

A Hybrid Static-Dynamic Analysis for Identifying BYOVD Vulnerabilities

Andrea Monzani^a, Antonio Parata^a, Davide Netti^a, Andrea Romaniello^a, Andrea Oliveri^b, Simone Aonzo^b, Davide Balzarotti^b and Andrea Lanzi^a

^aUniversity of Milan, Italy

^bEURECOM, France

ARTICLE INFO

Keywords:

BYOVD
Static analysis
Symbolic execution
Dynamic analysis
Security
Malware analysis
Sandbox

Abstract

Bring Your Own Vulnerable Driver (BYOVD) attacks abuse legitimate, signed Windows kernel drivers to perform privileged operations that would otherwise be impossible, even for users with administrative privileges. While dynamic analysis can provide runtime evidence of security-relevant driver behaviors, it is inherently limited to the code paths exercised during execution; conversely, static analysis can explore additional feasible paths but may over-approximate exploitability.

In this paper, we present a hybrid static-dynamic pipeline for identifying BYOVD-relevant driver vulnerabilities. Our approach combines an extended version of *IOCTLance* for symbolic analysis of driver-exposed attack surfaces with an enhanced *kernelmon*, a *DRAKVUF* module, for observing runtime exploitation primitives at the hypervisor layer. We extend the state-of-the-art taxonomy of observable behaviors from 7 to 15 categories and integrate new detectors on both the static and dynamic sides. We evaluate our pipeline on 6,508 signed kernel drivers and 9,109 execution parents. Static analysis identifies at least one suspicious behavior in 72.1% of known vulnerable drivers and 23.2% of potentially vulnerable drivers. Dynamic analysis observes security-relevant runtime behaviors in 63.2% and 31.9% of known-vulnerable and potentially-vulnerable driver groups, respectively. By correlating static and dynamic findings, our system exposes complementary behaviors that would be missed by either technique alone and contributes to the identification of 13 previously undocumented vulnerabilities, leading to multiple CVE assignments. Our dynamic evaluation is conducted on Windows 10 without Virtualization-Based Security (VBS)/Hypervisor-Enforced Code Integrity (HVCI); therefore, the reported dynamic detection rates should not be interpreted as measurements of driver exploitability under VBS/HVCI-enabled systems.

1. Introduction

Bring Your Own Vulnerable Driver (BYOVD) attacks abuse legitimate, digitally signed Windows kernel drivers to bypass security mechanisms, disable defenses, and perform privileged operations from user mode. In our previous work [39], we introduced a dynamic taxonomy of BYOVD behaviors and extended a virtualization-based sandbox to transparently trace kernel-level execution initiated by user-space processes. By correlating kernel traces, the system exposed real-world BYOVD abuses that were invisible to traditional sandboxes.

However, this approach inherits a structural limitation of dynamic analysis: observations are restricted to execution paths that are concretely exercised during a run. Vulnerable driver behaviors may remain undetected when guarded by conditional logic, which depends on specific runtime states, or are not triggered by the available execution parents.

This is particularly relevant to BYOVD analysis, where the existence of a vulnerable driver path does not imply that a corresponding exploit will be exercised during sandbox execution.

To address this coverage gap, we combine dynamic and static analysis: dynamic analysis provides concrete evidence of behaviors exercised at runtime, whereas static analysis enables systematic exploration of feasible driver paths independently of runtime triggers. Their combination, therefore, captures both observed behaviors and potentially exploitable capabilities undetected during a particular execution.

Prior research on adaptive defense, including moving-target defense and game-theoretic strategy selection [51, 52], highlights the importance of reliable information about attacker capabilities when selecting countermeasures. Although these approaches address a different layer of the defense problem, they motivate the need to characterize both observed and latent attacker capabilities. In the BYOVD context, this requires identifying both behaviors observed at runtime and privileged capabilities exposed by drivers that remain dormant during a particular execution – complementary information that our hybrid analysis system can provide.

Building on this approach, we extend both analysis components: (i) on the static side, we enhance *IOCTLance* [32] with new BYOVD-oriented detectors for previously unsupported misuse of privileged kernel functionality; (ii) on the dynamic side, we extend *kernelmon* with selective

doi: 10.1016/j.cose.2026.105194

✉ andrea.monzani@unimi.it (A. Monzani); antonio.parata@unimi.it

(A. Parata); davide.netti@studenti.unimi.it (D. Netti);

andrea.romaniello@studenti.unimi.it (A. Romaniello);

andrea.oliveri@eurecom.fr (A. Oliveri); simone.aonzo@eurecom.fr (S.

Aonzo); davide.balzarotti@eurecom.fr (D. Balzarotti);

andrea.lanzi@unimi.it (A. Lanzi)

ORCID(s): 0009-0009-3301-5253 (A. Monzani); 0009-0003-7027-7590

(A. Parata); 0009-0008-9406-5089 (D. Netti); 0009-0000-5073-0958 (A.

Romaniello); 0000-0001-7820-1927 (A. Oliveri); 0000-0001-9547-3502 (S.

Aonzo); 0000-0001-5957-6213 (D. Balzarotti); 0000-0002-1544-3758 (A.

Lanzi)

hypervisor-level instrumentation for memory and permission-related exploitation primitives. The enhanced sandbox monitors fine-grained kernel events and correlates them with the driver execution context, enabling the detection of runtime-dependent behaviors that are difficult to infer through symbolic execution alone. Figure 1 summarizes the unified pipeline. Kernel drivers are analyzed by the static component, whereas user-mode executables are executed in the dynamic environment; drivers loaded during execution are automatically forwarded to the static analyzer. This workflow combines potential capability discovery with concrete runtime evidence. We evaluate the approach on 6,508 signed kernel drivers and 9,109 execution parents, covering known vulnerable (KVD) and potentially vulnerable (PVD) drivers. KVD contains drivers with documented vulnerabilities, whereas PVD contains drivers selected according to the abusable kernel functions identified in our previous work. Static analysis identifies suspicious BYOVD-related behaviors in 72.1% of KVD and 23.2% of PVD drivers, while dynamic analysis observes runtime behaviors in 63.2% and 31.9% of successfully loaded KVD and PVD driver groups, respectively.

This paper makes the following contributions:

1. **Extended BYOVD behavior taxonomy.** We expand the taxonomy from 7 to 15 observable behaviors, covering additional exploitation primitives and runtime-dependent misuse patterns.
2. **Extended symbolic driver analysis.** We enhance *IOCTLance* with three BYOVD-oriented detectors, now integrated into its official repository (PR#4, PR#8, PR#15).
3. **Improved dynamic detection.** We extend *kernelmon* with selective hypervisor-level instrumentation for memory-based and taint-related exploitation primitives.
4. **Hybrid driver inspection.** We integrate static and dynamic analysis into a unified pipeline that combines potential capability discovery with runtime behavioral evidence.
5. **Real-world findings.** The resulting system contributed to the discovery of 13 previously undocumented vulnerable drivers, five of which have received CVE assignments.

Ethics Considerations. Our research involved the analysis and identification of security vulnerabilities in Windows drivers. Given the potential risks, we strictly followed *responsible disclosure* practices throughout the study. When vulnerabilities were attributed to specific vendors, we contacted them with detailed technical reports, including affected malware samples and remediation suggestions. Vendors were granted a 90-day period to acknowledge, verify, and address the issues before any public disclosure. We additionally reported all findings to Microsoft and MITRE. Of the 13 reported, five have received CVEs: CVE-2025-8412, CVE-2026-24076, CVE-2025-14963, CVE-2026-21709 and

CVE-2026-35553. Three drivers are still under vendor review. Table 15 summarizes their hashes, names, associated behaviors, and responsible disclosure timeline.

2. Background

Windows kernel drivers are privileged components that execute in kernel mode and provide access to hardware, low-level system services, and security-sensitive functionality. Modern Windows systems enforce *Driver Signature Enforcement* (DSE), which requires kernel drivers to be digitally signed before loading [35, 36]. However, DSE does not prevent vulnerabilities in legitimately signed drivers or constrain the semantics of their operations [37].

2.1. Bring Your Own Vulnerable Driver

Bring Your Own Vulnerable Driver (BYOVD) attacks exploit legitimate but vulnerable signed drivers embedded in malicious payloads [45]. Once loaded, these drivers execute with kernel privileges and can act as a *confused deputy*, performing privileged operations on behalf of user-mode applications. Typical abuse patterns include protected-process termination, arbitrary kernel-memory access, Code Integrity tampering, and execution of attacker-controlled code [22, 55]. BYOVD vulnerabilities are commonly exposed through IOCTL handlers, which mediate interactions between user-mode and kernel-mode code. These handlers process user-controlled inputs and may invoke privileged APIs such as *ZwOpenProcess*, *ZwTerminateProcess*, *MmMapIoSpace*, and *ZwMapViewOfSection*. Analyzing whether user-controlled data can influence such sensitive operations motivates the use of both static and dynamic analysis.

2.2. Static Analysis of Kernel Drivers

Static analysis inspects program code without executing it, enabling reasoning about control flow, data flow, and potential vulnerabilities [20]. Symbolic execution is particularly useful for kernel drivers because it can track user-controlled inputs from IOCTL buffers along feasible paths to privileged kernel operations. *IOCTLance* [32], for example, combines symbolic execution and taint analysis to explore IOCTL handlers and identify latent vulnerabilities. Static analysis, nevertheless, faces important limitations in kernel code. Drivers rely on complex kernel data structures, undocumented interfaces, indirect calls, and event-driven execution. Symbolic exploration is also subject to path explosion and scalability issues, particularly in the presence of input-dependent branching [23, 48].

2.3. Dynamic Analysis of Kernel Drivers

Dynamic analysis observes concrete program behavior during execution. In our previous work [39], we showed that hypervisor-based introspection provides visibility into driver execution while enabling suspicious activity to be attributed to specific drivers and processes. By instrumenting kernel API and execution paths, dynamic analysis can observe driver loading, IOCTL handling, privileged API invocations,

and memory modifications while remaining resilient to techniques such as packing and dynamic API resolution [2, 43]. Its main limitation is execution coverage. Vulnerable paths may depend on specific inputs, runtime conditions, or user-space interactions that are not reproduced during analysis. Dynamic analysis, therefore, underestimates the driver attack surface, observing only behaviors concretely exercised during execution [43].

3. Threat Model and Analysis Assumptions

Our hybrid pipeline is designed to integrate into a malware ingestion framework, where static and dynamic analysis produce artifacts for downstream automated tools or security analysts. For dynamic analysis, we assume an attacker is able to execute a user-mode program with administrative privileges and deploy and load legitimate signed Windows drivers. The attacker controls the inputs provided to the driver, including IOCTL requests, communication channels when applicable, input buffers, and the ordering of driver-interface invocations. Administrative access is therefore a precondition of our threat model, while user-mode privilege escalation is outside the scope of this work. We reproduce this scenario in a Windows 10 sandbox where samples run with administrative privileges. Windows Defender remains enabled, but real-time scanning excludes the analyzed files and processes to prevent known samples from being blocked before execution. Since BYOVD attacks frequently target security products, the sandbox also emulates the presence of common antivirus and EDR processes. We assume that the attacker may provide inputs capable of exercising vulnerable code paths, but we do not assume that every feasible path is triggered during analysis, as execution may depend on runtime conditions, system state, or the availability of a concrete exploit. Monitoring is performed outside the guest through Virtual Machine Introspection (VMI) at the hypervisor level; accordingly, we assume that guest code cannot tamper with the monitoring infrastructure. The static component, implemented using IOCTLance, conservatively reports BYOVD-relevant capabilities reachable along feasible symbolic paths, independently of whether they are exercised dynamically. Conversely, dynamic analysis provides concrete runtime evidence only for behaviors triggered by the executed sample and its inputs. The hybrid pipeline combines these complementary views, while neither component alone guarantees complete coverage or exploitability.

4. System Design

Building on the threat model defined above, our system combines static and dynamic analysis to provide complementary visibility into BYOVD-relevant driver behaviors. Static detections identify potentially exposed capabilities along feasible paths, whereas dynamic detections provide evidence of behaviors concretely exercised at runtime.

To clarify the preconditions and limitations of these two complementary approaches, we refer to the `ioctl` handler shown in Listing 1, which represents a simplified instance

of a *ProtectedServiceTermination* suspicious behavior. The typical implementation of a process-termination feature exposed by a driver generally involves: (i) registering the `ioctl` handler function during driver initialization (not shown), (ii) retrieving the `ioctl` code and its associated input buffer from kernel structures, (iii) validating the input parameters, and (iv) invoking the appropriate kernel functions, specifically `ZwOpenProcess` and `ZwTerminateProcess` in this case, to perform the requested action. Since `ZwOpenProcess` and `ZwTerminateProcess` are privileged kernel functions, failing to enforce or incorrectly implementing proper restrictions on their input parameters (such as the process ID and the associated handle of the process to be terminated) can lead to vulnerable code. This may allow an unprivileged application to perform actions that are normally forbidden in user mode, such as terminating a protected process [37].

