ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

COMSEC

Efficient Pipelining for Windows Driver Vulnerability Research

Master Thesis

Author: Tobias Oberdörfer

Tutor: Luca Cappiello

Supervisor: Prof. Dr. Kaveh Razavi

February 2024 to July 2024

Abstract

Within Microsoft Windows, which is ubiquitous in business environments, kernel device drivers are essential high-privilege components that enable communication with hardware. Despite being developed using robust frameworks, these drivers remain susceptible to software bugs. Their elevated access, coupled with Microsoft's reliance on third-party vendors for both their development and security, makes them appealing targets for malicious actors.

Continuous assessment of such drivers is hindered by the lack of a centralized repository, the necessity for a suitable environment for dynamic analysis of each driver, the complexities of static analysis due to diverse frameworks, and the lack of tooling compatibility with the ARM architecture for newer drivers.

The proposed system aims to bridge the gap in the ongoing security evaluation of Windows kernel drivers by continuously collecting and analyzing new drivers through both static and dynamic methods, building upon prior research. To date, over 26'000 drivers have been analyzed, with around 140 undergoing manual verifications, leading to the discovery of 28 vulnerabilities in 14 distinct drivers, including previously unknown (0-day) vulnerabilities in ten of them.

Acknowledgements

I want to express my gratitude to those who have supported and guided me throughout the completion of this thesis.

First and foremost, I am grateful to my advisor, Prof. Dr. Kaveh Razavi, for the opportunity to work under his supervision, his continued guidance, and his expertise. His encouragement and constructive feedback have been pivotal in shaping this work.

I also thank InfoGuard AG for providing the necessary computing resources and licenses. Special appreciation goes to my company supervisor, Luca Cappiello, for his support, including facilitating the possibility of completing this thesis within the company.

Lastly, I acknowledge all the reviewers, particularly Luca Marcelli, Mario Bischof, and Lucas Dodgson, for their valuable feedback on various drafts of this thesis. Their stimulating discussions, encouragement, and insightful suggestions have significantly enhanced the quality of this work.

Contents

Chapter 1

Introduction

1.1 Motivation

The ubiquity of the Windows operating system in business operations necessitates robust security measures to ensure uninterrupted functionality. Microsoft Windows is a complex system in which device drivers play a crucial role by facilitating interactions between applications and hardware components. These drivers, required to access hardware such as GPUs, BIOS, disks, network cards and others, operate with high-level privileges within the kernel. This privileged access and their communication with user space make drivers a prime target for various attacks.

The operating system's security is closely linked to the security of these kernel drivers, for which a significant percentage of Microsoft relies on third-party vendors to develop. Despite using Microsoft's frameworks like Windows Driver Model (WDM) [44], Windows Driver Framework (WDF) [55], and Driver Module Framework (DMF) [43], such software is not immune to bugs. Such vulnerabilities can be exploited to gain elevated privileges, deactivate security products, or even persist in firmware when direct hardware access is provided.

Despite the requirement that third-party drivers have to be signed by Microsoft, Microsoft does not thoroughly verify the code during this process; instead, relying on the vendors to secure their drivers. Additionally, no known third party continuously evaluates all newly released drivers. This results in the state of Windows driver security allowing threat actors to exploit vulnerabilities in signed third-party kernel drivers as part of their attack strategies [76] [58] [30].

A further personal motivation for this research is the opportunity to engage in practical research and collaborate with industry, gaining hands-on experience applicable to my future career while contributing directly to the security landscape of Windows.

Motivation for Industry Collaboration

Collaboration with industry was vital for the success of this thesis. By working closely with industry experts, access to invaluable knowledge, computing and licensing resources was gained that were otherwise difficult to obtain. These resources include access to a Security Operations Center (SOC) team with production environments across diverse industries and an operationally active Red Team with hands-on experience writing Windows exploits.

1.2 Challenges

While ensuring the security of any kernel driver is vital for the Windows operating system, several challenges complicate verifying this security.

