingerprints and Authenti-Hashes of drivers that are explicitly blocked from being loaded by the kernel. The list includes 420 unique driver hashes, with VT scores ranging from 0 to 32. Drivers from both sources form the foundation of the Known Vulnerable Drivers dataset. We merge the dataset into a unified dataset, and we apply the preliminary filtering Step ② described before. After this filtering phase, the KVD dataset contained 917 signed, 64-bit drivers (as shown in Table I), with 392 unique driver names.

From this list, we obtained a preliminary set of 186,705 execution parents. After sampling and clustering (Step ④), we obtained two groups of execution parents, categorized by their VT score. The first subset consists of 1948 execution parents with a VT score less than 6. These samples are still largely undetected by mainstream antivirus engines. The second subset includes 1047 executables with a VirusTotal score greater than 49, thus representing high-confidence malicious samples that are consistently flagged by the majority of detection engines. Following this selection procedure, the quantity of distinct drivers associated with at least one valid execution parent was reduced to 200.

The samples in these two groups are processed by our dynamic analysis system to examine how different classes of executables interact with drivers known to be vulnerable.

Potentially Vulnerable Drivers (PVD) Dataset. With the second dataset, we wanted to test whether our sandbox is able to discover drivers that, although not previously known to be vulnerable, exhibit characteristics commonly associated with BYOVD exploitation. This dataset assesses our approach's ability to detect emerging threats and behavioral patterns in the wild. For this reason, we specifically collected 64-bit Windows kernel drivers that statically import at least one function from the curated list of *Abusable Imports*, defined in Section III. After applying our filtering steps, we obtained 5589 signed, 64-bit drivers, as shown in Table I. When we categorize them based on their names, disregarding case sensitivity, we get 2000 unique driver names.

Table I: Datasets Cardinality.

Dataset	Initial Drivers	Execution Parents		Final Drivers
		< 6	> 49	
KVD	917 (392 distinct)	1948	1047	162
PVD	5589 (2000 distinct)	3582	2202	611

We then proceeded to download and cluster their corresponding execution parents, resulting in a final PVD dataset of 5784 executables: 3582 low-VT and 2202 high-VT samples. Following this selection procedure, the quantity of distinct drivers associated with at least one valid execution parent was reduced to 1,274. These, along with the associated drivers, form the core of our dynamic evaluation to identify novel BYOVD threats and assess the sandbox's detection capabilities beyond known vulnerabilities.

In conclusion, as can be inferred from Table I, in total we collected 8779 samples which in turn load a total of 773 drivers.

VI. Evaluation

To assess the effectiveness of our system in detecting Bring Your Own Vulnerable Driver (BYOVD) behaviors, we conducted a comprehensive evaluation guided by two research questions:

- (**RQ1**) Can our system successfully identify BYOVD behaviors triggered by known vulnerable drivers, as represented by the KVD dataset?
- (**RQ2**) Is our system capable of identifying novel instances of driver abuse in the PVD dataset, even when no prior reports of vulnerability exist?

A. Experimental Setup and System Configuration

All experiments were conducted in a virtualized environment built on the Xen hypervisor, with Windows 10 (version 22H2) as the guest operating system. The host machine was equipped with an Intel i7-12700 CPU @ 2.12GHz, 64 GB of RAM, and a 512GB NVMe SSD, providing the necessary resources for large-scale dynamic analysis. Our analysis system leveraged the *Kernelmon* plugin within the Drakvuf sandbox to trace kernel-level operations without modifying the guest system.

Each sample from the evaluation datasets was executed in our sandbox environment for a fixed duration of *two minutes*. This timeout was selected following prior guidelines on dynamic malware analysis, as proposed in [40]. During the execution, the analyzer actively monitors all potential behaviors defined in the *taxonomy* presented in Section III. In addition to generating a classification label for each execution, the system extracts a sequence of kernel function calls deemed relevant to the behavior observed. These function traces form the *Semantic Call Chains* associated with potentially malicious actions, such as process termination, memory remapping,

Table II: Evaluation results show executed samples, driver-loading behavior, and sandbox activity across full, low, and high VT score malware subsets.

Dataset	Executed samples			Samples loading at least one driver			Samples with observed behaviors		
	All	VT < 6	VT > 49	All	VT < 6	VT > 49	All	VT < 6	VT > 49
KVD	2995	1948 (65.04%)	1047 (34.96%)	1471	1003 (51.49% †)	468 (44.70% †)	304	194 (19.34% ‡)	110 (23.50% ‡)
PVD	5784	3582 (61.93%)	2202 (38.07%)	1412	868 (24.23% †)	544 (24.70% †)	442	167 (19.24% ‡)	275 (50.55% ‡)

† Percentage relative to the corresponding *executed samples* count

‡ Percentage relative to the corresponding *loading at least one driver* count

Table III: We report sandbox coverage by driver groups, highlighting how many had drivers loaded and showed observable behavior during execution.

Dataset	Driver Groups	Loaded	With an observed behavior	Without an observed behavior
KVD	162	106 (65.43%)	44 (41.51%)	62 (58.49%)
$\text{KVD}_{\text{VT}<6}$	139	91 (65.47%)	28 (30.77%)	63 (69.23%)
$\text{KVD}_{\text{VT}>49}$	95	64 (67.37%)	28 (43.75%)	36 (56.25%)
PVD	611	118 (19.31%)	48 (40.68%)	70 (59.32%)
$\text{PVD}_{\text{VT}<6}$	524	97 (18.51%)	38 (39.18%)	59 (60.82%)
$\text{PVD}_{\text{VT}>49}$	318	56 (17.61%)	19 (33.93%)	37 (66.07%)

or code integrity tampering, as detailed in Section IV. For samples exhibiting one or more suspicious behaviors, we conducted a preliminary inspection of the function trace to assess detection accuracy and context. In cases where further validation was required, a *manual analysis* was performed to confirm the nature of the reported behavior.

B. False Positives and False Negatives

Our system outputs behavioral information that may suggest whether a vulnerable driver has been exploited by a malware sample. In keeping with the traditional sandbox-based malware analysis approach, we focus on capturing and reporting runtime behavior, leaving detection and classification to downstream tools or human analysts. Crucially, because our sandbox only raises an alert once it observes the complete sequence of kernel events that formally defines a behavior with respect to our taxonomy, "False Positives" in the classic detection sense simply do not arise: every positive alert corresponds to a concrete execution trace rather than an inference or heuristic guess.

That said, determining whether a detected behavior constitutes an actual vulnerability, an intentional abuse, or a benign use case still requires the analyst to reconstruct the full attack chain, an inherently time-consuming task that demands human expertise. Thanks to our hypervisor-level visibility and precise kernel-event mapping, researchers can accurately map low-level actions to their respective attack classes and leverage existing user-mode tracing solutions to piece together the complete exploit vector.