From the point of view of **static analysis**, to detect the potential *ProtectedServiceTermination* vulnerability in the PE file of a driver, a symbolic execution tool should:

- Find the `IoctlHandler` function by navigating to the driver's entry point
- Explore all possible code paths inside the `IoctlHandler` function ④
- Reach the `KillProcess` function ⑪ (under some constraints on the `ioctl` code and the input buffer content ⑩)
- Simulate the kernel calls
- Identify taints, propagate them, and deduce that the process's handle entering the `ZwTerminateProcess` call can be indirectly controlled by a user-mode application, as the `ZwOpenProcess`'s main input parameter, `ClientId`, is directly dependent on the `ioctl` input buffer content.

Symbolic execution is a powerful and readily applicable technique for identifying suspicious behavior in driver code. However, its inherent need to explore all possible execution paths significantly limits its effectiveness. Analyzing complex or packed drivers can be particularly challenging, as the time and memory required to identify even the IOCTL handler address from the driver entry point may be prohibitive.

To identify the same vulnerability using **dynamic analysis** in a sandboxed environment, we must rely on a different set of preconditions:

- The driver containing the vulnerable code must be loaded into the kernel. In this case, a sample that interacts with the guest OS and deploys the driver is needed
- The `ioctl` responsible for the vulnerable functionality must be invoked ⑨
- Any internal trigger conditions required by the driver must be satisfied ⑩
- The kernel calls involved must return the expected values (i.e., no errors and no unexpected value changes that could alter the execution path)

Dynamic analysis offers a practical way to partially overcome the limitations of symbolic execution. By directly hooking kernel calls, it is possible to detect suspicious behavior embedded in packed or complex drivers with significantly lower overhead compared to symbolic execution. However, as discussed in the first requirement, this approach relies on the existence of a sample capable of loading and interacting with the driver.

This example illustrates the complementary detection capabilities of static and dynamic analysis and anticipates the results discussed later in the paper. Certain BYOVD behaviors can be identified by both approaches. Some behaviors may be observable exclusively through symbolic analysis, such as when a sample does not load a driver, loads a driver without invoking the vulnerable IOCTL, or invokes the IOCTL without satisfying the runtime conditions required to trigger the vulnerable behavior (e.g., specific kernel function return values). Conversely, others may be detectable only through dynamic analysis, for instance, when driver complexity, packing, or execution semantics depend on runtime attributes that cannot be accurately modeled symbolically.

```

1  #define IOCTL_PROCESS_CMD ...
2  #define TERMINATE_PROC_SUBCMD ...
3
4  NTSTATUS IoctlHandler(PDEVICE_OBJECT Device, PIRP Irp) {
5      DWORD Ioctl = GetIoctl(Device, Irp);
6      DWORD* InputBuffer = GetInputBuffer(Device, Irp);
7      switch (Ioctl)
8      {
9          case IOCTL_PROCESS_CMD:
10             if (InputBuffer[0] == TERMINATE_PROC_SUBCMD) {
11                 KillProcess(InputBuffer[1]);
12             }
13             return STATUS_SUCCESS;
14         }
15     }
16
17     void KillProcess(DWORD Pid) {
18         OBJECT_ATTRIBUTES ObjectAttributes;
19         InitializeObjectAttributes(&ObjectAttributes, NULL,
20             <- OBJ_INHERIT, NULL, NULL);
21
22         CLIENT_ID ClientId;
23         ClientId.UniqueProcess = (HANDLE)u1ProcessID;
24         ClientId.UniqueThread = NULL;
25
26         HANDLE hProcess;
27         ZwOpenProcess(&hProcess, PROCESS_ALL_ACCESS,
28             <- &ObjectAttributes, &ClientId);
29         ZwTerminateProcess(hProcess, 0);
30         ZwClose(hProcess);
31     }

```

Listing 1: Terminate vulnerability example

5. Observable Suspicious Behavior

In our prior work, we introduced observable suspicious behaviors capturing security-relevant outcomes of BYOVD attacks, including protected process termination, unsafe memory mappings, and Code Integrity tampering. Subsequent analysis, however, showed that vulnerable drivers may expose privileged kernel capabilities without triggering a complete attack chain during dynamic execution. We therefore extend the taxonomy with intermediate misuse patterns and exploitation-relevant primitives. These behaviors do not necessarily indicate a concrete exploit, but conservatively identify cases where user mode can influence privileged kernel operations. We generalize the previous behaviors (B1-B7) beyond a specific monitoring architecture and introduce new behaviors (B8-B15) covering user-influenced kernel interactions and sensitive interfaces.

[B1] Protected Service Termination Detection – The system detects unauthorized protected-process termination by reconstructing the sequence from handle acquisition (e.g., `ZwOpenProcess` or `ObReferenceObjectByHandle`) to `ZwTerminateProcess`. A representative vulnerable driver is `mhyprot2.sys` [53].

[B2] Privileged User Handle Leakage Monitoring – The system monitors `ZwCreate` and `ZwOpen` operations where `OBJ_KERNEL_HANDLE` is absent, which may expose privileged handles to user mode. Dynamic analysis additionally monitors their use through `ObpReferenceObjectByHandle`. This behavior occurs in the vulnerable `ElbyCDIO.sys` driver [26].

[B3] Detection of Unsafe Pool Allocations – The system monitors non-pageable kernel allocations with writable and executable (WX) permissions, including `PAGE_EXECUTE_READWRITE` memory allocated from `NonPagedPool`. This behavior is reported dynamically only when execution in the allocated region is observed, avoiding alerts on legitimate allocations.

[B4] Dynamic Kernel Code Execution Tracking – The system detects execution within dynamically allocated WX kernel regions and attributes it to the responsible driver or IOCTL. Such events may represent the final stage of exploits that map attacker-controlled code into kernel memory [8].

[B5] Mismatched Memory Mapping Detection – The system intercepts remapping operations such as `MmMapIoSpace` and `MmMapLockedPagesSpecifyCache`, comparing requested permissions with those of the original region. Permission mismatches, such as mapping non-executable memory as executable, are reported as potentially unsafe remappings [7].

[B6] Real-Time Code Integrity Tampering Detection – The system monitors modifications to security-sensitive Code Integrity structures, including `g_CiOptions` [10] and callback tables [41], recording the responsible instruction and execution context. This captures attempts to disable kernel integrity mechanisms through vulnerable drivers, including patterns associated with B14 and B15 [1].

[B7] Detection of Remote Handle Closure – The system detects cross-process handle manipulation involving

KeStackAttachProcess followed by ObCloseHandle. Such behavior can disrupt security processes, as in *AuKill*, which abuses *procexp.sys* to close AV process handles [25].

[B8] Detection of Arbitrary Memory Writes in Kernel Space – During IOCTL handling, the system identifies memory writes whose destination may be controlled through the IOCTL input. IOCTLance determines whether the destination address derives from a value dereferenced from the IOCTL buffer, while *kernelmon* monitors driver write instructions and checks whether their destination is derived from input-buffer values. Such primitives can modify sensitive kernel data, as demonstrated by *gdrv.sys* (CVE-2018-19320) [6].

[B9] Detection of Arbitrary MSR Writes in Kernel Space – The system monitors Model-Specific Register (MSR) writes and assesses whether their index and value are user-controlled. IOCTLance instruments *wrmsr* instructions, while *kernelmon* relies on hypervisor-level MSR notifications. Such primitives, exposed by drivers like *DirectIo64.sys* [45], can be exploited as shown in [12].

[B10] Detection of Arbitrary Port I/O in Kernel Space – The system monitors out instructions during IOCTL handling and reports operations whose port and value originate from user-controlled input. Both IOCTLance and *kernelmon* instrument these instructions, as the VMI layer used by DRAKVUF does not expose hypervisor-level I/O notifications. *Inpoutx64.sys* exposes such a primitive [50], which can enable low-level disk operations [13].

[B11] Detection of Tainted Physical Memory Mapping – The system monitors *NtOpenSection* calls targeting *\Device\PhysicalMemory* and subsequent *NtMapViewOfSection* operations, flagging user-controlled mapping offsets or sizes. Dynamic analysis also monitors subsequent writes or execution in the mapped pages, as observed with *WinRing0.sys* [46].

[B12] Detection of Tainted Kernel Arguments – The system detects user-controlled arguments passed to privileged APIs, including *MmMapIoSpace(Ex)*, *ZwMapViewOfSection(Ex)*, and object names provided through *OBJECT_ATTRIBUTES*. This behavior is also exhibited by *ElbyCDIO.sys* (see **B2**).

[B13] Detection of Executable or Writable Memory Remapping – The system traps remapped regions to detect execution in previously non-executable memory or writes to previously non-writable memory. This provides more precise dynamic validation of the remapping vulnerabilities described in [7].

[B14] Detection of Unsafe RtlQueryRegistryValues Calls – The system detects *RtlQueryRegistryValues* invocations using *RTL_QUERY_REGISTRY_DIRECT*, which may enable type-confusion vulnerabilities under specific registry configurations, as observed in *termdd.sys* [1].

[B15] Detection of RtlQueryRegistryValues Tampering – The system identifies concrete tampering associated with *RtlQueryRegistryValues* by monitoring malformed *RtlpQueryRegistryDirect* invocations and in-memory modifications to the

Table 1

Implementation status for supported behaviors across the static and dynamic components.

B#	Behavior	IOCTLance	kernelmon
B1	ProtectedServiceTermination	+	o
B2	PrivilegedUserHandleLeak	o	o
B3	UnsafePoolAllocation		o
B4	DynamicCodeExecution		o
B5	MismatchedMemoryMapping		o
B6	CodeIntegrityTampering		o
B7	RemoteHandleClosed	+	o
B8	PotentialArbitraryWrite	o	+
B9	PotentialArbitraryMsWrite	o	+
B10	PotentialArbitraryOutWrite	o	+
B11	PotentialArbitraryPhyMemMap	o	+
B12	PotentialKernelArgumentasTaint	o	+
B13	RemappedMemoryXOrW		+
B14	UnsafeRtlQueryRegistryCall	+	+
B15	RtlQueryRegistryTampering		+

o Behavior already implemented before this extension

+ Behavior implemented with this extension

(empty) Behavior not implemented due to unsupported preconditions for the analysis type

Table 2

Datasets cardinalities.

	Drivers Initial Set (*VDIS Dataset)	Execution Parents		Drivers With an Execution Parent (*VD Dataset)
		< 6	> 49	
Known	919 (392 distinct)	1961	1436	369 (162 distinct)
Potentially	5589 (1840 distinct)	3617	2095	1150 (603 distinct)

QueryTable. This provides a dynamic validation mechanism for the unsafe patterns captured by **B14**.

Table 1 summarizes behavior support across the static and dynamic components: o denotes previously implemented behaviors, + denotes behaviors introduced in this work, and an empty entry indicates unsupported preconditions. Table 14 provides the complete taxonomy, including detection goals, static and dynamic patterns, and representative real-world examples.

6. Dataset

We analyze two driver datasets constructed using the same methodology introduced in our previous work. As before, our analysis does not consider drivers in isolation. In addition to the driver binaries themselves, we collect and analyze the corresponding user-space executables responsible for deploying and potentially exploiting them. These executables, referred to as execution parents, are defined as samples observed writing the driver binary to disk prior to its loading. This ensures consistency with our earlier study while enabling a direct comparison of results across static, dynamic, and hybrid analyzes.

6.1. Known Vulnerable Drivers (KVD) Dataset

The first dataset, referred to as KVD (Known Vulnerable Drivers), contains drivers that are already acknowledged by the security community as vulnerable. It serves as a control group for evaluating the effectiveness of the actual symbolic and dynamic analysis components.

To construct the Known Vulnerable Drivers dataset, we first define the Known Vulnerable Drivers Initial Set (KVDIS) by collecting driver samples from two authoritative sources that track Windows drivers with documented security issues. The first source is the *Living Off The Land Drivers (LOLDrivers)* project [34], a community-maintained repository that catalogs Windows drivers exhibiting publicly known vulnerabilities or design flaws. At the time of collection, LOLDrivers listed 1805 drivers, with VirusTotal (VT) scores ranging from 0 to 63. The second source is the *Microsoft Vulnerable Driver Blocklist*, a security mechanism enabled by default on Windows 11 that prevents the kernel from loading drivers with known vulnerabilities. At the time of collection, this blocklist contained 420 unique driver hashes, with VT scores between 0 and 32.