A logistical challenge lies in the collection of all Windows drivers. Despite Microsoft signing all drivers, there is no centralized repository. Microsoft also provides a recommended block list of known vulnerable drivers, which is incomplete, making it tricky to identify them. Furthermore,

the continuous creation and publication of new or updated device drivers necessitate ongoing vulnerability assessments.

On the technical side, dynamic analysis is difficult because of the need for the appropriate environment to execute drivers, as kernel drivers typically require a fully functional kernel and their specific devices. Conversely, static analysis is difficult because of the driver complexity and the lack of ARM architecture support for existing tools. The latter is of heightened concern by the recent release of the new Windows 11 ARM devices [8].

Methodologically, optimizing the time of security researchers is critical. Since fixing vulnerabilities is costly and usually requires manual verification, it is imperative to avoid wasting efforts on false positives, which do not enhance security, while also preventing false negatives, which might give a false sense of security.

1.3 Solution

To enhance the current state of Windows driver security, this thesis implemented a system that gathers and analyzes drivers continuously. The system applies tools from previous research while extending one to support a broader range of drivers. The results are then presented in a format that enables security researchers to verify possible vulnerabilities efficiently.

The system continuously collects drivers from multiple sources while checking them against a list of known vulnerable drivers. Where possible, it employs static and dynamic analysis techniques. Specifically, it uses IDA Pro scripting for static analysis and fuzz testing for dynamic analysis. This approach aims to reduce the limitations of both methods. Lastly, the system prioritizes the most likely vulnerable drivers, allowing researchers to focus their time on the highest-value targets.

Using this system, over 46'000 drivers were gathered, and over 27'000 were automatically checked. Of the first 5000 drivers gathered, approximately 140 of those identified as most likely vulnerable were manually checked. This process resulted in identifying 14 unique drivers with 28 vulnerabilities. Four drivers had published vulnerabilities, while ten drivers contained previously unknown (0-days) vulnerabilities that could potentially allow privilege escalation.

The source code and the latest updated list of known vulnerable drivers are released publicly to allow others to contribute to this system or run it themselves. [1]

1.4 Overview

Chapter 2 provides an overview of related work to this thesis. Subsequently, Chapter 3 outlines the necessary background information, including concise summaries of the tools utilized. In Chapter 4, the problem challenges are elaborated upon, followed by the design decisions to address them in the developed system. Chapter 5 details the implementation of the system components, most of which are individually assessed in Chapter 6 besides the evaluation of the complete system. Lastly, Chapter 7 discusses the findings and their implications, concluding with a summary of the entire work in Chapter 8.

[1] `https://github.com/Toroto006/windows-kernel-driver-pipeline`

Chapter 2

Related Work

Vulnerability research is a cornerstone of contemporary security practices, including within the Microsoft Windows environment. Kernel drivers, essential operating system components often interfacing directly with hardware, are a significant attack vector on Windows. Research into vulnerabilities of these drivers is well-established; for instance, projects such as ioctlfuzzer and iofuzz utilized random mutation fuzzing to identify vulnerabilities in these highly privileged programs over a decade ago. Similarly, ioctlbf combined fuzzing with static analysis of the driver under examination.

Microsoft has also contributed to this field with developer tools like IoAttack and its successor, DF-Fuzz. In addition to fuzzing, symbolic execution has been explored in various studies. For instance, "High-Coverage Security Testing for Windows Kernel Drivers" combined symbolic execution with fuzzing, while studies such as ScrewedDrivers and POPKORN uncovered new vulnerabilities using solely symbolic execution. The latest advancement in this area is IOCTLance, which builds on POPKORN but extends its capabilities to identify additional vulnerabilities.

Name of Work	Type	Release Date	Drivers Tested	Constraints
ioctlfuzzer [19]	T	2009	-	No ARM
ioctlbf [32]	T	2012	-	Only WDM, no ARM
High-Coverage Security Testing for Windows Kernel Drivers [60]	P	2012	24	Only WDM, no ARM
iofuzz [15]	T	2014	-	No ARM
ScrewedDrivers [16]	P	2019	"hundreds"	Only WDM
POPKORN [20]	P	2022	212	Only WDM
Hunting Vulnerable Kernel Drivers [22]	T	2023	328	No ARM
IOCTLance [86]	T	2023	328	Only WDM
IoAttack [45] / DF - Fuzz [42]	T	-	-	Source Required

Table 2.1: This table presents an overview of some previous work related to Windows kernel driver vulnerability research, detailing the release date, type of work (tool or paper), the number of drivers tested if mentioned, and specific constraints such as only supporting the old Windows Driver Model (WDM), no ARM drivers, or requirement of driver source code.