On the other hand, False Negatives can arise in three cases: I) whenever a sample exercises a vulnerability pattern that our taxonomy does not cover, II) a code bug in our hooks or reconstruction rules, and III) if the relevant code path is never triggered within the execution window. In the first and second case, we developed our taxonomy with due scientific rigor, but it is possible that unknown techniques exist that indeed we cannot estimate–as the bugs in our code. In the third case, we will clarify better the possible causes of missed code coverage due to time limits or evasive behavior in Section VII.

C. RQ1: Detection of Known BYOVD Behaviors

For our first experiment, we analyzed execution parents associated with known vulnerable drivers (KVD dataset). Given that these drivers have been extensively studied and exploited in the wild and are documented in resources such as LOLDrivers and Microsoft's blocklist, we expect high confidence in the identification of suspicious behaviors, defined in our taxonomy, when these drivers are loaded and triggered within the sandbox environment. The likelihood of observing BYOVD activity increases further when the associated execution parent has a high VT score, which indicates a higher probability of malicious intent.

For this experiment, we executed a total of 2995 (i.e., 1948 + 1047) samples associated with 162 distinct known vulnerable drivers part of the initial KVD dataset as shown in Table I. Among these executions, our sandbox identified a total of 304 instances of execution parents that exhibited at least one suspicious behavior, representing approximately 10% of the analyzed samples.

Table II summarizes the results regarding the samples' evaluation. About half of the executions (1524 samples) did not exhibit any driver loading behavior, which prevented further analysis by our system. Based on our observations during the experiments, this lack of activity can be attributed to several factors: some samples may require user interaction to trigger their payloads, others serve solely as installers that drop drivers without executing them, and some may fail to run due to missing dependencies. It is also important to note that the execution parent relation provide by VirusTotal does not guarantee that the driver was actually loaded by the sample, but only that it was dropped onto the disk. Moreover, we conducted several statistical tests, reported in Appendix A-A due to space constraints, and here we discuss the statistically significant ones. We found that (I) once a driver is loaded, high-VT PVD samples show behaviors far more frequently ($p_{\text{Fisher}} < 10^{-6}$) than low-VT, (II) high-VT KVD samples actually load drivers less often than low-VT ones ($p_{\text{Fisher}} = 0.0004$), and (III) high-VT PVD samples exhibit much more suspicious behavior w.r.t. KVD.

Further investigation comparing their execution time suggests that phenomena II and III likely stem from a single underlying reason, i.e., evasive techniques. In case II, low-VT KVD drivers that have loaded at least one driver have been running longer than their high-VT counterparts ($p_{\text{Mann–Whitney}} = 1.81 \times 10^{-3}$, medium effect size). This could be due to the use of evasive techniques, which can be employed by the sample to detect the analysis environment and stop execution. While in case III, even if high-VT KVD drivers have shown less behavior, they have been running for a longer time than high-VT PVD drivers ($p_{\text{Mann–Whitney}} = 0.020$, medium effect size). Also in this case, we presume that the time-stalling evasive technique continued to keep the process alive but without performing additional actions.

Table III summarizes the results of our evaluation by analyzing distinct driver names in our KVD dataset. Our sandbox demonstrates promising performance in identifying driver-related suspicious behavior. Specifically, it achieves a precision of 43.75% for drivers associated with high VT score samples–indicating strong alignment with widely recognized threats and validating the effectiveness of our behavior-based detection approach. Even among low VT score samples, often representing novel or less detectable threats, the sandbox maintains a respectable precision of 30.77%, showcasing its potential for uncovering previously unknown or stealthy malicious activities.

To investigate the remaining 56.25% of samples for which no behaviors were detected, we combined manual analysis and, when possible for known drivers, we reviewed online reports from cybersecurity companies. This investigation revealed that the vast majority of these drivers are exploited for arbitrary memory read/write operations. As discussed in Section IV-C, our sandbox does not trace this type of vulnerability, which would require an instruction-level tracing, memory taint analysis, or extensive use of data breakpoints that would greatly affect the performance of the introspected OS. We manually reviewed the 36 clusters of drivers associated with high-score samples to assess whether the sandbox might be missing any behaviors described in our taxonomy. Upon closer inspection, and by analyzing the trace of the driver's execution context (e.g., IOCTL handler invocations), we determined that the majority of these drivers either exhibit arbitrary memory write capabilities[1] or are closely related variants, just with a different name, of other drivers for which the behavior was already observed and classified. Arbitrary memory writes, if performed outside monitored areas, are currently outside the scope of our dynamic-analysis system due to limitations discussed in section VII.

Moreover, Table III also shows that VT score is a weak discriminator: once a driver is actually loaded, the proportion of groups exhibiting at least one BYOVD behavior is quite similar. We conducted several statistical tests that are reported in Appendix A-A due to space limitations. The absence of statistically significant differences (at $\alpha = 0.05$) means that the role of VT scores is largely eliminated when aggregating by driver name rather than by sample. Consequently, the stronger VT scores effects observed at the sample level (Table II) are not present because many binaries map to the same driver, and likely benign or evasive samples dilute any correlation.

Table IV summarizes the observed behavior chains uncovered during dynamic execution. The most frequently detected behaviors include *MismatchedMemoryMapping* (172 occurrences), *PrivilegedUserHandleFromKernelLeak* (56 occurrences) and *ProtectedServiceTermination* (45 occurrences). Notably, 42 out of the 45 protected process terminations were observed in samples with a VirusTotal score greater than 49, providing strong evidence of malicious intent and BYOVD exploitation in high-confidence malware. It is worth noting, however, that the dynamic analysis reports provided by VirusTotal do not indicate any driver activity. Therefore, it is plausible that the high maliciousness scores associated with these execution parents originated from user-space malicious actions unrelated to the BYOVD behavior carried out by the samples. Moreover, *MismatchedMemoryMapping* can simply reflect a "bad" programming practice involving an API call that relaxes memory protections for a region (for example, remapping a page from execute to write or vice versa) and can therefore be benign. In contrast, terminating a protected process is almost invariably malicious, hence the stark difference in detection rates shown in our Table IV.

Our system was also able to reconstruct several complex behavioral chains, reflecting multi-stage exploitation techniques commonly observed in BYOVD attacks. The semantic call chains corresponding to these behaviors were analyzed to ensure accurate attribution to both the responsible driver and the originating user-mode process. When warranted, manual validation was conducted to confirm the result of the attack. In particular, all instances of *ProtectedServiceTermination* and *CodeIntegrityTampering* were manually reviewed and confirmed to be correct, highlighting the precision of the detection pipeline. It is important to note that while the sandbox is responsible for detecting suspicious behaviors at runtime, behavior validation is conducted in the post-processing phase of our full malware analysis pipeline. Thus, the sandbox serves as a reliable mechanism for flagging potentially malicious activities, which might then be subjected to deeper contextual inspection by an analyst. These results demonstrate that our sandbox accurately detects BYOVD activities by monitoring for behavior sequences characteristic of real-world attacks, fulfilling the objectives set out in RQ1.