We merge these two sources into a unified dataset and apply a filtering step to retain only drivers that are compatible with modern 64-bit Windows systems and are digitally signed with valid signatures. After filtering, the resulting *Known Vulnerable Drivers Initial Set* (KVDIS) contains 919 drivers, corresponding to 392 distinct driver names (as shown in Table 2).

Driver names were obtained from VirusTotal's `meaningful_name` attribute, which provides a human-readable identifier for each driver sample. We use this attribute to cluster drivers by name, grouping together samples that likely correspond to different versions of the same driver family. As a result, a single name-based cluster may include multiple versions of a driver, including both vulnerable and patched releases. This clustering allows us to reason at the driver-family level while preserving per-sample analysis granularity.

We then apply filtering criteria that were also adopted and validated in our previous work [39]. We use VirusTotal's *Relations* feature to retrieve all execution parents that wrote one of the KVDIS drivers to disk during prior dynamic analysis performed by VirusTotal. From the initial pool of 186,705 execution parents, we apply sampling and clustering to derive two distinct groups based on their VT scores: the first contains 1961 execution parents with VT scores below 6, representing samples that remain largely undetected by mainstream antivirus engines; the second comprises 1436 execution parents with VT scores above 49, corresponding to high-confidence malicious binaries consistently flagged by most detection engines. Specifically, we retained all parents in the second group while, for the first group, we excluded samples larger than 50 MB – as these are commonly installers that do not have driver interactions – and retained a single representative sample from each remaining cluster.

After this selection process, the number of distinct drivers linked to at least one valid execution parent is

reduced to 162, defining the final Known Vulnerable Drivers (KVD) dataset.

As stated above, drivers in the KVDIS dataset are evaluated using the actual IOCTLance framework, while drivers in the KVD dataset, for which an execution parent is available by definition, are additionally processed by our dynamic analysis system.

6.2. Potentially Vulnerable Drivers (PVD) Dataset

The second dataset, named PVD (Potentially Vulnerable Drivers), includes drivers gathered from in-the-wild sources for which no prior vulnerability information exists. This collection is used to test the system's capability to uncover previously undocumented instances of BYOVD exploitation. To this end, we collected signed 64-bit Windows kernel drivers that statically import at least one kernel function from the curated set of *Abusable Imports* defined in [39], defining the Potentially Vulnerable Drivers Initial Set (PVDIS), analogous to the KVDIS dataset. This dataset consists of 5589 drivers, corresponding to 1840 unique driver names (Table 2). Then, we downloaded and clustered their associated execution parents using the same criteria as the KVD dataset, yielding a final PVD dataset of 5712 executables, composed of 3617 low-VT samples and 2095 high-VT samples.

Overall, as summarized in Table 2, our full collection comprises 6,508 symbolically analyzed drivers and 9,109 samples that load a total of 765 name-distinct drivers.

Summary of Notation. We distinguish between *sample-level* and *group-level* units. A *driver sample* denotes a specific driver binary identified by its SHA-256 hash. A *driver group* denotes a set of samples sharing the same `meaningful_name` in VirusTotal, which we use as a proxy for a driver family. We use *KVDIS* and *PVDIS* to denote the initial driver-only collections used for static analysis, and *KVD* and *PVD* to denote the subsets for which at least one execution parent is available, enabling dynamic analysis as well.

7. Static Analysis Design and Evaluation

This section presents the design and evaluation of our static analysis component, which analyzes driver binaries, as shown in Figure 2. We build on *IOCTLance* [32], which primarily targets memory-safety vulnerabilities by tracking user-controlled data from IOCTL buffers. However, BYOVD attacks may also exploit legitimate but insufficiently protected kernel functionality without relying on memory corruption. We therefore extend *IOCTLance* with detectors for additional BYOVD-relevant exploitation primitives and misuse patterns.

7.1. Extending IOCTLance

Our extensions follow a conservative capability-discovery approach: the static component identifies privileged functionality reachable from user-controlled inputs along feasible symbolic paths, rather than proving full exploitability. Because symbolic execution remains subject to path explosion, timeouts, and environment-modeling limitations,

Table 3

We report IOCTLance coverage across individual drivers and driver groups (clustered by name), indicating how many drivers with at least one *IOCTLance finding* (as it is called by IOCTLance) were successfully detected compared to the previous version of IOCTLance.

Dataset	Drivers	Previous IOCTLance Detections [†]		Actual IOCTLance Detections [‡]		Δ Coverage	
		By driver hash*	By driver name	By driver hash*	By driver name	By driver hash*	By driver name
KVDIS	919	663 (72.1%)	291 (74.2%)	663 (72.1%)	292 (74.5%)	0 (0.0%)	+1 (+0.3%)
PVDIS	5589	1181 (21.1%)	368 (20.0%)	1296 (23.2%)	412 (22.4%)	+115 (+9.7%)	+44 (+12.0%)

[†] IOCTLance commit add3e8c

[‡] IOCTLance commit fe08d36

* Drivers with at least one observed behavior

static findings are interpreted as candidate BYOVD-relevant capabilities requiring manual or dynamic validation.

Some suspicious behaviors defined in Section 5 were already supported by *IOCTLance* and remain unchanged, including Privileged User Handle Leakage (**B2**), Arbitrary Memory Writes (**B8**), Arbitrary MSR Writes (**B9**), Arbitrary Port I/O (**B10**), Tainted Physical Memory Mapping (**B11**), and Tainted Kernel Arguments (**B12**).

We extended *IOCTLance* to detect additional suspicious behaviors, specifically Protected Service Termination (**B1**) and Remote Handle Closure (**B7**), which were previously unsupported and represent relevant BYOVD misuse patterns. These behaviors introduce challenges beyond traditional memory-safety analysis. The main one is that they do not rely on direct memory corruption but on kernel handle semantics and API call sequences. As a result, detecting them requires modeling handle propagation, structured inputs, and kernel object interactions, which cannot be captured by memory-safety-focused analysis alone. To address this, we extended symbolic execution to track handle-related data flows and propagate taint across kernel APIs involved in handle acquisition and closure. This allows the analysis to correlate IOCTL-controlled inputs with security-sensitive handle operations.

Detection of Unsafe RtlQueryRegistryValues Calls (**B14**) is newly implemented. This behavior was derived from our observation of malware abusing the legitimate Microsoft termdd.sys driver to disable Code Integrity enforcement, which motivated its formal modeling. Because exploitability depends on registry state available only at runtime, the detector relies on a conservative pattern-based approximation in static analysis. By explicitly modeling this misuse pattern, we identified additional drivers affected by similar vulnerabilities, some of which are discussed in Section 10.

Finally, the remaining suspicious behaviors – Unsafe Pool Allocation (**B3**), Dynamic Kernel Code Execution (**B4**), Mismatched Memory Mapping (**B5**), Code Integrity Tampering (**B6**), Executable or Writable Memory Remapping (**B13**), and RtlQueryRegistryValues Tampering (**B15**) – were not integrated into the static component, as they strongly depend on runtime memory attributes or execution contexts that cannot be reliably inferred through symbolic execution without significant over-approximation.

7.2. Evaluation of the Static Analyzer

We evaluated the effectiveness of the static analysis component by running *IOCTLance* on all drivers in the KVDIS and PVDIS datasets. The analysis was performed on the INDACO platform, a high-performance computing infrastructure at the University of Milan [54], using the default *IOCTLance* configuration.

IOCTLance is built on top of angr (version 9.2.18) and relies on Claripy (version 9.2.18) for symbolic-expression management and constraint solving. We retained the default solver configuration provided by the framework and did not introduce custom solver heuristics or backend-specific tuning.

The analysis machine was equipped with an Intel Xeon Gold 6326 at 2.3 GHz and 1024 GB of RAM. In our experiments, we use IOCTLance 1 hour timeout for the overall symbolic execution process and 40 seconds for the analysis of each individual I/O control code. Both the overall execution and per-IOCTL timeouts are managed internally by IOCTLance, which records the start time when the analysis of an IOCTL begins and checks the elapsed time at each subsequent execution step. The exploration strategy was Depth-First Search (DFS), which maintains only one active execution state at a time while placing all other states in the deferred stash until the active state either terminates or encounters an error. No loop bounds were enforced during symbolic execution. Additionally, IOCTLance's custom path-explosion detection mechanism was enabled, as it is by default in the framework.

Table 5 summarizes the execution results of IOCTLance on both the KVDIS and PVDIS datasets. The table reports the total number of analyses that could not be completed, together with the corresponding failure causes. In particular, unsuccessful analyses fall into four categories: unsupported drivers, IOCTL handler search timeouts, execution timeouts, and cases where no IOCTL handler could be located.

Notably, 47.4% of PVDIS drivers are classified as unsupported because IOCTLance's current IOCTL-handler discovery procedure requires the driver to import IoCreateDevice. This limitation belongs to the underlying IOCTLance framework; our extensions focus on supporting additional BYOVD-relevant behaviors rather than additional driver architectures. Extending IOCTLance to these drivers is left for future work. Table 6 reports the execution time and peak

Table 4
Observed behaviors for the two dataset

B#	Behavior	KVDIS detected drivers	PVDIS detected drivers
B1	ProtectedServiceTermination [†]	11 (9 distinct)	29 (13 distinct)
B2	PrivilegedUserHandleFromKernelLeak	60 (17 distinct)	241 (70 distinct)
B7	RemoteHandleClosed [†]	16 (1 distinct)	5 (1 distinct)
B8	PotentialArbitraryWrite	281 (124 distinct)	93 (50 distinct)
B9	PotentialArbitraryMsrWrite	284 (99 distinct)	38 (24 distinct)
B10	PotentialArbitraryOutWrite	402 (149 distinct)	121 (66 distinct)
B11	PotentialArbitraryPhyMemMap	231 (133 distinct)	26 (20 distinct)
B12*	PotentialMmMapArgumentsTaint	370 (131 distinct)	140 (60 distinct)
B12*	PotentialObjectNameTaint	25 (18 distinct)	39 (20 distinct)
B12*	PotentialSectionHandleTaint	3 (1 distinct)	3 (2 distinct)
B14	UnsafeRtlQueryRegistryCall [†]	37 (14 distinct)	152 (46 distinct)

[†] Contribution of this work

* B12 has been separated into three distinct supported taints for clarity

memory consumption measured during IOCTLance runs. The reported values were collected using the time utility.

To assess the impact of our enhancements, we compared the results produced by the previous version of *IOCTLance* with those obtained by the current version, incorporating the BYOVD-oriented behaviors. Table 3 summarizes the results of this symbolic evaluation, together with the net improvement achieved by the enhanced version of the tool compared to the previous one. Detection rates are reported both at the individual driver level (identified by SHA-256 hash) and at the cluster level, where drivers are grouped according to their internal names; this distinction accounts for scenarios in which multiple versions of the same driver coexist, and only some of them are vulnerable.

Overall, the updated analyzer achieves higher detection rates under both evaluation criteria, demonstrating an improved ability to identify drivers that expose BYOVD-relevant capabilities across distinct binaries and driver families. For the KVDIS dataset, where the vast majority of drivers already match at least one *vulnerability* (as defined by *IOCTLance*), no significant difference is observed. In contrast, the PVDIS dataset shows a more pronounced improvement: 115 additional drivers (44 when name clusters are considered) now have at least one *vulnerability* (again, according to the *IOCTLance* definition). In particular, Protected Service Termination (B1) and especially Unsafe RtlQueryRegistryValues Calls (B14) emerge as widespread suspicious behaviors that are not fully captured by prior static approaches and therefore warrant further investigation. Conversely, Remote Handle Closure (B7) was observed only in driver families already present in KVDIS. Table 4 reports the occurrences of behaviors across the two datasets.

7.3. False Positives

IOCTLance reports potentially vulnerable driver paths associated with specific IOCTL codes. However, determining whether a reported path corresponds to an actual vulnerability, an intentional use of a potentially dangerous primitive, or a benign use case ultimately requires manual validation by an analyst.

False positives may arise primarily from two causes: (I) the implementation of *IOCTLance* does not model, or may miss, kernel-level mechanisms such as exception handlers, validation routines, or guard functions that constrain or otherwise protect tainted input data; (II) constraints applied to tainted data are not sufficiently reflected in the analysis results.

To estimate the prevalence and causes of false positives, we manually evaluated a stratified random sample of *IOCTLance* findings from PVDIS, sampling independently within each behavior type. In total, we manually inspected 100 *IOCTLance* findings and identified 39 false positives, as reported in Table 7. Since this evaluation relies on the manual inspection of the analyzed drivers, the reported classification may be limited by our analysis and understanding of the drivers' behavior. In particular, we classify a reported path as a true positive when manual inspection indicates that it provides a plausibly exploitable primitive or enables a security-relevant operation. This classification reflects an analyst assessment of the reported path and should not be interpreted as definitive proof of exploitability. Consistent with our threat model, we assume that an attacker abusing a vulnerable driver already possesses administrative privileges within the guest machine. Therefore, checks performed by the driver to verify membership in the Administrators group or the presence of administrative privileges are assumed to be successfully satisfied by the attacker.