Another recent approach was taken by VMware's Threat Analysis Unit (TAU) in their blog post "Hunting Vulnerable Kernel Drivers" [22] using static analysis through decompiler scripting. While TAU also published proof-of-concept exploits for the discovered vulnerabilities,

well-known strategies such as Token Stealing [2], and privilege escalations using Model Specific Registers [84] had been documented much earlier.

As summarized in Table 2.1, all these prior efforts have constraints that limit their applicability to a specific subset of drivers. For instance, Microsoft's DF-Fuzz requires source code, which is typically not accessible to third parties for security evaluations. Furthermore, the studies that mention the number of drivers tested report testing on only a few hundred drivers, whereas the total number of existing drivers reaches the thousands. This thesis aims to develop a pipeline that allows the continuous application of previous research at scale while extending support to the ARM architecture in light of the recent release of Windows 11 for ARM.

Chapter 3

Background

This chapter provides the background required to comprehend this thesis. The initial section (Section 3.1) delineates threat actor categorizations and clarifies their interest in Windows drivers. Section 3.2 contains multiple subsections that progressively introduce Windows-related and thesis-relevant concepts in greater detail. The final two sections, Section 3.3 and Section 3.4, offer background on the tools and projects employed throughout this thesis, both for automated analysis and driver collection.

3.1 Windows Threat Actors

This thesis searches for vulnerabilities attackers exploit after initial access to a Windows operating system, aiming to escalate privileges or bypass security measures. This section discusses industry classifications of threat actors and their specific interest in Windows kernel drivers for this purpose.

Threat Actors and Their Categorization in Industry

Threat actors are individuals or groups that exploit vulnerabilities in systems, including endpoint devices, servers, or cloud infrastructure, to gain unauthorized access, steal sensitive data, disrupt operations, or cause harm to organizations.

In the cybersecurity industry, threat actors are typically categorized based on their motivations and levels of sophistication. IBM provides a publicly accessible categorization [25], whereas the InfoGuard Red Team's categorization, outlined below, is not publicly available in such a format but is very similar. Motivations include espionage, financial gain, political activism, sabotage, and entertainment or learning. The main categories are nation-state adversaries, cybercriminals, insiders, hacktivists, and thrill seekers.

Threat actors' level of sophistication does not have to be correlated to their motivations, although those with similar motivations often exhibit similar levels of sophistication. From a defender's perspective, it is practical to prioritize defence based on the sophistication of the threat rather than solely on their motivation. It is helpful because any adversary would prefer using less sophisticated methods if effective, as more advanced techniques typically require higher resource investment.

The Red Team at InfoGuard AG categorizes threats to a client's overall Windows Domain security posture into four levels, as illustrated in Figure 3.1. Although these levels seem named according to the threat actors' motivations, any threat actor at a higher level is assumed to possess all the capabilities of those at lower levels. Thus, the higher the level, the more sophisticated the threat actor is considered to be.

At the lowest level, script kiddies are assumed to use only publicly known exploits without modification and to employ them on a large scale to discover vulnerable targets. Opportunistic

Figure 3.1: Categorization of threat actor sophistication for InfoGuard Red Team assessments. The colour coding indicates the defence level, where green signifies adequate protection with current defences, and red indicates insufficient protection.

criminals go a step further by setting up open-source attacking infrastructures and might be modifying some known exploits. Organized crime groups reach a higher sophistication level by developing their malware, command-and-control infrastructure, and possibly novel exploits.

At the highest level of sophistication are usually nation-state adversaries, who are presumed to have nearly unlimited resources. These resources might include dedicated research divisions for discovering new vulnerabilities and integrating their exploits into their attack infrastructure.