D. RQ2: Detection of Novel or Suspicious Behaviors

To evaluate our system's ability to identify previously unreported or stealthy BYOVD threats, we analyzed samples from the PVD dataset. Since these drivers are signed and lack existing CVEs or blocklist entries, successful detections in this

[1]See the KDU project for a short list of drivers exhibiting arbitrary memory writes https://github.com/hfiref0x/KDU

Table IV: Observed Behavior Chains in the Execution Parents of the KVD Dataset

Observed Behavior Chain	Total	VT < 6	VT > 49
MismatchedMemoryMapping	171	148	23
PrivilegedUserHandleFromKernelLeak	56	31	25
ProtectedServiceTermination	45	3	42
UnsafePoolAllocation → DynamicCodeExecution	12	6	6
MismatchedMemoryMapping → UnsafePoolAllocation → DynamicCodeExecution	11	3	8
UnsafePoolAllocation → DynamicCodeExecution → MismatchedMemoryMapping	7	3	4
CodeIntegrityTampering → MismatchedMemoryMapping	1	0	1
CodeIntegrityTampering	1	0	1

context would result in the discovery of new BYOVD vectors still unknown to the security community. For this experiment, we executed a total of 5784 samples (i.e., 3582 + 2202) associated with 611 distinct potentially vulnerable drivers part of the initial PVD dataset as shown in Table I. As it happens for the execution parents of the KVD dataset, and due to the same reasons, the majority of the samples (4372 samples) did not exhibit any driver loading behavior, which prevented further analysis by our system. Among the executions, our sandbox identified 442 (8%) instances of suspicious behavior that can be associated with 48 drivers grouped by name, demonstrating the presence of potentially exploitable activity in a nontrivial portion of the analyzed samples. To further assess the risk, we manually investigated. In six cases, we were able to confirm that the malware sample successfully abused the driver to execute an attack. For the remaining drivers, when we only observed dynamic code execution, the exploitation requirements are more complex, and thus they require further analysis to determine their viability. We remind that *none* of these drivers were known to be used in BYOVD attacks.

Table V provides a breakdown of the most commonly observed behavior chains. The most prevalent detection was *MismatchedMemoryMapping*, occurring in 391 instances, followed by *ProtectedServiceTermination* (25 occurrences), and *PrivilegedUserHandleFromKernelLeak* (16 occurrences). Although these drivers are not publicly flagged as malicious, the presence of such behaviors strongly suggests that they may be used in BYOVD attacks. These patterns reflect classic BYOVD exploitation chains used to disable security tools, map executable memory, or bypass Code Integrity protections.

The analysis of the execution parents of PVD dataset exhibiting an observed behavior led to the discovery of previously undocumented drivers exhibiting BYOVD-like behavior. For instance, we identified multiple signed drivers—including variants of `Zemana.sys`, `MTKill.sys`, and `TfsMonSys.sys`—that enabled EDR termination or dynamic kernel code execution. We have submitted our findings to the LOLDrivers project as newly identified threats.

While our sandbox is primarily responsible for surfacing these behaviors, validation occurs in the downstream analysis pipeline. Therefore, also in this case, we verified all reported behaviors through a manual inspection of the semantic call chains to confirm their legitimacy.

Table V: Observed Behavior Chains in the Execution Parents of the PVD Dataset

Observed Behaviors Chain	Total
MismatchedMemoryMapping	391
ProtectedServiceTermination	25
PrivilegedUserHandleFromKernelLeak	16
CodeIntegrityTampering	5
UnsafePoolAllocation → DynamicCodeExecution	3
MismatchedMemoryMapping → ProtectedServiceTermination	1
UnsafePoolAllocation	1

E. Case Studies

Our sandbox enabled the identification of several noteworthy cases involving previously underreported or misclassified BYOVD attacks. Here we discuss three notable examples of vulnerable drivers detected using our sandbox, not present in LOLDrivers nor in Microsoft's blocklist, that demonstrate the effectiveness of our approach. We emphasize that these drivers can be loaded on modern Windows versions. The hashes of the drivers are reported in Table VII in the Appendix.

`kavservice.sys` – Targeted EDR Termination Our system identified four samples that leveraged the `kavservice.sys` driver to terminate protected processes. Our sandbox reconstructed the complete behavior chain: starting from the creation of the `\Device\MTKillDevice`, its symbolic link `\\.\MTKill`, and the invocation of IOCTL `0x222000` to perform process termination. Captured kernel call chains included `ZwOpenProcess` and `ZwTerminateProcess`, confirming the exploit. Manual analysis revealed that this driver was designed explicitly for this purpose, but we do not have sufficient context to claim that it was purposely developed for malicious use.

`termdd.sys` – Abusing a Legacy Microsoft Driver Four samples with VT scores ≤ 5 were found abusing `termdd.sys`, a legacy Terminal Services driver signed by Microsoft. The exploitation involved an arbitrary memory write to disable Code Integrity by modifying the `ci!g_CiOptions` variable, thus allowing the execution of unsigned drivers. Although this vulnerability has been known since 2010, the driver has not been added to the blocklist. Our sandbox accurately captured the memory tampering behavior and flagged the associated semantic call chain, emphasizing the risks posed by this legacy, yet still valid, signed drivers.

`probmon.sys` – Minifilter-Based Process Termination We observed one sample abusing `probmon.sys`, a legitimate Minifilter driver signed by ITM System, to perform unauthorized process termination. Rather than using IOCTL calls, the malware leveraged the `FilterSendMessage` API to communicate with the driver. Our system traced the non-standard communication channel and accurately attributed the behavior, demonstrating the ability to detect misuse even when drivers do not expose traditional device interfaces.

VII. Final Discussion

A. Thresholds of Execution Time and VT score

As with many malware analysis papers, determining the "best" thresholds, how long to run samples, and how to select them based on VT score remains an open and orthogonal problem. For this reason, we adopted state-of-the-art values [37], [40]. Consequently, our goal is to maximize coverage rather than exhaustively test different thresholds; adopting community-accepted settings lets us devote resources to analyzing a much broader corpus.

B. Memory-Based Exploits and the Scalability Barrier

Drivers might be vulnerable to powerful memory-write primitives that enable arbitrary kernel memory modification—an attractive capability for attackers aiming to disable security mechanisms or inject stealthy payloads. While it is theoretically possible to trace these writes using dynamic techniques such as memory breakpoints, doing so comprehensively would incur a prohibitive performance cost in virtualized environments. As a result, our system currently monitors only a limited set of memory regions, such as those related to Code Integrity enforcement, where modification is strongly indicative of exploitation. However, this selective approach may miss other high-value kernel targets. Identifying which memory regions to monitor *a priori* is non-trivial, as attackers continuously evolve their tactics to avoid detection.