Manual inspection revealed three main sources of disagreement between static findings and analyst assessment. First, some drivers enforce additional access-control or validation checks that are not fully modeled by *IOCTLance*, including restrictions on the process interacting with the driver. Second, some reported primitives are constrained by hardcoded values, partial taints, or hardware-specific mappings, limiting their security relevance. Finally, in some cases *IOCTLance* correctly identifies a potentially unsafe

Table 5

IOCTLance execution statistics. The tables report the total number of analyses that certainly failed (due to a missing output file) and partial analyses (where an output file is present, but the results may be incomplete), along with the specific reason responsible for each analysis failure.

Dataset	Driver	Failed analyses				Incomplete analyses		
		Count	Unsupported drivers [†]	No IOCTL handler detected	Execution timeout	Count	No IOCTL handler detected	Execution timeout
KVDIS	919	84 (9.1%)	73 (7.9%)	9 (1.0%)	2 (0.2%)	135 (14.7%)	60 (6.5%)	75 (8.2%)
PVDIS	5589	2698 (48.3%)	2651 (47.4%)	33 (0.6%)	14 (0.3%)	1124 (20.1%)	626 (11.2%)	498 (8.9%)

[†] IOCTLance does not handle drivers that do not import `IoCreateDevice`

Table 6

IOCTLance execution time and memory statistics. The tables report the minimum, maximum, and mean execution times (in seconds), as well as the maximum memory consumption.

Dataset	Execution time				Maximum memory			
	Min	Max	Avg.	Median	Min	Max	Avg.	Median
KVDIS	3.6 s	3597.9 s	329.7 s	87.8 s	231 MB	81156 MB	1933.4 MB	810 MB
PVDIS	1.4 s	3598.3 s	300.7 s	10.0 s	120 MB	481971 MB	1427.3 MB	369 MB

pattern, but concrete driver-specific semantics or memory layouts prevent it from providing a practically exploitable primitive. These cases illustrate the expected overapproximation of capability-oriented static analysis.

7.4. False Negatives

Static non-detections may arise for different reasons. A driver may expose a vulnerable behavior that is not modeled by our current taxonomy or detection logic, or symbolic execution may fail to fully analyze the driver due to scalability limitations, complex control flow, path explosion, timeouts, or environment-dependent state.

Table 8 summarizes the main reasons why 256 driver samples in the KVDIS set have no behavior reported by IOCTLance. Unlike the execution statistics in Table 5, this analysis considers the detection outcome rather than the analysis status: it includes all KVDIS drivers for which IOCTLance reported no behavior, including failed analyses, partial analyses that produced no findings before termination, and completed analyses with no detections.

The Unsupported category corresponds to drivers that are not supported by the static tool and matches the count reported in Table 5. IOCTL handler not found covers cases where the IOCTL handler address could not be identified, while Timed-out before output denotes drivers for which no output file was produced before the analysis timed out. No IOCTL codes detected refers to cases where IOCTLance identified the handler but did not explore any IOCTL codes. Finally, All IOCTL codes timed-out covers drivers for which the exploration of every identified IOCTL timed out.

We did not manually investigate cases for which the absence of a detection was already associated with an explicit analysis failure, such as unsupported drivers, failure to identify the IOCTL handler, or symbolic-execution timeouts.

We instead focused the manual analysis on the remaining cases for which the absence of a detection could not be directly explained by these failure conditions. For example, the vulnerable logic may be implemented in handlers such as `IRP_MJ_READ` or `IRP_MJ_WRITE`, which are outside the scope of IOCTLance's current analysis.

The 82 samples in the *Other limitations* category correspond to 19 name-based driver groups and our analysis revealed several recurring causes of these non-detections. Some vulnerabilities depend on driver state that is not represented by IOCTLance's default symbolic environment; for instance, initialization-dependent values stored in the `.data` section may prevent the exploration of a vulnerable IOCTL. Other cases involve vulnerability classes that are not currently modeled by IOCTLance, such as race conditions. We also observed cases in which an output was produced, but symbolic execution timed out before exploring the vulnerable IOCTL. Finally, for a few driver groups, public information was insufficient to associate the analyzed binary with a specific vulnerability, and notably some groups contained both vulnerable and apparently non-vulnerable versions. Thus, these non-detections cannot all be interpreted as confirmed false negatives.

The following two cases illustrate examples of failure modes:

- `SSPORT.sys` – The driver is vulnerable a global-buffer overflow caused by an unchecked `strncpy`. This vulnerability does not match any of the behaviors currently modeled by IOCTLance and is therefore missed independently of symbolic path exploration.
- `LMInfo.sys` – The driver is vulnerable to a race condition that allows an arbitrary handle to be duplicated from the `SYSTEM` process. Since this vulnerability does not

Table 7

False-positive analysis of IOCTLance findings across behavior types. Samples were randomly selected within each behavior class and manually inspected. TP/FP labels reflect analyst assessment rather than definitive proof of exploitability. Per-behavior and aggregate precision values refer to the manually inspected cases.

B#	Behavior	# of detected driver groups	Manually inspected cases	TP	FP	Precision
B1	ProtectedServiceTermination	13	11	7	4	63.6%
B2	PrivilegedUserHandleFromKernelLeak	70	13	1	12	7.7%
B7	RemoteHandleClosed	1	1	0	1	0%
B8	PotentialArbitraryWrite	50	16	7	9	43.8%
B9	PotentialArbitraryMsrWrite	24	8	7	1	87.5%
B10	PotentialArbitraryOutWrite	66	14	12	2	85.7%
B11	PotentialArbitraryPhyMemMap	20	7	3	4	50%
B12*	PotentialMmMapArgumentsTaint	60	14	13	1	92.9%
B12*	PotentialObjectNameTaint	20	4	3	1	75%
B12*	PotentialSectionHandleTaint	2	2	2	0	100%
B14	UnsafeRtlQueryRegistryCall	46	10	6	4	60%
Total		412 [†]	100	61	39	61%

[†] The row totals differ from the total distinct names because some drivers appear in multiple classes.

* B12 has been separated into three distinct supported taints for clarity

Table 8

Breakdown of causes associated with KVDIS samples for which IOCTLance reports no behavior. The table distinguishes explicit analysis failures from other limitations associated with behaviors not captured by the current taxonomy or by IOCTLance.

Category	# of drivers	# of distinct names
Unsupported	73 (28.5%)	51 (43.2%)
IOCTL handler not found	69 (27.0%)	33 (28.0%)
Timed-out before output	2 (0.8%)	2 (1.7%)
No IOCTL codes detected	12 (4.7%)	8 (6.8%)
All IOCTL codes timed-out	18 (7.0%)	7 (5.9%)
Other limitations	82 (32.0%)	19 (16.1%)
Total	256	118 [†]

[†] The row totals differ from the total distinct names because some drivers appear in multiple classes.

correspond to any of the behaviors currently modeled by IOCTLance, it remains undetected regardless of symbolic path exploration.

8. Dynamic Analysis Design and Evaluation

This section presents the design and evaluation of our dynamic analysis component, shown in Figure 3. We build on *kernelmon*, the DRAKVUF-based plugin introduced in our previous work [30] to trace kernel-level execution and detect BYOVD behaviors. The previous implementation primarily relied on API-level tracing to monitor driver loading, IOCTL communication, and security-sensitive kernel operations. While effective at reconstructing BYOVD behavior chains, it could miss exploitation primitives requiring fine-grained inspection of memory writes or kernel-function parameters.

We therefore extend *kernelmon* with detectors for these BYOVD-relevant runtime primitives.

8.1. Behavior Modeling

In Section 5 (*Observable Suspicious Behaviors*), we defined the taxonomy of behaviors that the system must detect. Behaviors **B1-B7** were already implemented in *kernelmon* in our previous work and remain architecturally unchanged. These include high-impact primitives such as protected process termination, code integrity tampering, and dynamic code execution.

In this work, we extended our sandbox to cover behaviors **B8-B15**. Extending the dynamic analyzer required addressing two fundamental design challenges. First, many exploitation primitives cannot be inferred solely from API invocations and instead require taint-detection logic. Second, naïvely introducing instruction-level monitoring or full memory tracing would incur prohibitive performance overhead, undermining sandbox scalability.

To balance detection accuracy and performance, we adopt a *selective instrumentation strategy*. Rather than tracing all kernel instructions or monitoring all memory accesses, the current *kernelmon* version activates fine-grained monitoring only in security-relevant contexts. In particular, deeper inspection is scoped to driver execution and IOCTL-handling paths, and hypervisor-level breakpoints are placed only on identified memory regions of interest. This avoids global instrumentation while preserving visibility into exploitation-relevant operations.

8.2. New Observable Behaviors

Mismatched Memory Mapping (**B5**) has been updated with a bug fix that reduces false classifications caused by unhandled remapping patterns. Behaviors **B8** through **B12** have been implemented to align the dynamic analyzer with the detection philosophy of *IOCTLance*. However, instead of

performing full dynamic taint tracking – which would introduce substantial overhead – we employ lightweight heuristic inference over IOCTL input buffers. By placing selective hooks on critical kernel functions and tracing only memory instructions executed during IOCTL handling, *kernelmon* detects when user-controlled data propagates to privileged kernel arguments or to memory addresses subject to write operations. This targeted approach enables dynamic identification of exploitation primitives, such as arbitrary writes, MSR manipulation, and I/O port abuse, while ensuring that instrumentation costs remain proportional to driver interaction intensity rather than overall system activity. Additionally, `Unsafe RtlQueryRegistryValues` calls (**B14**) have been integrated into *kernelmon*, mirroring the detection logic implemented in *IOCTLance* while also validating actual misuse at runtime. Unlike its static counterpart, the dynamic version confirms whether vulnerable invocation patterns are exercised under concrete execution conditions.

8.3. Runtime-Dependent Behaviors

Behaviors **B13** and **B15** strongly depend on runtime memory attributes and are designed to complement **B5** and **B14**, respectively. These behaviors require hypervisor-level breakpoints and inspection of page permissions to detect execution in previously non-executable regions or abuse of registry-based primitives. Crucially, breakpoints are deployed only on memory regions dynamically identified as suspicious (e.g., writable-executable or remapped pages), avoiding blanket memory monitoring. Unlike purely configuration-based detections, these mechanisms provide evidence of actual runtime exploitation, increasing confidence in detected behaviors while limiting over-approximation.

8.4. Evaluation

All dynamic-analysis 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 at 2.12 GHz, 64 GB of RAM, and a 512 GB NVMe SSD, providing sufficient resources for large-scale dynamic experimentation. Our analysis system leveraged the *kernelmon* plugin within the DRAKVUF sandbox to trace kernel-level operations without modifying the guest system.

To assess the effectiveness of the *kernelmon* plugin, we adopted the same experimental methodology introduced in our previous study, enabling a direct comparison between the previous and current dynamic analyzers. As observed in the earlier evaluation and summarized in Table 10, a substantial portion of the samples (47.0% and 75.5% for the KVD and PVD sets, respectively) did not exhibit driver-loading behavior, thereby preventing further runtime analysis.

In our prior work, 56.3% of high-VT KVD driver groups for which no behavior was detected were attributed to the absence of instrumentation for memory-based exploitation primitives. The extensions introduced in this work explicitly address this limitation by incorporating instruction-level tracing and taint-aware heuristics. As reported in Table 9, detections for high-VT distinct KVD driver groups increase

to 71.9%, while for low-VT KVD drivers, the rate improves from 30.8% to 57.1%. These gains demonstrate that the newly implemented behaviors (**B8-B15**) successfully capture exploitation patterns that were previously undetected. For the PVD dataset, the refined implementation of *MismatchedMemoryMapping* reduces over-approximation and mitigates false positives introduced by unhandled remapping patterns. Overall, the current dynamic component improves both detection coverage and confidence in observed behaviors, particularly for memory- and permission-based exploitation scenarios that require concrete runtime validation. These results confirm that the extensions to *kernelmon* effectively address prior coverage limitations and significantly strengthen the dynamic analysis component. Table 11 shows how dynamic analysis coverage varies when considering only the loaded driver groups versus the driver groups associated with potential execution parents.