Why Target Windows Kernel Drivers?

Windows, as an operating system, is part of a broader Windows Active Directory in most industry cases. Such a Windows Domain is the de facto standard for many businesses. Because control over it gives access to all devices within the domain, for attackers, it is often the ultimate target. Initial access vectors into a Windows Domain frequently begin with a single Windows system, which could be a client endpoint compromised through phishing [12] or a server exploited due to vulnerabilities in internet-facing services. Initial access is often obtained with unprivileged user permissions, as normal users should never have administrative access in a corporate setting, and web services should operate under restricted user accounts. Even if administrative access is achieved due to misconfigurations, modern Endpoint Detection and Response (EDR) systems are usually in place to thwart many attack vectors, even with elevated privileges.

Windows Kernel drivers are part of the trusted computing base for the Windows operating system [53] and hold very high privileges. Consequently, these drivers must be developed with security in mind, as they become prime targets for attackers post-initial access. Despite recommended threat modelling concepts and Microsoft's advocacy for security by design [54], exploitable vulnerabilities in drivers continue to surface. Within the "STRIDE" approach to threat modelling (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege) [31] outlined by Microsoft, Information Disclosure and Elevation of privilege are particularly interesting for attackers after initial access.

The reasons for targeting kernel drivers are twofold. First, if initial access is achieved with unprivileged user rights, many critical attack objectives, such as lateral movement, persistence, or causing harm, are not feasible without administrative access. For example, lateral movement may require the ability to proxy through the system, persistence often involves modifying system settings, and executing ransomware would necessitate extensive file access. Therefore,

attackers are highly attracted to exploiting vulnerabilities in kernel drivers that are accessible by unprivileged users and can provide privilege escalation.

Second, even if attackers gain administrative privileges, whether through initial access or other means, most EDR systems can detect malicious tools in user space, regardless of whether they are run as normal or administrative users. This detection capability exists in EDRs because they have kernel-level components. Attackers must employ novel techniques that bypass detection or circumvent the EDR system by incorporating kernel-level components. However, a default-configured Windows operating system only allows loading signed drivers. Thus, attackers face the challenge of either signing a malicious driver, which is not sustainable for most except nation-state adversaries or exploiting vulnerabilities in theoretically benign drivers to achieve the same effects.

3.2 Windows Operating System

This section summarises the key aspects of the Windows operating system that are important to this thesis. It begins by describing the various security levels and then provides an overview of the different types of Windows kernel drivers. Subsequently, the access control mechanisms employed by Microsoft Windows for kernel drivers are discussed, followed by some available kernel APIs and intrinsic to kernel drivers.

3.2.1 Conceptual Process Privilege Levels in Windows

This section outlines the conceptual privilege levels of processes in Windows, which are crucial for understanding the significance of discovering vulnerabilities in Windows kernel drivers for privilege escalation. These levels are only conceptual, as Windows supports a more complex access control system for specific resources that are less relevant to this thesis.

Figure 3.2 illustrates the four conceptual privilege levels: the least privileged at the top and the most privileged at the bottom. The top three operate in user mode and are categorized into sandboxed, normal or non-admin, and admin or service processes. Typically, user applications run with non-admin privileges or are sandboxed, like browsers. Non-admin is the most common starting point for threat actors after compromising a user or device. Processes that need to invoke privileged Windows APIs, such as for changing system settings or passwords, require admin privileges. These higher privileges are granted to a process with administrator account credentials but should be inaccessible to unauthorized processes lacking these credentials. Windows services, including user-space drivers, generally run at this administrator privilege level. Although Microsoft explicitly does not define the boundary between admin privileges in user mode and kernel mode as a security boundary [49], kernel mode provides substantially greater capabilities.

3.2.2 Types of Windows Kernel Drivers

Windows kernel drivers can be broadly divided into Plug-and-Play (PnP) drivers and Non-PnP (Non-PnP) drivers. This section aims to help readers understand the distinctions between these types of drivers, their initiation mechanisms, and the frameworks available for their development.