Adding this coverage would require hooks for low-level memory operations, an approach that would incur a prohibitive runtime overhead. Extending the framework to monitor these primitives without sacrificing scalability is an important avenue for future work.

C. Detection of Emerging and Unknown BYOVD Behaviors

Our detection logic relies on well-studied behaviors, but BYOVD is evolving, and new drivers can introduce attack surfaces beyond our taxonomy. Uncovering these behaviors often demands manual analysis, deep code inspection, and reverse engineering. Enhancing detection will mean extending our framework—e.g., with machine-learning anomaly detection or automatic signature generation from high-fidelity traces—to cut manual effort and speed discovery of stealthy or obfuscated exploitation patterns.

D. Impact of Virtualization-Based Security (VBS)

Modern Windows systems are increasingly adopting security hardening features under the umbrella of Virtualization-Based Security (VBS), including Hypervisor-Enforced Code Integrity (HVCI). These mechanisms elevate the barrier for loading unsigned or maliciously modified drivers. While VBS represents a substantial improvement in OS hardening, it is not foolproof—numerous known-vulnerable drivers still load under HVCI as evidenced by our analysis. Our current sandbox does not simulate or analyze environments where VBS is actively enforced, potentially overlooking how these protections influence BYOVD viability. Future iterations of our system should emulate VBS-enabled configurations to understand which drivers are still exploitable in such contexts and to refine detection strategies accordingly.

E. Sandbox Evasion and Anti-Virtualization

The reliable execution of some malware samples can be influenced by the presence of a hypervisor. Prior research has shown that various malware families incorporate anti-analysis techniques capable of fingerprinting virtualized or sandboxed environments to evade detection and hinder execution [41], [42], [43], [44]. To evaluate the stealthiness of our sandbox and identify potential detection artifacts, we tested it using *Al-Khaser* [45], a tool specifically designed to assess evasive techniques. To distinguish genuine detection signals from false positives, we compared Al-Khaser's output on our sandboxed environment with its behavior on a clean Windows 10 installation running on bare metal. Our analysis revealed that 6.1% (19 out of 309) of Al-Khaser's detection techniques triggered alerts even on the bare-metal baseline, indicating these are false positives unrelated to virtualization. In contrast, 8.1% (25 out of 309) of the techniques flagged our sandbox environment. After excluding the false positives, only 1.6% (5 out of 306) of the detection methods uniquely identified our sandbox as an analysis environment. These results highlight the high level of transparency achieved by our sandbox.

VIII. Related Work

A. Virtualization-based Malware Analysis

Dynamic malware analysis, thanks to various techniques from bare-metal execution to code emulation, has become an essential method for examining the behavior of malicious software [46]. A key challenge in this context is determining the placement of the monitoring software. Garfinkel et al. [47] were among the first to propose an architecture in which the Intrusion Detection System (IDS) is no longer co-located with the malware on the host system but is instead fully isolated while still retaining complete visibility into the system's state—an approach called Virtual Machine Introspection (VMI). The introspection system must have detailed knowledge of the internal workings of the observed component in order to extract meaningful information [48], [49]. Early hypervisor-based systems focused primarily on kernel integrity protection, such as Petroni et al.'s Livewire [50] and SecVisor [51], which enforced control-flow integrity and memory protections within the kernel. Rkprofiler [6] is a sandbox-based tracking system designed to analyze kernel-level malware behavior by monitoring the execution of QEMU intermediate instructions, constructing a partial call graphs of the code traversed by the malware and offering valuable insights into its structure and execution flow. More recently, Leon et al. [52] proposed a hybrid dynamic analysis platform in which a trusted monitoring agent is installed within the guest OS to help bridge the semantic gap while leveraging a hypervisor for isolation and trace collection. Building on these foundations, *Ether*, introduced by Dinaburg et al. [53], leveraged hardware virtualization extensions (Intel VT) to perform transparent analysis of malware from outside the guest operating system, minimizing the risk of detection and tampering by the malware itself. *DRAKVUF*, developed by Lengyel et al. [28], is a prominent VMI-based framework that uses Xen to hook Windows kernel functions at runtime without injecting any code into the guest. It allows passive monitoring of malware execution at scale while preserving transparency. Our system builds on DRAKVUF by introducing a dedicated plugin, `kernelmon`, specifically designed to trace the runtime behavior of signed vulnerable drivers as abused in BYOVD attacks. In parallel, *HyperCheck* [54] and *HyperSentry* [55] extended VMI-based monitoring by integrating with System Management Mode (SMM) to perform stealthy, runtime validation of kernel integrity. These systems offer tamper resistance but do not reconstruct the behavior chains or attribute kernel-level activity to initiating user-space processes—key capabilities provided by our approach. Unlike previous systems, our sandbox not only enables high-fidelity kernel monitoring but also addresses the fundamental challenge of tracing the control flow within Windows kernel drivers.

B. Kernel Analysis and Driver Abuse Detection

Unlike the static analysis tool [25], which can only flag potentially malicious drivers based on the presence of dangerous APIs, our approach employs dynamic analysis to evaluate/extract the actual driver behavior based on kernel-level functions and events tracing, capturing complex behaviors such as dynamic code execution, memory mappings, and code integrity tampering. Beyond virtualization-based analysis, a significant body of research has addressed the detection of malicious activities directly within the kernel, with a particular focus on rootkits and driver-based threats. Early efforts such as *Copilot* [8] pioneered hardware-assisted runtime monitoring by verifying the integrity of kernel memory effectively detecting unauthorized modifications caused by rootkits. Later, Yin et al. proposed *HookScout* [9], a static and dynamic analysis system that detects kernel rootkits by identifying anomalous control flow caused by function hooking. Although effective against known hooking strategies, such techniques often struggle against obfuscation and dynamic resolution, which are common in modern BYOVD drivers. Several works have explored dynamic control-flow tracing and execution monitoring to detect abuse of kernel interfaces, especially for rootkit detection [10], [11]. Such systems use taint analysis for tracking specific events that happen in the kernel. In contrast, our system is designed not only to detect violations of kernel security policies but also to provide fine-grained semantic traces of driver-level execution. This enables a deeper understanding of how malicious user-space processes interact with vulnerable drivers to subvert kernel protections—something not addressed by prior rootkit detection or kernel integrity enforcement systems.

C. Bring Your Own Vulnerable Driver (BYOVD)

Despite its significance and rising use by sophisticated threat actors, BYOVD remains underexplored in academic research. The cybersecurity industry has provided most of the insights, documenting real-world attacks involving vulnerable drivers used to disable security tools, deploy rootkits, or tamper with protected memory. For example, Check Point [3] described the systematic abuse of the `truesight.sys` driver as part of an EDR evasion campaign. Their work also introduced a large-scale driver hunting methodology combining YARA rule filtering with manual analysis to identify drivers exposing dangerous kernel functionality. Climent-Pommeret [56] expanded on this by showing how to discover and validate drivers capable of killing protected processes using static imports and control flow analysis.