To assess the sensitivity of dynamic coverage to the number and diversity of available execution parents, we performed a post-hoc analysis over the executed samples. For each driver group exhibiting at least one behavior, we randomly sampled increasing fractions of its associated parents (10%, 25%, 50%, and 75%) and measured the fraction of unique behaviors recovered with respect to the complete parent set. Each sampling step was repeated 100 times to remove dependencies on parent ordering. On average, 10%, 25%, 50%, and 75% of the parents recovered 71.9%, 84.2%, 91.2%, and 97.9% of the behaviors for KVD, and 67.4%, 81.1%, 91.6%, and 97.3% for PVD, respectively. These results indicate diminishing behavioral returns as additional parents are considered. However, the observed saturation does not imply behavioral completeness: since the set of triggerable behaviors of a driver is unknown a priori, any additional parent may still exercise a previously unseen IOCTL or runtime condition. Therefore, our pipeline analyzes all available execution parents rather than using empirical saturation as a stopping criterion. Parent diversity is partially controlled by our dataset construction strategy. As described in Section 5, low-VT execution parents are clustered and representative samples are retained from the resulting clusters, reducing redundant executions while preserving heterogeneous parent families. Thus, the observed diminishing returns should be interpreted with respect to the diverse set of execution parents available in our dataset rather than as evidence that the complete behavioral space of a driver has been explored.

8.5. False Positives and False Negatives

Our dynamic analysis component reports behavioral evidence that may indicate whether a vulnerable driver has been abused by a malware sample. Consistent with traditional sandbox-based malware analysis, the system focuses on observing and reporting security-relevant runtime behaviors while leaving final detection decisions and sample classification to downstream tools or human analysts.

Table 9

We report sandbox coverage by driver groups, highlighting how many drivers had loaded and showed observable behavior during execution. As reported in Section 8.2, the detection of **B5** has been updated with a bug fix that reduces false classifications caused by previously unhandled remapping patterns, resulting in a lower number of detected behaviors in the PVD set *compared to the old kernelmon version*.

Dataset	Previous kernelmon version				Actual kernelmon version				Δ coverage on observed behaviors
	Driver Groups	Loaded	With an observed behavior	Without an observed behavior	Driver Groups	Loaded	With an observed behavior	Without an observed behavior	
KVD	162	106 (65.4%)	44 (41.5%)	62 (58.5%)	162	106 (65.4%)	67 (63.2%)	39 (36.8%)	+23 (+52.3%)
KVD _{VT<6}	139	91 (65.5%)	28 (30.8%)	63 (69.2%)	139	91 (65.5%)	52 (57.1%)	39 (42.9%)	+24 (+85.7%)
KVD _{VT>49}	95	64 (67.4%)	28 (43.8%)	36 (56.2%)	95	64 (67.4%)	46 (71.9%)	18 (28.1%)	+18 (+64.3%)
PVD	611	118 (19.3%)	48 (40.7%)	70 (59.3%)	603	116 (19.2%)	37 (31.9%)	79 (68.1%)	-11 (-22.9%)
PVD _{VT<6}	524	97 (18.5%)	38 (39.2%)	59 (60.9%)	517	92 (17.8%)	27 (29.3%)	65 (70.7%)	-11 (-28.9%)
PVD _{VT>49}	318	56 (17.6%)	19 (33.9%)	37 (66.0%)	315	57 (18.1%)	23 (40.4%)	34 (59.6%)	+4 (+21.1%)

Table 10

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	3397	1961 (57.7%)	1436 (42.3%)	1802	1006 (51.3% †)	796 (55.4% †)	1010	472 (46.9% ‡)	538 (67.6% ‡)
PVD	5712	3617 (63.3%)	2095 (36.7%)	1401	861 (23.8% †)	540 (25.8% †)	340	183 (21.3% ‡)	157 (29.1% ‡)

† Percentage relative to the corresponding *executed samples* count

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

Table 11

We show how dynamic analysis coverage varies when considering only the loaded driver groups versus the driver groups associated with potential execution parents.

Dataset	#	Loaded		Observed		
		#	Perc	#	Perc	ExecPa Perc
KVD	162	106	65.4%	67	63.2%	41.4%
KVD _{VT<6}	139	91	65.5%	52	57.1%	37.4%
KVD _{VT>49}	95	64	67.4%	46	71.9%	48.4%
PVD	603	116	19.2%	37	31.9%	6.1%
PVD _{VT<6}	517	92	17.8%	27	29.3%	5.2%
PVD _{VT>49}	315	57	18.1%	23	40.4%	7.3%

An alert is generated only when the complete sequence of kernel events corresponding to a behavior in our taxonomy is observed. For certain behaviors labeled as *Potential*, the system additionally applies lightweight inference heuristics to approximate exploitability under runtime conditions. Unlike our previous implementation, this extended design may introduce limited **false positives** due to heuristic reasoning. Nevertheless, determining whether an observed behavior represents an actual vulnerability, intentional abuse, or a benign use case still requires manual reconstruction of the full attack chain. This validation step is inherently complex and requires significant domain expertise.

False negatives may arise in three primary scenarios:

(i) when a sample exhibits a new vulnerability pattern not covered by our taxonomy, (ii) due to implementation errors in hook placement or event-reconstruction logic, or (iii)

when the relevant code path is not triggered within the sandbox execution window. The latter reflects an intrinsic limitation of execution-based analysis and further motivates integration with static techniques.

9. Hybrid Analysis

While Sections 7 and 8 evaluate the static and dynamic components independently, this section quantifies the overlap and divergence between behaviors identified by *IOCT-Lance* (static analysis) and *kernelmon* (dynamic analysis). As highlighted in Figure 4, the input sample must be an executable file, not a driver, because it must be run.

9.1. Formalization of Hybrid Detection Results

Table 12 and Table 13 summarize the overlap and divergence between the static and dynamic analyzes for drivers that were successfully monitored during sandbox execution (see Table 9 and Table 10 for the corresponding runtime statistics).

Let $\mathcal{B} = \{b_1, b_2, \dots, b_m\}$ denote the set of observable behaviors defined in Section 5 and let $\mathcal{S}$ denote the set of executed samples. For each executed sample $s \in \mathcal{S}$, the dynamic analysis produces

$$DR(s) = (D_s, B_s),$$

where D_s is the set of drivers loaded during the execution of s , and $B_s(d) \subseteq \mathcal{B}$ denotes the set of behaviors dynamically observed for driver d .

Table 12

Hybrid detections for drivers loaded and observed during samples execution (refer to Table 9 for the number of loaded drivers in the KVD group).

B#	Behavior	Dynamic Component	Static Component	Hybrid Detections	Δ Coverage Over Dynamic Component		Δ Coverage Over Static Component	
B1	ProtectedServiceTermination	10	7	15	+5	(+50.0%)	+8	(+114.3%)
B2	PrivilegedUserHandleFromKernelLeak	1	10	10	+9	(+900.0%)	0	(0.0%)
B7	RemoteHandleClosed	1	1	1	0	(0.0%)	0	(0.0%)
B8	PotentialArbitraryWrite	21	31	33	+12	(+57.1%)	+2	(+6.5%)
B9	PotentialArbitraryMsWrite	1	36	36	+35	(+3500.0%)	0	(0.0%)
B10	PotentialArbitraryOutWrite	24	57	57	+33	(+137.5%)	0	(0.0%)
B11	PotentialArbitraryPhyMemMap	12	21	21	+9	(+75.0%)	0	(0.0%)
B12*	PotentialMmMapArgumentsTaint	29	47	47	+18	(+62.1%)	0	(0.0%)
B12*	PotentialObjectNameTaint	7	9	16	+9	(+128.6%)	+7	(+77.8%)
B12*	PotentialSectionHandleTaint	0	0	0	0	(0.0%)	0	(0.0%)
B14	UnsafeRtlQueryRegistryCall	2	5	6	+4	(+200.0%)	+1	(+20.0%)

* B12 has been separated into three distinct supported taints for clarity

Table 13

Hybrid detections for drivers loaded and observed during samples execution (refer to Table 9 for the number of loaded drivers in the PVD group).

B#	Behavior	Dynamic Component	Static Component	Hybrid Detections	Δ Coverage Over Dynamic Component		Δ Coverage Over Static Component	
B1	ProtectedServiceTermination	6	4	8	+2	(+33.3%)	+4	(+100.0%)
B2	PrivilegedUserHandleFromKernelLeak	1	7	8	+7	(+700.0%)	+1	(+14.3%)
B7	RemoteHandleClosed	0	0	0	0	(0.0%)	0	(0.0%)
B8	PotentialArbitraryWrite	6	8	13	+7	(+116.7%)	+5	(+62.5%)
B9	PotentialArbitraryMsWrite	1	8	9	+8	(+800.0%)	+1	(+12.5%)
B10	PotentialArbitraryOutWrite	6	11	13	+7	(+116.7%)	+2	(+18.2%)
B11	PotentialArbitraryPhyMemMap	0	1	1	+1	(0.0%)	0	(0.0%)
B12*	PotentialMmMapArgumentsTaint	9	15	17	+8	(+88.9%)	+2	(+13.3%)
B12*	PotentialObjectNameTaint	3	6	8	+5	(+166.7%)	+2	(+33.3%)
B12*	PotentialSectionHandleTaint	0	1	1	+1	(0.0%)	0	(0.0%)
B14	UnsafeRtlQueryRegistryCall	8	5	8	0	(0.0%)	+3	(+60.0%)

* B12 has been separated into three distinct supported taints for clarity

Similarly, the static analysis associates with each driver d the set

$$SR(d) = B_d,$$

where $B_d \subseteq \mathcal{B}$ contains the behaviors identified through symbolic execution.

Since dynamic observations are only available for loaded drivers, we restrict the comparison to the set

$$L = \bigcup_{s \in S} D_s.$$

For each behavior $b \in \mathcal{B}$ we define

$$Dyn_b = \{d \in L \mid \exists s \in S, d \in D_s \wedge b \in B_s(d)\},$$

and

$$Sta_b = \{d \in L \mid b \in B_d\}.$$

The values reported in Tables 12 and 13 are obtained from the cardinalities of

$$|Dyn_b|, \quad |Sta_b|, \quad |Dyn_b \cup Sta_b|,$$

together with the exclusive detections

$$Sta_b \setminus Dyn_b, \quad Dyn_b \setminus Sta_b,$$

While the previous set-based formulation defines the quantitative metrics used to compare static and dynamic detections, we further formalize how evidence from the two components is combined for the same behavior. We classify the source of evidence as:

$$C(s, d, b) = \begin{cases} \text{JOINT,} & \text{if } b \in B_d \wedge b \in B_s(d), \\ \text{STA,} & \text{if } b \in B_d \wedge b \notin B_s(d), \\ \text{DYN,} & \text{if } b \notin B_d \wedge b \in B_s(d), \\ \text{NONE,} & \text{if } b \notin B_d \wedge b \notin B_s(d), \end{cases}$$

where

- **JOINT** denotes a behavior detected by both analysis components ($Sta_b \cap Dyn_b$),
- **STA** a static-only detection ($Sta_b \setminus Dyn_b$),
- **DYN** a dynamic-only detection, and ($Dyn_b \setminus Sta_b$)
- **NONE** the absence of a detection in both analyses.

$C(s, d, b)$ characterizes the provenance and agreement of the available evidence rather than directly establishing end-to-end exploitability.

Specifically, static detection indicates that the driver exposes a potentially exploitable capability or unsafe pattern along a feasible symbolic path. In the absence of a corresponding dynamic detection, we therefore interpret the finding as *potentially exploitable*, but not as evidence that the behavior has been exercised or that exploitation has been demonstrated. Dynamic detection provides concrete runtime proof that behavior b is exercised during execution; however, whether such an observation represents actual abuse or only the runtime exercise of an exploitation primitive depends on the semantics of the behavior in question. Consequently, a joint detection strengthens the evidence that the statically identified code path is reachable and concretely exercised but does not automatically imply successful misuse.

This distinction is necessary because the behaviors in our taxonomy have different security semantics. For behaviors such as **B1**, **B2**, and **B7**, the dynamically observed behavior itself corresponds to the security-relevant action (a dynamic **B1** detection indicates that the process-termination behavior was concretely exercised). In contrast, behaviors labeled as *Potential* describe exploitation primitives or unsafe conditions: their dynamic observation confirms that the corresponding primitive or unsafe pattern is exercised at runtime, but does not necessarily demonstrate actual abuse.

B14 and **B15** provide a representative example. **B14** identifies a potentially unsafe `RtlQueryRegistryValues` invocation and can be detected by both static and dynamic analysis. Thus, a **JOINT B14** finding indicates that the unsafe pattern identified as feasible by static analysis is also exercised during concrete execution. However, this does not, by itself, demonstrate that the pattern is successfully exploited. Concrete runtime tampering associated with this pattern is captured separately by **B15**, which is observable only through dynamic analysis.