In parallel, VMware's Threat Analysis Unit (TAU) proposed an automated approach to identify vulnerable IOCTL handlers using IDA scripting and triage heuristics [57]. Their research uncovered dozens of previously undocumented IOCTL-based attack surfaces and confirmed that malware frequently leverages vulnerable drivers for stealthy code execution or privilege escalation. While Microsoft maintains a blocklist of known vulnerable drivers and community projects like LOLDrivers collect signatures and behaviors, these measures remain reactive and incomplete. Many signed, exploitable drivers are still actively used in the wild, often with little detection.

IX. Conclusions

This work shines a light on BYOVD attacks by pairing a five-stage behavior-centric taxonomy with a hypervisor-based sandbox that transparently follows every jump from user space down to kernel instructions. Across 8,779 malware samples loading 773 drivers, our system discovered driver abuse that traditional user-mode sandboxes miss and enabled the responsible disclosure of seven unknown vulnerable drivers. Overall, our results demonstrate that this study lays the first brick for closing a long-standing visibility gap and equips defenders with actionable intelligence to harden Windows systems.

Acknowledgement

This work was supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the Italian MUR. Neither the European Union nor the Italian MUR can be held responsible for them. Moreover, this work was also funded by two government grants managed by the French National Research Agency with references: "ANR-22-PECY-0007" and "ANR-23-IAS4-0001."

ETHICS CONSIDERATIONS

Our research involved analyzing and identifying security vulnerabilities in Windows drivers. Due to the potential risks associated with these vulnerabilities, we adhered to responsible disclosure guidelines throughout the study. Specifically, when we traced the vulnerabilities back to the respective vendors, we contacted them and provided detailed technical information, the malware samples that abuse them, and remediation suggestions. We communicated with the vendors and gave them 90 days to acknowledge, verify, and address the issues prior to any public disclosure. Additionally, we reported the vulnerabilities to Microsoft, MITRE, and the LOLDrivers project.

As of July 2025, we have submitted seven confirmed vulnerable drivers. Their hashes are reported in Table VII in the Appendix. We obtained `CVE-2024-26506` from MITRE for `probmon.sys`. However, we have not yet received any feedback from either Microsoft or the other companies involved.

REFERENCES

[1] M. Poslušný, "Signed kernel drivers – Unguarded gateway to Windows' core," https://www.welivesecurity.com/2022/01/11/signed-kernel-drivers-unguarded-gateway-windows-core/, 2022, [Online; accessed 18-April-2024].

[2] B. Cyber, "Qilin Ransomware and the Hidden Dangers of BYOVD," https://blackpointcyber.com/blog/qilin-ransomware-and-the-hidden-dangers-of-byovd/, 2025, [Online; accessed 18-April-2025].

[3] Checkpoint, "Silent Killers: Unmasking a Large-Scale Legacy Driver Exploitation Campaign," https://research.checkpoint.com/2025/large-scale-exploitation-of-legacy-driver/, 2024, [Online; accessed 24-March-2025].

[4] H. Yin, Z. Liang, and D. Song, "Hookfinder: Identifying and understanding malware hooking behaviors," 2008.

[5] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures." in *Ndss*, 2011.

[6] C. Xuan, J. Copeland, and R. Beyah, "Toward revealing kernel malware behavior in virtual execution environments," in *Recent Advances in Intrusion Detection*, E. Kirda, S. Jha, and D. Balzarotti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 304–325.

[7] H. Zhang, L. Zhao, A. Yu, L. Cai, and D. Meng, "Ranker: Early ransomware detection through kernel-level behavioral analysis," *IEEE Transactions on Information Forensics and Security*, 2024.

[8] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor." in *USENIX security symposium*. San Diego, USA, 2004, pp. 179–194.

[9] H. Yin, P. Poosankam, S. Hanna, and D. Song, "Hookscout: Proactive binary-centric hook detection," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2010, pp. 1–20.

[10] A. Lanzi, M. I. Sharif, W. Lee *et al.*, "K-tracer: A system for extracting kernel malware behavior." in *NDSS*, 2009.

[11] 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*, 2007, pp. 116–127.

[12] M. Leszczyński, A. Kliś, H. Jasudowicz, P. Srokosz, K. Cieśliński, A. Wróbel, and J. Jedynak, "DRAKVUF Sandbox," https://github.com/CERT-Polska/drakvuf-sandbox, 2022, [Online; accessed 19-March-2025].

[13] Monzani, Parata and Oliveri, Aonzo and Balzarotti, Lanzi, "Unveiling byovd threats - artifact," https://doi.org/10.5281/zenodo.15864111, 2025, malware and driver dataset, accessed on 2025-04-19.

[14] Microsoft, "Driver signing," [Online; accessed 18-April-2025]. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing

[15] ——, "Driver signing policy," [Online; accessed 18-April-2025]. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-

[16] M. Intelligence, "I solemnly swear my driver is up to no good: Hunting for attestation signed malware," 2022, [Online; accessed 18-April-2025]. [Online]. Available: https://cloud.google.com/blog/topics/threat-intelligence/hunting-attestation-signed-malware

[17] Microsoft, "Best practices for constraining high privileged behavior in kernel mode drivers," https://learn.microsoft.com/en-us/windows-hardware/drivers/driversecurity/driver-security-dev-best-practices, 2024, [Online; accessed 18-April-2025].

[18] ——, "Driver security checklist," https://learn.microsoft.com/en-us/windows-hardware/drivers/driversecurity/driver-security-checklist, 2024, [Online; accessed 18-April-2025].

[19] ——, "Configure added LSA protection," https://learn.microsoft.com/en-us/windows-server/security/credentials-protection-and-management/configuring-additional-lsa-protection, 2023, [Online; accessed 18-April-2025].

[20] E. Tamura, "Revisiting MiniFilter Abuse Technique to Blind EDR," https://tierzerosecurity.co.nz/2024/09/18/blind-edr-revisited.html, 2024, [Online; accessed 18-April-2025].

[21] G. Landau, "Forget vulnerable drivers - Admin is all you need," https://www.elastic.co/security-labs/forget-vulnerable-drivers-admin-is-all-you-need, 2023, [Online; accessed 18-April-2025].

[22] D. Pogonin and I. Korkin, "Microsoft Defender Will Be Defended: MemoryRanger Prevents Blinding Windows AV," *arXiv preprint arXiv:2210.02821*, 2022.

[23] J. Forshaw, "Injecting Code into Windows Protected Processes using COM - Part 1," https://googleprojectzero.blogspot.com/2018/10/injecting-code-into-windows-protected.html, 2018, [Online; accessed 18-April-2025].

[24] R. Boonen, "Capcom Rootkit Proof-Of-Concept," https://fuzzysecurity.com/tutorials/28.html, 2017, [Online; accessed 16-April-2025].

[25] J. Vinopal, "Breaking Boundaries: Investigating Vulnerable Drivers and Mitigating Risks," https://research.checkpoint.com/2024/breaking-boundaries-investigating-vulnerable-drivers-and-mitigating-risks/, 2024, [Online; accessed 18-April-2025].

[26] yara, "https://yara.readthedocs.io/en/stable/writingrules.html," https://yara.readthedocs.io/en/stable/writingrules.html, 2025, [Online; accessed 11-July-2025].

[27] J. Beulich, "Xen Project Wiki," https://wiki.xenproject.org/wiki/Main_Page, [Online; accessed 27-December-2024].

[28] 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*, 2014, pp. 386–395.

[29] P. Yosifovich, A. Ionescu, M. E. Russinovich, and D. A. Solomon, *Windows Internals 7th Edition, part 1*. Microsoft Press, 2017.

[30] A. Allievi, A. Ionescu, M. E. Russinovich, and D. A. Solomon, *Windows Internals 7th Edition, part 2*. Microsoft Press, 2021.

[31] A. Chester, "g_CiOptions in a Virtualized World," https://blog.xpnsec.com/gcioptions-in-a-virtualized-world/, 2022, [Online; accessed 12-March-2025].

[32] D. Nababkin, "The dusk of g_CiOptions: circumventing DSE with VBS enabled," https://blog.cryptoplague.net/main/research/windows-research/the-dusk-of-g_cioptions-circumventing-dse-with-vbs-enabled, 2024, [Online; accessed 12-March-2025].

[33] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 237–252. [Online]. Available: https://doi.org/10.1145/945445.945468

[34] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003, pp. 196–207.

[35] K. Cheng, Y. Zheng, T. Liu, L. Guan, P. Liu, H. Li, H. Zhu, K. Ye, and L. Sun, "Detecting vulnerabilities in linux-based embedded firmware with sse-based on-demand alias analysis," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 360–372.

[36] 2025, https://developers.virustotal.com/reference/files.

[37] S. Zhu, J. Shi, L. Yang, B. Qin, Z. Zhang, L. Song, and G. Wang, "Measuring and modeling the label dynamics of online {Anti-Malware} engines," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2361–2378.

[38] N. B. Michael Haag, Jose Hernandez, "Living off the land drivers," https://www.loldrivers.io/, 2025.

[39] Microsoft, "Microsoft recommended driver block rules," https://learn.microsoft.com/en-us/windows/security/application-security/application-control/app-control-for-business/design/microsoft-recommended-driver-block-rules, 2025.

[40] A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti, "Does every second count? time-based evolution of malware behavior in sandboxes," in *NDSS 2021, Network and Distributed Systems Security Symposium*. Internet Society, 2021.

[41] A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes *et al.*, "Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion," in *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings 19*. Springer, 2016, pp. 165–187.

[42] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 1009–1024, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:12665057

[43] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware dynamic analysis evasion techniques," *ACM Computing Surveys (CSUR)*, vol. 52, pp. 1 – 28, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:219884145

[44] L. Maffia, D. Nisi, P. Kotzias, G. Lagorio, S. Aonzo, and D. Balzarotti, "Longitudinal study of the prevalence of malware evasive techniques," *ArXiv*, vol. abs/2112.11289, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:245353367

[45] A. Faouzi, "Al-khaser," https://github.com/ayoubfaouzi/al-khaser, 2025.

[46] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, Mar. 2008. [Online]. Available: https://doi.org/10.1145/2089125.2089126

[47] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *Ndss*, vol. 3, no. 2003. San Diega, CA, 2003, pp. 191–206.

[48] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *2011 IEEE symposium on security and privacy*. IEEE, 2011, pp. 297–312.

[49] A. More and S. Tapaswi, "Virtual machine introspection: towards bridging the semantic gap," *Journal of Cloud Computing*, vol. 3, pp. 1–14, 2014.

[50] N. L. Petroni and M. W. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Conference on Computer and Communications Security*, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:14014247

[51] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Symposium on Operating Systems Principles*, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:89864

[52] R. S. Leon, M. Kiperberg, A. A. Leon Zabag, and N. J. Zaidenberg, "Hypervisor-assisted dynamic malware analysis," *Cybersecurity*, vol. 4, pp. 1–14, 2021.

[53] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 51–62. [Online]. Available: https://doi.org/10.1145/1455770.1455779

[54] F. Zhang, J. Wang, K. Sun, and A. Stavrou, "Hypercheck: A hardware-assistedintegrity monitor," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 4, pp. 332–344, 2014.

[55] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "Hypersentry: enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 38–49. [Online]. Available: https://doi.org/10.1145/1866307.1866313

[56] A. Climent-Pommeret, "Finding and exploiting process killer drivers with LOL for 3000$," https://alice.climent-pommeret.red/posts/process-killer-driver/, 2023, [Online; accessed 27-December-2024].

[57] T. Haruyama, "Hunting Vulnerable Kernel Drivers," https://blogs.vmware.com/security/2023/10/hunting-vulnerable-kernel-drivers.html, 2023, [Online; accessed 27-December-2024].

Appendix A
Appendix

Table VI: List of the *Abusable Imports* usually found in kernel drivers, along with a brief description of how each function can be misused.

Imported Function	Description
`IoCreateDevice` `IoCreateDeviceSecure`	If not called correctly, they can expose a device object with a weak discretionary ACL to the user mode, opening the possibility for an unprivileged application to communicate with the driver.

`MmMapIoSpace`
`MmMapIoSpaceEx`
`IoAllocateMdl`
`MmMapLockedPagesSpecifyCache`
	They can be used to map a specified *physical* address into a virtual address to access it directly.

`MmSystemRangeStart`
`ProbeForRead`
`ProbeForWrite`
	They can be used to obtain information about the memory configuration of a running system.

`ObCloseHandle`	It can be used to close handles that can be bounded to a specific process and usually passed to the user mode to reference kernel objects. Although closing a handle is a legitimate operation, a driver can cause a process to terminate by closing its handles.
`ZwMapViewOfSection`	It can be used to map *section* objects to the virtual memory. This technique is commonly used to map a library into different processes or to map physical memory ranges.

`ZwOpenProcess`
`ZwOpenThread`
`ZwOpenProcessTokenEx`
`ZwOpenProcessTokenEx`
`ZwCreateFile`
`IoCreateFile`
`ZwOpenSymbolicLinkObject`
	They can be used to obtain a *privileged* handle to an object in order to manipulate it without any security check on the caller.