We therefore deliberately avoid assigning a globally *high-confidence* BYOVD label solely from the static-dynamic combination.

9.2. Hybrid Analysis on KVD Dataset

The KVD dataset provides a controlled baseline, as it consists of drivers already known to be vulnerable. From a manual analysis, the majority of discrepancies found in Table 12 between static and dynamic detections can be attributed to execution constraints rather than modeling inconsistencies. In particular, the *Static-Component* category contributes substantially for two main reasons: (i) as discussed in Section 8, a sample may load a vulnerable driver

but fail to trigger a specific vulnerable behavior at runtime; (ii) dynamic detection relies on concrete execution conditions, which may not capture all statically reachable attack surfaces.

At the same time, the two analyzes are complementary and highlight different vulnerabilities. In some cases, *kernelmon* detects behaviors not identified statically, reflecting runtime-dependent exploitation semantics or execution-specific effects, while in other cases, static analysis performs better according to the motivations provided earlier. For example, for **B9** (`PotentialArbitraryMsrWrite`), only 1 driver group is detected by both analyzes, while 35 corresponds to statically detected behaviors that cannot be dynamically confirmed.

9.3. Hybrid Analysis on PVD Dataset

A similar pattern emerges in the PVD dataset, which consists of drivers not previously classified as vulnerable. Static analysis continues to make a substantial contribution relative to dynamic analysis, consistent with the observations from the KVD results. For instance, for **B8** (`PotentialArbitraryWrite`), only 1 driver group is detected by both analyzes, while the majority (7 drivers) falls into the *Static-only* category.

9.4. Hybrid Analysis Results Discussion

The distribution observed in Tables 12 and 13 highlights the complementary visibility of the two techniques. The `IObitUnlocker` case (see Section 10) illustrates a runtime-dependent exploitation pattern not fully captured by the static component but observable under concrete sandbox execution. Conversely, other cases such as drivers exposing arbitrary-write primitives illustrate the opposite scenario: static analysis can identify latent capabilities that may not be exercised during a particular execution, while dynamic analysis provides evidence when the corresponding behavior is concretely triggered. More generally, static analysis explores feasible driver paths independently of runtime triggers, whereas dynamic analysis is limited to behaviors exercised during execution. Because symbolic-execution paths do not map one-to-one to dynamic behavior traces, we quantify this complementarity at the behavior and driver-group level rather than by directly comparing path counts. The two components, therefore, expose different and complementary regions of the BYOVD attack surface.

10. Real World Case Studies

In this section, we present three representative examples of newly identified vulnerable drivers, each uncovered through a distinct approach: static analysis only, dynamic analysis only, and the combination of both. As of this writing, all of them can be loaded and exploited on the latest version of Windows 11 (with Virtualization-Based Security (VBS) features disabled). Their hashes are reported in Table 15.

btfilter.sys (CVE-2026-24076) – Arbitrary Write Using `RtlQueryRegistryValues`. Due to the absence of execution

parents, no dynamic analysis was performed on this driver. Our new version of IOCTLance uncovered two versions of the same Qualcomm driver that contain a buffer-overflow flaw rooted in a type-confusion bug within a function called by the `DriverEntry` routine. This bug enables an attacker to trigger an arbitrary write in kernel memory, making these drivers viable candidates for BYOVD-style exploitation. Successful exploitation can disable Code Integrity by modifying the `ci!g_CiOptions` variable, thereby allowing the execution of unsigned drivers.

```

1  __int64 sub_14001E000()
2  {
3      ...
4      // Only relevant code displayed
5      QueryTable[0].Flags = 32;
6      QueryTable[0].Name = L"DownloadFirmwareFlags";
7      QueryTable[0].EntryContext = &dw_140016038; //
      ↳ Initialized to 0x10007
8      QueryTable[0].DefaultType = 0x4000000;
9      QueryTable[0].QueryRoutine = 0;
10     QueryTable[14].Name = L"FWLogEnabled";
11     QueryTable[14].EntryContext = &dw_140016084;
12     QueryTable[14].DefaultType = 0x4000000;
13     QueryTable[14].Flags = 32;
14     QueryTable[14].QueryRoutine = 0;
15     RtlQueryRegistryValues(RTL_REGISTRY_ABSOLUTE |
      ↳ RTL_REGISTRY_OPTIONAL, Path, &QueryTable, 0, 0);
16     ...
17 }
```

Listing 2: `btfilter.sys` vulnerable pseudocode

Listing 2 shows vulnerable pseudocode from the driver, where the `RTL_QUERY_REGISTRY_DIRECT` flag is used without the safe `RTL_QUERY_REGISTRY_TYPECHECK` counterpart. The memory region at `&dw_140016038` – intended to store the default value and the actual one if present in the registry 7 – can be overflowed by setting a value of type `REG_BINARY`, allowing an attacker to overwrite up to `0x10007` bytes. The overflow can reach and modify the memory region at `&dw_140016084` 11, enabling simulation of a `UNICODE_STRING` structure and the abuse of the `FWLogEnabled` value to write arbitrary data into a tainted string buffer whose pointer is controlled by the attacker.

IObitUnlocker.sys – Protected Process Termination. Dynamic analysis identified the *ProtectedServiceTermination* behavior, whereas static analysis failed to capture the taint-propagation logic and therefore reported no vulnerabilities in the driver. However, process termination does not occur through a direct tainted invocation of `ZwTerminateProcess`. Instead, termination emerges as a side effect of file-unlocking logic that indirectly leads to privileged kernel operations [18].

Listing 3 presents part of the driver's termination functionality. The `sub_1574C` routine performs process termination while using only a weak safeguard to avoid killing critical system processes. The driver invokes this routine

```

1  __int64 __fastcall sub_1574C(HANDLE ProcessId,
      ↳ PUNICODE_STRING ProcessImageName, _DWORD *a3)
2  {
3      // Checks if the process's name (from ProcessImageName) is
      ↳ not in the following listing
4      // to continue execution of the function
5      // [
6      //  L"IObitUnlocker.exe", "System", "csrss.exe",
7      //  "lsass.exe", "lsm.exe", "smss.exe",
8      //  "System Idle Process", "wininit.exe",
9      //  "winlogon.exe", "userinit.exe"
10     // ]
11
12     HANDLE ProcessHandle = 0;
13     NTSTATUS Result;
14     Result = OpenProcess(&ProcessHandle, ProcessId, 0);
15     Result = ZwTerminateProcess(ProcessHandle, 0);
16     DbgPrint("ZwTerminateProcess: 0x%08x", Result);
17     ZwClose(ProcessHandle);
18 }
```

Listing 3: `IObitUnlocker.sys` termination feature

to terminate processes that hold handles to files requiring unlocking.

iseflashx64.sys – Arbitrary Write Using `RtlQueryRegistryValues`. Both the static and dynamic analysis tools reported the *UnsafeRtlQueryRegistryCall* (B14) behavior. Our hybrid system uncovered one Insyde Software Corp. driver that contains a buffer-overflow flaw rooted in a type-confusion bug within a function called by the `DriverEntry` routine. This defect enables an attacker to trigger an arbitrary write in kernel memory, making the driver a viable candidate for BYOVD-style exploitation. Successful exploitation can disable Code Integrity by modifying the `ci!g_CiOptions` variable, thereby allowing the execution of unsigned drivers.

Listing 4 shows vulnerable pseudocode from the driver. Similarly to the previous case, the `RTL_QUERY_REGISTRY_DIRECT` flag is used without the safe `RTL_QUERY_REGISTRY_TYPECHECK` counterpart. The memory region at `&dw_1E880` – meant to hold a zero value and the actual value if present in the registry 14 – can be overflowed by assigning a value of type `REG_SZ`. This forces the `CMOSData` context bytes to become non-zero 20, thereby enabling a `REG_BINARY` overflow. This condition can be exploited to simulate a `UNICODE_STRING` structure and abuse the `CMOSOffset1` value to write arbitrary data into a tainted string buffer whose pointer is controlled by the attacker.

11. Limitations

Execution Time and VT Score Thresholds

As in many malware-analysis studies, selecting appropriate execution-time limits and VirusTotal (VT) score thresholds remains an inherently open and orthogonal challenge. Rather than performing exhaustive parameter tuning, we adopted widely accepted thresholds and execution windows consistent with prior work [57, 27]. Our objective is

```

1  int __fastcall sub_11B20(PUNICODE_STRING LogPath)
2  {
3      struct _RTL_QUERY_REGISTRY_TABLE QueryTable[10];
4      int v6 = 0;
5      PoolWithTag = ExAllocatePoolWithTag(PagedPool, 0x1002u,
6      0x206B6444u);
7      LogPath->Buffer = PoolWithTag;
8      LogPath->Length = 4098;
9      LogPath->MaximumLength = 4098;
10     memset(PoolWithTag, 0, 0x1002u);
11     memset(QueryTable, 0, sizeof(QueryTable));
12
13     QueryTable[3].Name = L"OpenClosePerLog";
14     QueryTable[3].EntryContext = &dword_1E880;
15     QueryTable[3].DefaultData = &v6;
16     QueryTable[3].Flags = 32;
17     QueryTable[3].DefaultType = 4;
18     QueryTable[3].DefaultLength = 4;
19     QueryTable[6].Name = L"CMOSData";
20     QueryTable[6].EntryContext = &dword_1E88C;
21     QueryTable[6].DefaultData = &v6;
22     QueryTable[7].Name = L"CMOSOffset1";
23     QueryTable[7].EntryContext = &byte_1E890;
24     QueryTable[6].Flags = 32;
25     QueryTable[6].DefaultType = 4;
26     QueryTable[6].DefaultLength = 4;
27     QueryTable[7].Flags = 32;
28     QueryTable[7].DefaultData = &v6;
29     QueryTable[7].DefaultType = 4;
30     QueryTable[7].DefaultLength = 4;
31
32     ...
33     LODWORD(PoolWithTag) = RtlQueryRegistryValues(
34         0,
35         L"\\Registry\\Machine\\Software\\Insyde\\InsydeFlash",
36         QueryTable,
37         0,
38         0);
39     ...
40 }

```

Listing 4: iscf1ash.sys vulnerable pseudocode

not to optimize these parameters in isolation but to maximize behavioral coverage under realistic and reproducible experimental conditions. Using community-established settings allows us to focus on evaluating detection effectiveness across a broad corpus while preserving comparability with existing literature.

Improvements of Dynamic Coverage

As outlined in Section 3, the intended use of our hybrid pipeline approach – particularly the dynamic sandbox – is to provide insights to analysts examining specific drivers or malware samples. As discussed throughout the paper, the effectiveness of the dynamic analysis is inherently tied to the availability of one or more samples that trigger suspicious driver behaviors. Fuzzing and concolic execution are valuable techniques for generating new inputs; however, our

goal is not to discover new exploits or automatically exercise previously unseen execution paths; rather, it is to provide a safe environment that captures and records abusive driver behaviors during execution.

Detection of Emerging and Unknown BYOVD Behaviors

Our detection logic is grounded in a behavioral taxonomy derived from known BYOVD exploitation patterns. However, the BYOVD ecosystem is evolving, and novel attack techniques may fall outside the current taxonomy. Identifying such behaviors often requires manual reverse engineering and deep semantic analysis of driver logic. Although our system captures comprehensive kernel traces that enable post-hoc investigation, automated discovery of previously unknown exploitation patterns remains an open research challenge. Future extensions could integrate anomaly detection, behavioral clustering, or automated rule generation from high-fidelity traces to reduce manual effort and accelerate the discovery of stealthy or obfuscated BYOVD techniques.

Impact of Virtualization-Based Security

Modern Windows systems increasingly deploy hardening mechanisms under the umbrella of Virtualization-Based Security, including Hypervisor-Enforced Code Integrity (HVCI). In parallel, Microsoft's recent policy changes have phased out trust in the legacy cross-signed driver program, making it progressively more difficult to load unsigned or tampered drivers [38]. Although HVCI is among the most effective defenses against the execution of malicious kernel code, prior research has demonstrated that some of the protections provided by VBS can still be circumvented despite the additional complexity and engineering effort involved [33].

Our current sandbox does not emulate VBS-enabled environments due to limitations in the Xen/Drakvuf support for nested virtualization; however, a simple assessment confirmed that all the drivers listed in Table 15 remain loadable on a fully updated Windows 11 25H2 system with VBS enabled, confirming the importance of having a full (static/dynamic and hybrid) platform to detect vulnerable drivers and their abuse by malicious software.