`ZwSuspendthread`
`ZwAdjustPrivilegesToken`
`ZwDeleteFile`
`ZwDeleteKey`
	They are functions that can be used to manipulate handles opened with the previously mentioned methods.

`ZwTerminateProcess`	It can be used to terminate a process, also PPL ones.

Table VII: Reported Drivers

Driver Name	SHA1
`kavservice.bin`	be80f4d2a669f60354703a21daffb7b2128de190
`probmon.sys`	7310d6399683ba3eb2f695a2071e0e45891d743b
`IoBitUnlocker.sys`	abeedc8ac31eaee1948d3f56aa6c212cd9dc8c3a
`IoBitUnlocker.sys`	0e6ef35f6f68be6d72e4a225494c02557d39cacc
`Zemana.sys`	63399ac91e92f0c92ffaeac43616e1b7c77a9791
`TfSysMon.sys`	94493d7739c5ee7346da31d9523404d62682b195
`TrueSight.sys`	28c37b1c0af4a2a75a9662544fb3181a71c45dd2

A. *Formalization of the Analysis Pipeline*

To comprehensively capture the end-to-end behavior of BYOVD attacks and formalize our dynamic analysis system, we structure it into four distinct stages: (1) Execution Environment Setup, (2) Runtime Trace Collection, (3) Post-Execution Behavioral Analysis, and (4) Behavioral Classification and Attribution.

Stage 1: Execution Environment Setup – Let $\mathcal{E}$ be the controlled sandbox environment defined as:

$$\mathcal{E} = (\mathcal{V}, \mathcal{K}, \mathcal{M})$$

Where: $\mathcal{V}$ is a Xen-based hypervisor instance. $\mathcal{K}$ is the guest OS kernel and the introspection module that tracks its activity (e.g., Windows 10 with *Kernelmon*). $\mathcal{M}$ is the sample under test. The sandbox initializes a VM $v_i \in \mathcal{V}$ with $\mathcal{K}$, loads $\mathcal{M}$, and enables logging through the Drakvuf-based plugin *kernelmon*.

Stage 2: Runtime Trace Collection – During the sample execution, *kernelmon* intercepts system-level events and records them in our database as a serialized JSON trace $\mathcal{T}$.

$$\mathcal{T} = \{e_1, e_2, \dots, e_n\}, \quad e_i = (\mathsf{fn}_i, \mathsf{ts}_i, \mathsf{args}_i, \mathsf{ctx}_i)$$

Each event e_i contains the following information:

- fn_i: hooked kernel function name;
- ts_i: timestamp of invocation;
- args_i: function arguments and function address;
- ctx_i: execution context (including driver, process ID, process Name, parent process ID, privileges.).

Stage 3: Post-Execution Behavioral Analysis – Let $\mathcal{B} = \{b_1, b_2, \dots, b_m\}$ be the set of observable behaviors defined in Section III (e.g., *ProtectedProcessTermination*, *UnsafePoolAllocation*), we define a mapping function:

$$\Phi : \mathcal{T} \to \mathcal{B}$$

where for each $b_j \in \mathcal{B}$, $\Phi^{-1}(b_j) \subset \mathcal{T}$ is the minimal subset of events needed to confirm the behavior b_j.

Additionally, the reverse dataflow tracking of the behaviour b_j is performed by constructing the call chain:

$$\mathcal{C}_{b_j} = \langle e_i, e_k, \dots \rangle \quad \text{where } \Phi(\mathcal{C}_{b_j}) = b_j$$

Stage 4: Behavioral Classification and Attribution – The analysis results are expressed as a tuple:

$$\mathcal{R} = (\mathsf{DIDs}, \mathsf{PIDs}, \mathcal{B}, \mathcal{C})$$

where DIDs is the set of driver IDs (module names, hashes), PIDs is the set of originating process IDs, $\mathcal{B}$ are the Detected Behaviors, and $\mathcal{C}$ are the Corresponding Semantic Call Chains.

Each trace is evaluated against our set of potentially exploitable behaviors. $\mathcal{V}_{DB}$, defined in Section III, to determine the presence of BYOVD activity:

$$\mathsf{is\_BYOVD}(\mathcal{R}) = \begin{cases} \texttt{True}, & \text{if } \exists b_j \in \mathcal{B},\ b_j \text{ matches } \mathcal{V}_{DB} \\ \texttt{False}, & \text{otherwise} \end{cases}$$

End-to-End Pipeline – The full pipeline is then defined as:

$$\mathcal{P}(\mathcal{M}) = \mathsf{is\_BYOVD} \circ \Phi \circ \mathcal{T} \circ \mathcal{E}$$

Where $\mathcal{P}(\mathcal{M}) = \texttt{True}$ signals the detection of a BYOVD-based exploit triggered by the malware sample $\mathcal{M}$.

Statistical tests for Table II (sample-level)

Using 2×2 contingency analyses, we compared the proportions at the two different stages. (I) the proportion of executed samples that loaded at least one driver and (II) the proportion of loader samples that exhibited a BYOVD behavior, contrasting high- vs. low-VT groups for KVD and PVD. Fisher's exact test served as the primary inference method, complemented by Pearson's χ^2 and two-proportion z-tests; effect sizes are reported as risk differences (RD) and odds ratios (OR). In KVD, the load rate was lower for high-VT samples (468/1047, 44.7%) than for low-VT samples (1003/1948, 51.5%) [RD = -6.8%, OR = 0.76, $p_{\text{Fisher}} = 0.0004$], whereas the behavior rate among loaders was only slightly higher (110/468, 23.5% vs. 194/1003, 19.3%) and not significant [RD = $+4.2\%$, OR = 1.28, $p_{\text{Fisher}} = 0.072$]. In PVD, load rates were essentially identical (544/2202, 24.7% vs. 868/3582, 24.2%) [RD = $+0.5\%$, OR = 1.03, $p_{\text{Fisher}} = 0.68$], but the behavior rate among loaders was markedly higher for high-VT samples (275/544, 50.6% vs. 167/868, 19.2%) [RD = $+31.3\%$, OR = 4.29, $p_{\text{Fisher}} < 10^{-6}$]. Overall, VT score strongly correlates with observed behavior only in PVD, while differences either reverse or vanish at other stages.

Table VIII: Table II counts and Wilson 95% confidence intervals (CI) for proportions at two stages: loading (executed $\to$ loaded) and behavior (loaded $\to$ behavior).

Dataset	VT group	Executed	Loaded	Behav.	Load prop	CI_{low}	CI_{high}	Beh prop	CI_{low}	CI_{high}
KVD	Low VT	1948	1003	194	0.515	0.494	0.536	0.193	0.170	0.218
KVD	High VT	1047	468	110	0.447	0.416	0.479	0.235	0.197	0.277
PVD	Low VT	3582	868	167	0.242	0.229	0.256	0.192	0.167	0.219
PVD	High VT	2202	544	275	0.247	0.229	0.266	0.506	0.465	0.547

Table IX: High- vs. low-VT comparisons for Table II at each stage. Fisher's exact p-values are primary. Proportions use Wilson CIs.