12. Related Work

12.1. Static Malware Analysis

Static analysis enables systematic reasoning about program control and data flow without execution [20], while symbolic execution explores feasible paths using symbolic inputs. Despite its effectiveness, symbolic execution suffers from path explosion, solver overhead, and limited environment modeling [5], challenges that are particularly relevant to kernel and driver analysis. Prior work has applied static and hybrid techniques to kernel vulnerability detection [4, 3, 24]. POPKORN [21] combines taint tracking and symbolic execution to identify misuse of sensitive kernel APIs, while

IOCTLance explores IOCTL handlers in Windows drivers. Existing approaches, however, primarily target memory-safety issues or require precise environment modeling. In contrast, we use static analysis for BYOVD capability discovery, identifying privileged functionality reachable along feasible driver paths, and complement these findings with dynamic runtime evidence.

12.2. Virtualization-based Malware Analysis

Dynamic malware analysis observes program behavior at runtime [16]. Virtual Machine Introspection (VMI), introduced by Garfinkel et al. [19], enables monitoring from outside the guest while preserving visibility into its internal state [17]; however, bridging the semantic gap requires detailed OS knowledge [15, 40]. Early systems such as Livewire [44] and SecVisor [47] focused on kernel integrity, while Rkprofiler [56] used instruction-level tracing. Later systems improved transparency and observability, including Ether [14], DRAKVUF [30], K-tracer [29], and hybrid monitoring approaches [31, 49, 28]. Dynamic analysis nevertheless remains limited to behaviors triggered during execution and can be affected by environment-aware samples [42]. This is particularly relevant to driver analysis, where vulnerable paths may require specific inputs, runtime conditions, or execution parents. Our work extends VMI-based analysis with behavior-driven BYOVD detection, combining API-level tracing and selective fine-grained memory instrumentation.

12.3. Bring Your Own Vulnerable Driver

BYOVD attacks abuse legitimate signed drivers to perform privileged operations but remain underexplored in academic research. Industry studies have documented real-world abuse and proposed heuristic detection techniques [9, 11, 22], typically based on signatures, static heuristics, or manual analysis. Existing static and dynamic approaches provide complementary but incomplete views: static analysis identifies potentially reachable capabilities, whereas dynamic analysis is restricted to behaviors exercised during execution. Our hybrid pipeline bridges these views by combining symbolic discovery of BYOVD-relevant capabilities with hypervisor-based runtime observation and correlating the resulting evidence. This captures both latent and exercised driver behaviors that may be missed by either technique alone.

13. Conclusions

In this work, we present a hybrid static-dynamic pipeline for identifying BYOVD-relevant vulnerabilities in Windows kernel drivers. By combining an extended version of IOCTLance with an enhanced kernelmon sandbox, we broaden the existing BYOVD taxonomy from 7 to 15 behaviors and improve the detection of both potentially exploitable driver capabilities and security-relevant runtime behaviors. Our evaluation of 6,508 signed kernel drivers and 9,109 execution parents confirms the complementary nature of the two approaches. Static analysis can identify potentially

exploitable capabilities along feasible paths that are not exercised by the available execution parents, while dynamic analysis provides concrete evidence of behaviors triggered during execution. Their combination, therefore, provides a broader characterization of the BYOVD attack surface than either component alone. The practical relevance of the approach is further demonstrated by the identification and responsible disclosure of 13 previously undocumented vulnerable drivers, five of which received CVE assignments. Our dynamic experiments were conducted on Windows 10 without Virtualization-Based Security (VBS) or Hypervisor-Enforced Code Integrity (HVCI). Therefore, the reported dynamic detection rates characterize this experimental setting and should not be interpreted as a measurement of exploitability on VBS/HVCI-enabled Windows systems. Evaluating end-to-end exploitability under these configurations remains an important direction for future work.

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. This work was also funded by government grant managed by the French National Research Agency with reference: ANR-22-PECY-0007

References

- [1] , 2016. Some fun with vintage bugs and driver signing enforcement. https://web.archive.org/web/20160808182709/https://kat.lua.cz/posts/Some_fun_with_vintage_bugs_and_driver_signing_enforcement/. [Online; accessed 11-February-2026].
- [2] Afianian, A., Niksefat, S., Sadeghiyan, B., Baptiste, D., 2019. Malware dynamic analysis evasion techniques: A survey. *ACM Computing Surveys (CSUR)* 52, 1–28.
- [3] Bai, J.J., Li, T., Lu, K., Hu, S.M., 2021. Static detection of unsafe {DMA} accesses in device drivers, in: 30th USENIX Security Symposium (USENIX Security 21), pp. 1629–1645.
- [4] Bai, X., Xing, L., Zheng, M., Qu, F., 2020. idea: Static analysis on the security of apple kernel drivers, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 1185–1202.
- [5] Baldoni, R., Coppa, E., D’elia, D.C., Demetrescu, C., Finocchi, I., 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 1–39.
- [6] Brandt, A., Loman, M., 2020. Living off another land: Ransomware borrows vulnerable driver to remove security software. <https://www.sophos.com/en-us/blog/living-off-another-land-ransomware-borrows-vulnerable-driver-to-remove-security-software>. [Online; accessed 23-July-2026].
- [7] Casvella, L., 2025a. Byovd to the next level (part 1) — exploiting a vulnerable driver (cve-2025-8061). <https://blog.quarkslab.com/exploiting-lenovo-driver-cve-2025-8061.html>. [Online; accessed 23-July-2026].
- [8] Casvella, L., 2025b. Byovd to the next level (part 2) — rootkit like it’s 2025. https://blog.quarkslab.com/exploiting-lenovo-driver-cve-2025-8061_part2.html. [Online; accessed 23-July-2026].
- [9] Checkpoint, 2024. Silent Killers: Unmasking a Large-Scale Legacy Driver Exploitation Campaign. <https://research.checkpoint.com/20>

- 25/large-scale-exploitation-of-legacy-driver/. [Online; accessed 11-February-2026].
- [10] Chester, A., 2022. `g_CiOptions` in a Virtualized World. <https://blog.xpnsec.com/gcioptions-in-a-virtualized-world/>. [Online; accessed 11-February-2026].
 - [11] Climent-Pommeret, A., 2023. Finding and exploiting process killer drivers with LOL for 3000\$. <https://alice.climent.red/posts/process-killer-driver/>. [Online; accessed 11-February-2026].
 - [12] Dafchev, I., 2023. Exploiting wrmsr in vulnerable drivers. <https://idafchev.github.io/blog/wrmsr/>. [Online; accessed 23-July-2026].
 - [13] Dekel, K., 2021. Cve-2021-21551- hundreds of millions of dell computers at risk due to multiple bios driver privilege escalation flaws. <https://www.sentinelone.com/labs/cve-2021-21551-hundreds-of-millions-of-dell-computers-at-risk-due-to-multiple-bios-driver-privilege-escalation-flaws/>. [Online; accessed 23-July-2026].
 - [14] Dinaburg, A., Royal, P., Sharif, M., Lee, W., 2008. Ether: malware analysis via hardware virtualization extensions, in: Proceedings of the 15th ACM Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA. p. 51–62. URL: <https://doi.org/10.1145/1455770.1455779>.
 - [15] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W., 2011. Virtuoso: Narrowing the semantic gap in virtual machine introspection, in: 2011 IEEE symposium on security and privacy, IEEE. pp. 297–312.
 - [16] Egele, M., Scholte, T., Kirda, E., Kruegel, C., 2008. A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. 44. URL: <https://doi.org/10.1145/2089125.2089126>, doi:10.1145/2089125.2089126.
 - [17] Fattori, A., Lanzi, A., Balzarotti, D., Kirda, E., 2015. Hypervisor-based malware protection with accessminer. Computers & Security 52, 33–50. URL: <https://www.sciencedirect.com/science/article/pii/S0167404815000395>, doi:<https://doi.org/10.1016/j.cose.2015.03.007>.
 - [18] Fitzl, C., 2026. CVE-2020-14974 & CVE-2020-14975 - IOBit Unlocker 1.1.2 - Local Privilege Escalation. https://theevilbit.github.io/posts/iobit_unlocker_lpe/. [Online; accessed 11-February-2026].
 - [19] Garfinkel, T., Rosenblum, M., et al., 2003. A virtual machine introspection based architecture for intrusion detection., in: Ndss, San Diego, CA. pp. 191–206.
 - [20] Gosain, A., Sharma, G., 2015. Static analysis: A survey of techniques and tools, in: Intelligent Computing and Applications: Proceedings of the International Conference on ICA, 22-24 December 2014, Springer. pp. 581–591.
 - [21] Gupta, R., Dresel, L.P., Spahn, N., Vigna, G., Kruegel, C., Kim, T., 2022. Popcorn: Popping windows kernel drivers at scale, in: Proceedings of the 38th Annual Computer Security Applications Conference, pp. 854–868.
 - [22] Haruyama, T., 2023. Hunting Vulnerable Kernel Drivers. <https://blogs.vmware.com/security/2023/10/hunting-vulnerable-kernel-drivers.html>. [Online; accessed 11-February-2026].
 - [23] He, J., Sivanrupan, G., Tsankov, P., Vechev, M., 2021. Learning to explore paths for symbolic execution, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 2526–2540.
 - [24] Hu, Y., Wang, W., Hunger, C., Wood, R., Khurshid, S., Tiwari, M., 2021. Achyb: A hybrid analysis approach to detect kernel access control vulnerabilities, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 316–327.
 - [25] Klopsch, A., 2023. 'AuKill' EDR killer malware abuses Process Explorer driver. <https://news.sophos.com/en-us/2023/04/19/aukill-edr-killer-malware-abuses-process-explorer-driver/>. [Online; accessed 11-February-2026].
 - [26] Kolja, 2025. Cve-2025-1865 - local privilege escalation in virtual clonedriver kernel driver. <https://neodyme.io/en/advisories/cve-2025-1865/>. [Online; accessed 23-July-2026].
 - [27] Küchler, A., Mantovani, A., Han, Y., Bilge, L., Balzarotti, D., 2021. Does every second count? time-based evolution of malware behavior in sandboxes, in: NDSS 2021, Network and Distributed Systems Security Symposium, Internet Society.
 - [28] Lanzi, A., Balzarotti, D., Krügel, C., Christodorescu, M., Kirda, E., 2010. Accessminer: using system-centric models for malware protection, in: Conference on Computer and Communications Security. URL: <https://api.semanticscholar.org/CorpusID:6605344>.
 - [29] Lanzi, A., Sharif, M.I., Lee, W., et al., 2009. K-tracer: A system for extracting kernel malware behavior., in: NDSS.
 - [30] Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A., 2014. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system, in: Proceedings of the 30th annual computer security applications conference, pp. 386–395.
 - [31] Leon, R.S., Kiperberg, M., Leon Zabag, A.A., Zaidenberg, N.J., 2021. Hypervisor-assisted dynamic malware analysis. Cybersecurity 4, 1–14.
 - [32] Lin, Z., 2025. IOCTLance. <https://github.com/zeze-zeze/iocltance>. [Online; accessed 11-February-2026].
 - [33] McGarr, C., 2022. Exploit development: No code execution? no problem! Living the age of vbs, hvci, and kernel cfg. <https://connormcgarr.github.io/hvci/>. [Online; accessed 20-July-2026].
 - [34] Michael Haag, Jose Hernandez, N.B., 2025. Living off the land drivers. <https://www.loldrivers.io/>.
 - [35] Microsoft, a. Driver signing. URL: <https://learn.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing>. [Online; accessed 11-February-2026].
 - [36] Microsoft, b. Driver signing policy. URL: <https://learn.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later->. [Online; accessed 11-February-2026].
 - [37] Microsoft, 2024. 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>. [Online; accessed 11-February-2026].
 - [38] Microsoft, 2026. Advancing windows driver security: Removing trust for the cross-signed driver program. <https://techcommunity.microsoft.com/blog/windows-itpro-blog/advancing-windows-driver-security-removing-trust-for-the-cross-signed-driver-pro/4504818>. [Online; accessed 20-July-2026].
 - [39] Monzani, A., Parata, A., Oliveri, A., Aonzo, S., Balzarotti, D., Lanzi, A., et al., 2026. Unveiling byovd threats: Malware's use and abuse of kernel drivers, in: Network and Distributed System Security (NDSS) Symposium 2026, NDSS. pp. 1–19.
 - [40] More, A., Tapaswi, S., 2014. Virtual machine introspection: towards bridging the semantic gap. Journal of Cloud Computing 3, 1–14.
 - [41] Nababkin, D., 2024. 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>. [Online; accessed 11-February-2026].
 - [42] Nappa, A., Papadopoulos, P., Varvello, M., Gomez, D.A., Tapiador, J., Lanzi, A., 2021. Pow-how: An enduring timing side-channel to evade online malware sandboxes, in: European Symposium on Research in Computer Security (ESORICS), Springer. pp. 95–115. doi:10.1007/978-3-030-88418-5_5.
 - [43] Or-Meir, O., Nissim, N., Elovici, Y., Rokach, L., 2019. Dynamic malware analysis in the modern era—a state of the art survey. ACM Computing Surveys (CSUR) 52, 1–48.
 - [44] Petroni, N.L., Hicks, M.W., 2007. Automated detection of persistent kernel control-flow attacks, in: Conference on Computer and Communications Security. URL: <https://api.semanticscholar.org/CorpusID:14014247>.
 - [45] Poslušný, M., 2022. Signed kernel drivers - Unguarded gateway to Windows' core. <https://www.welivesecurity.com/2022/01/11/signed-kernel-drivers-unguarded-gateway-windows-core/>. [Online; accessed 11-February-2026].
 - [46] SentinelOne, 2026. Cve-2020-14979: Evga precision x1 privilege escalation flaw. <https://www.sentinelone.com/vulnerability-dat>

- abase/cve-2020-14979/. [Online; accessed 23-July-2026].
- [47] Seshadri, A., Luk, M., Qu, N., Perrig, A., 2007. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses, in: Symposium on Operating Systems Principles. URL: <https://api.semanticscholar.org/CorpusID:89864>.
- [48] Sharif, M., Lanzi, A., Giffin, J., Lee, W., 2008. Impeding malware analysis using conditional code obfuscation.
- [49] Sharif, M.I., Lee, W., Cui, W., Lanzi, A., 2009. Secure in-vm monitoring using hardware virtualization, in: Proceedings of the 16th ACM Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA. p. 477–487. URL: <https://doi.org/10.1145/1653662.1653720>, doi:10.1145/1653662.1653720.
- [50] Starke, N., 2021. Inputx64.sys windows driver analysis. <https://starkeblog.com/windows/kernel/driver/2021/05/15/inputx64.sys-windows-driver-analysis.html>. [Online; accessed 23-July-2026].
- [51] Tan, J., Jin, H., Zhang, H., Zhang, Y., Chang, D., Liu, X., Zhang, H., 2023. A survey: When moving target defense meets game theory. Computer Science Review 48, 100544. doi:10.1016/j.cosrev.2023.100544.
- [52] Tan, J., Zheng, T., Jin, H., Liu, Y., Zhang, H., Tian, Z., 2025. A strategy-making method for IIoT PLC honeypoint defense against attacks based on the time-delay evolutionary game. IEEE Transactions on Information Forensics and Security 20, 11528–11543. doi:10.1109/TIFS.2025.3601442.
- [53] TrendMicro, 2022. Ransomware actor abuses genshin impact anti-cheat driver to kill antivirus. https://www.trendmicro.com/en_us/research/22/h/ransomware-actor-abuses-genshin-impact-anti-cheat-driver-to-kill-antivirus.html. [Online; accessed 23-July-2026].
- [54] UNIMI, 2026. Infrastruttura di calcolo per il trattamento di DATI Complessi. <https://www.indaco.unimi.it/>. [Online; accessed 11-February-2026].
- [55] Vinopal, J., 2024. Breaking Boundaries: Investigating Vulnerable Drivers and Mitigating Risks. <https://research.checkpoint.com/2024/breaking-boundaries-investigating-vulnerable-drivers-and-mitigating-risks/>. [Online; accessed 11-February-2026].
- [56] Xuan, C., Copeland, J., Beyah, R., 2009. Toward revealing kernel malware behavior in virtual execution environments, in: Kirda, E., Jha, S., Balzarotti, D. (Eds.), Recent Advances in Intrusion Detection, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 304–325.
- [57] Zhu, S., Shi, J., Yang, L., Qin, B., Zhang, Z., Song, L., Wang, G., 2020. Measuring and modeling the label dynamics of online {Anti-Malware} engines, in: 29th USENIX Security Symposium (USENIX Security 20), pp. 2361–2378.

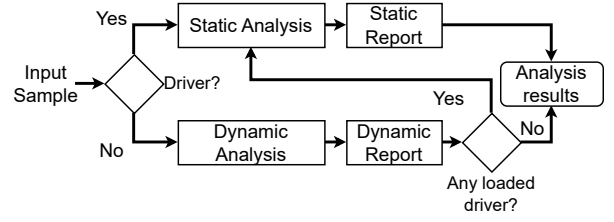


Figure 1: Unified Sample Analysis Pipeline.

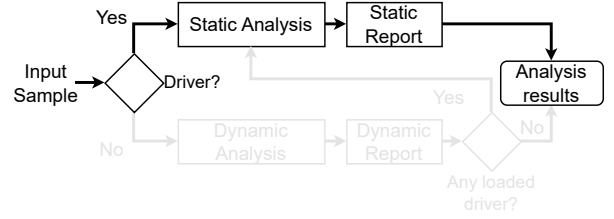


Figure 2: Static analysis pipeline (drivers only).

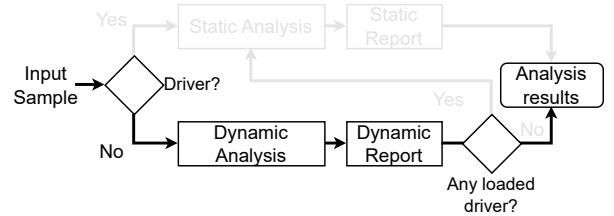


Figure 3: Dynamic analysis pipeline.

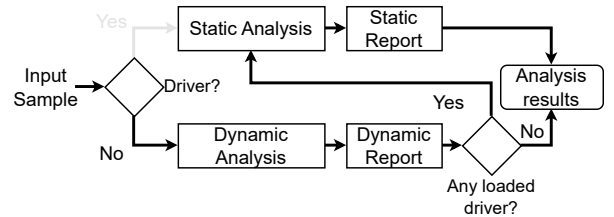


Figure 4: Hybrid analysis pipeline.

Appendix

Table 14: Summary of the observable BYOVD behaviors. Behaviors B1-B7 refine and generalize the taxonomy introduced in our previous work, whereas B8-B15 represent newly introduced behaviors.

Behavior	Detection Goal	Static Features (IOCTLance)	Dynamic Features (kernelmon)	Example
B1	Protected process termination.	Tracks taint propagation from IOCTL inputs through ZwOpenProcess to ZwTerminateProcess, identifying user-controlled process handles.	Correlates process-handle acquisition (ZwOpenProcess, ObReferenceObjectByHandle) with subsequent calls to ZwTerminateProcess.	mhyprot2.sys [53]
B2	Leakage of privileged kernel handles to user mode.	Detects unsafe handle creation/opening paths where privileged handles are returned without OBJ_KERNEL_HANDLE.	Monitors ZwCreate*/ZwOpen* and verifies subsequent user-mode use through ObpReferenceObjectByHandle.	ElbyCDIO.sys [26]
B3	Writable and executable kernel pool allocation.	–	Detects executable nonpaged pool allocations created through ExAllocatePoolWithTag.	WX allocation (discussion in [8])
B4	Execution of dynamically generated kernel code.	–	Monitors execution inside dynamically allocated WX memory and correlates execution with the responsible driver and IOCTL.	LnvMSRIO.sys [8]
B5	Unsafe memory remapping with elevated permissions.	–	Intercepts MmMapIoSpace and MmMapLockedPagesSpecifyCache, comparing requested and original memory permissions.	Code remapping (discussion in [7])
B6	Tampering with Windows Code Integrity.	–	Monitors writes targeting g_CiOptions and other Code Integrity structures.	CI bypass
B7	Closure of handles belonging to another process.	Tracks handle propagation across KeStackAttachProcess and ObCloseHandle.	Correlates remote process attachment with cross-process handle closure.	AuKill [25]
B8	Potential arbitrary kernel memory write.	Identifies write instructions whose destination address symbolically depends on IOCTL-buffer data.	Places breakpoints on write instructions and checks whether the destination address originates from the IOCTL buffer.	gdrv.sys [6]
B9	Potential arbitrary MSR write.	Tracks symbolic dependencies of wrmsr operands.	Intercepts MSR writes at the hypervisor layer and verifies whether operands derive from IOCTL inputs.	DirectIo64.sys [45, 12]
B10	Potential arbitrary port I/O.	Tracks symbolic dependencies of out instruction operands.	Monitors runtime execution of out instructions and correlates operands with IOCTL inputs.	Inpoutx64.sys [50]
B11	Potential physical memory mapping.	Tracks tainted parameters passed to NtOpenSection and NtMapViewOfSection.	Correlates physical-memory mappings functions (NtOpenSection) with subsequent calls to NtMapViewOfSection with tainted parameters.	WinRing0.sys [46]
B12	User-controlled arguments passed to privileged kernel APIs.	Tracks taint propagation toward sensitive API parameters (e.g., to MmMapIoSpace, to ZwMapViewOfSection, OBJECT_ATTRIBUTES fields).	Verifies whether privileged API arguments derive from IOCTL-controlled data.	ElbyCDIO.sys [26]
B13	Execution or writes in suspicious remapped memory.	–	Places breakpoints on remapped regions detected with B5 and monitors execution or writes violating original page permissions.	Memory remapping (discussion in [7])
B14	Unsafe RtlQueryRegistryValues invocation.	Models vulnerable invocations using RTL_QUERY_REGISTRY_DIRECT under attacker-controlled registry configurations.	Detects potential vulnerable invocations of unsafe registry-query patterns.	termdd.sys [1]
B15	Runtime tampering of registry-query structures.	–	Detects concrete malformed RtlpQueryRegistryDirect invocations and modifications of the associated QueryTable during execution.	termdd.sys [1]

Table 15: Reported Drivers

Driver Name	SHA256 (first 6 bytes)	Vuln	Coordinated Disclosure Timeline
HWAUDIO0s2Ec.sys	5abe477517f5 10fb4c4b93e1 13a2a2c57025	B1	[2025-02-12] Vulnerability reported to Huawei PSIRT ¹ [2025-03-07] Huawei responds that the vulnerability has been fixed in 2023 [2025-03-25] Security advisory Huawei-SA-20250325-01-PC-en created [2025-03-27] Huawei informs us that CVE-2023-52972 has already been assigned before our submission
iscflashx64.sys (See Section 10)	c27b8b619910	B14	[2025-09-08] Vulnerability reported to Insyde ² [2025-09-16] Insyde responds that that an updated version of the driver is available since 2016, and that in 2017 they requested Microsoft to prevent the installation and loading of any components signed with their legacy certificate.
virtio_rng.sys	1f0ed057b98e	B14	[2025-09-12] Vulnerability reported to SUSE ³ [2025-10-03] SUSE commits a patch to the vmdp project [2025-10-16] Initial acknowledgment received [2025-10-17] CVE-2025-8412 assigned
btfilter.sys (See Section 10)	415bfd549ce 8544781f315c	B14	[2025-09-08] Vulnerability reported to Qualcomm ⁴ [2025-09-30] Initial acknowledgment received [2026-01-29] CVE-2026-24076 assigned
fekern.sys	009a13eeea05	Bx	[2025-05-28] Vulnerability reported to Trellix via HackerOne ⁵ [2025-07-27] Initial acknowledgment received [2026-02-26] CVE-2025-14963 assigned and fix released
vdk.sys	1ca23e5a2a57 2b5e3d36dcba	B14	[2025-10-02] Vulnerability reported to Veeam via HackerOne ⁶ [2025-11-17] Mitre assigns to us CVE-2025-63750 [2026-03-05] Vulnerability acknowledgment by vendor and fix in validation stages [2026-03-12] CVE-2026-21709 assigned by vendor
tosrfec.sys	01aa894cb84c	B14	[2025-10-23] Vulnerability reported to JPCERT/CC [2025-11-11] Vendor acknowledges vulnerability to JPCERT/CC [2025-12] Updated driver in development by vendor [2026-03-16] Fixed driver for general users available with March Windows Update [2026-04-13] CVE-2026-35553 assigned by JPCERT/CC
-	-	B14	[2025-10-09] Vulnerability reported to vendor [2026-02-11] Vendor still processing the vulnerability report Still no answer from the vendor
-	-	B1	[2025-12-25] Vulnerability reported to MITRE Still no answer from MITRE

B1 Exploited to kill the protected Windows Defender process MsMpEng.exe

B14 Exploited to abuse a vulnerable RtlQueryRegistryValues call and overwrite the g_CiOptions to disable CodeIntegrity

Bx Exploited to dump the memory of lsass.exe when running as a protected process. This behavior is not part of our taxonomy because we observed a specific taint on KeStackAttachProcess for this driver while testing behavior **B7**, which is itself based on this kernel function, as explained in Section 5

¹Huawei Disclosure Guidelines

²Insyde Disclosure Guidelines

³SUSE Disclosure Guidelines

⁴Qualcomm Disclosure Guidelines

⁵Trellix HackerOne Page

⁶Veeam HackerOne Page