Dataset	Stage	OR	p_{Fisher}	χ^2	p_{χ^2}	z	p_z	RD	RR
KVD	Load rate	0.7615	0.0004	12.5608	0.0004	-3.5441	0.0004	-0.0679	0.8681
KVD	Behavior rate	1.2813	0.0723	3.3720	0.0663	1.8363	0.0663	0.0416	1.2151
PVD	Load rate	1.0259	0.6821	0.1650	0.6846	0.4062	0.6846	0.0047	1.0194
PVD	Behavior rate	4.2912	0.0000	152.4672	0.0000	12.3478	0.0000	0.3131	2.6273

OR: odds ratio; RD: risk difference (HighVT–LowVT).

Then, to quantify the difference in observed behaviors, we compared the two high-VT groups with exact and asymptotic proportion tests. Fisher's exact test on the 2×2 table (110/468 vs. 275/544) was decisive ($p < 10^{-15}$; OR = 0.30), and a two-proportion z-test yielded $z = -8.84$, $p_{\text{Fisher}} = 1.2 \times 10^{-18}$. The risk difference (RD) was -27.1% with a Newcombe 95% CI of $[-34.9\%, -18.8\%]$; the risk ratio (RR) was 0.47 (Katz 95% CI: [0.39, 0.56]). Cohen's $h = -0.58$ indicates a large standardized effect.Thus, high-VT PVD drivers are far more likely to exhibit BYOVD behaviors than high-VT KVD drivers.

Table X: High-VT KVD vs. High-VT PVD behavior rates.

Group	Behaviors	Loaders	Proportion	95% CI (Wilson)
High-VT KVD	110	468	0.235	[0.197, 0.277]
High-VT PVD	275	544	0.506	[0.465, 0.547]
Between-group effects (High-VT KVD vs. High-VT PVD):				
RD = -0.270 (95% CI: $[-0.349, -0.188]$)				
RR = 0.465 (95% CI: $[0.387, 0.559]$)				
OR = 0.301 ($p_{\text{Fisher}} < 10^{-15}$)				
$z = -8.84$ ($p = 1.2 \times 10^{-18}$)				
Cohen's $h = -0.58$				

Then, given that high-VT KVD samples actually load drivers less often than low-VT ones, we compared their execution time. We adopted the Mann–Whitney U test (non-parametric location test) because the distributions departed from normality. With ties handled via average ranks and a two-sided $\alpha = 0.05$, we obtained $U = 1{,}182{,}191.0$ and $p = 1.81 \times 10^{-3}$, leading to rejection of the null hypothesis that the two distributions have the same central tendency. To quantify the magnitude and direction of the effect, we reported Cliff's delta $\delta = -0.375$. It indicates a medium effect size and a shift favoring the low-VT group (negative sign). This means that low-VT KVD drivers that have loaded at least one driver have been running longer than their high-VT counterparts.

Finally, we also compared with the same statistical methodology the execution time of high-VT KVD and PVD. We obtained $U = 1{,}804{,}191.0$ with $p = 0.020$, and $\delta = 0.385$. This medium effect size indicates that high-VT KVD drivers have been running for a longer time than high-VT PVD drivers.

Statistical analysis of Table III (driver-group level)

We compared the proportion of driver *groups* exhibiting at least one observed BYOVD behavior between high- and low-VT strata for both datasets (KVD and PVD). Wilson 95% confidence intervals (CI) are reported for proportions; Fisher's exact, two-proportion z-tests, and risk-difference CIs (Newcombe) were used for inference.

Table XI: Driver-group proportions with observed behaviors (Wilson 95% CI).

Dataset	VT group	Behav.	No behav.	Total	Prop.	CI_{low}	CI_{high}
KVD	High VT	28	36	64	0.4375	0.323	0.559
KVD	Low VT	28	63	91	0.3077	0.222	0.409
PVD	High VT	19	37	56	0.3393	0.229	0.470
PVD	Low VT	38	59	97	0.3918	0.301	0.491

Table XII: High vs. low VT comparisons per dataset.

Dataset	OR	p_{Fisher}	χ^2	p_{χ^2}	z	p_z	RD	95% CI_{RD}
KVD	1.75	0.23	2.74	0.10	1.66	0.098	+0.130	[–0.086, +0.337]
PVD	0.80	0.61	0.42	0.52	-0.65	0.518	-0.053	[–0.262, +0.170]

OR: odds ratio; RD: risk difference (HighVT–LowVT).

At the driver-group level, VT score does not significantly predict whether a loaded driver exhibits observable BYOVD behaviors. For KVD, 43.8% of high-VT vs. 30.8% of low-VT driver groups showed behaviors (Fisher $p = 0.23$; RD = +13.0%, 95% CI [–8.6%, +33.7%]). For PVD, the proportions were 33.9% vs. 39.2% (Fisher $p = 0.61$; RD = –5.3%, 95% CI [–26.2%, +17.0%]). These wide, overlapping intervals support the conclusion that VT score is a weak discriminator once a driver is actually loaded; the stronger VT-related differences seen at the *sample* level (Table II) largely vanish when aggregating by driver name.

Appendix B
Artifact Appendix

A. Description & Requirements

1) How to access: Artifact is available on Zenodo at the following link: https://doi.org/10.5281/zenodo.15864111

2) Hardware dependencies: Intel CPU with virtualization support (VT-x) and with Extended Page Tables (EPT)

3) Software dependencies: DRAKVUF (tag 1.0 from github) and drakvuf-sandbox, tag 0.18.2.

4) Benchmarks: Drivers and executable samples. Hashes has been provided in the artifact.

B. Artifact Installation & Configuration

Source code and dataset are provided. To correctly install and configure the artifact for a complete evaluation, both Drakvuf and the drakvuf-sandbox framework must be compiled from the source code and installed. Drakvuf-sandbox documentation explains how to configure a Windows guest machine where samples will be detonated. Additional information is available upon request by contacting the authors.

C. Major Claims

- (C1): We implemented *kernelmon*, a plugin for DRAKVUF capable of tracing kernel-level functions and extract context and parameters information. This is detailed in sub-section *A. System Architecture & Requirements* from section *IV. APPROACH*.
- (C2): We implemented *kernelmon_analysis*, an analyzer that is able to reconstruct the function call data-flow and identifies suspicious behaviors. Our system was able to uncover suspicious behaviors in previously unknown drivers. This is detailed by the case studies provided in sub-section *E. Findings and Highlights* from section *VI. EVALUATION*.
- (C3): We constructed a *dataset* of drivers and executable samples to test and validate our sandbox extension. This is detailed in section *V. DATASET DESIGN AND METHODOLOGY*.

D. Evaluation

None