PnP drivers are designed to manage hardware devices dynamically detected and configured by the operating system. These drivers utilize the PnP manager, a subsystem within the Windows OS that handles the recognition and configuration of hardware components. A key characteristic of PnP drivers is their reliance on their PnP stack; no PnP driver can be instantiated without its full stack.

A notable aspect of PnP drivers is that the `WdfDeviceCreateDeviceInterface` function creates device interfaces for communication. User-space applications can enumerate such device

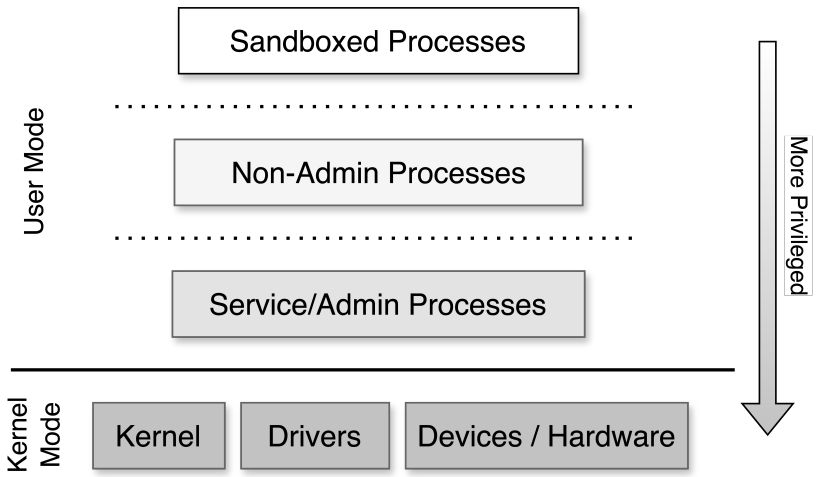


Figure 3.2: Conceptual process privilege levels in Windows. Executing functions in a higher privilege level for an unauthorized process should theoretically be impossible.

interfaces transparently; as such, any driver that employs `WdfDeviceCreateDeviceInterface` must inherently support a PnP stack, reflecting the dynamic and adaptable nature of PnP drivers.

In contrast, Non-PnP drivers are typically associated with software or legacy hardware that does not require dynamic detection and configuration. These drivers are initiated through the service manager, which loads them as services. The absence of a PnP stack in Non-PnP drivers means they do not engage in the automatic hardware detection and configuration processes managed by the PnP manager. Instead, they rely on startup configurations defined by the corresponding system service through the service manager.

Three frameworks assist in creating Windows kernel drivers, each varying in complexity, capability, and suitability for modern driver development practices.

- **Windows Driver Model (WDM)** [44]: The WDM is an outdated framework. Despite its age and complexity, WDM drivers are still operational within the Windows ecosystem. Still, they are not recommended for use in new development due to higher error rates and, hence, vulnerabilities. WDM offers limited support for operating system interactions and state-machine implementation, particularly during power-state transitions.

- **Windows Driver Framework (WDF)** [55]: The Kernel-Mode Driver Framework (KMDF) within WDF enhances WDM by encapsulating essential boilerplate code, thereby reducing the potential for errors and vulnerabilities. KMDF abstracts complex operating system interactions, simplifies development, and promotes secure practices, facilitating the creation of robust drivers.

- **Driver Module Framework (DMF)** [43]: DMF is the newest addition to the Windows driver development frameworks. Built on top of WDF, DMF allows for modularization of functionality to achieve more straightforward and structured drivers. This framework facilitates extracting driver components into distinct modules, promoting code reuse and modular design. Since DMF uses WDF APIs, it functions as a WDF extension. Despite its modularization, DMF does not inherently protect against vulnerabilities within the compiled code. Consequently, DMF drivers are treated and evaluated as WDF drivers in this work.

3.2.3 Possible Interaction Methods for Windows Kernel Drivers

Regardless of the method used to develop a kernel driver, many kernel drivers function as a component within a more extensive system that includes user-space applications. Even if a specific kernel driver does not require direct interaction with a user-space program, it is still improbable that the driver will operate in isolation from other systems beyond the operating system and the hardware/devices it manages. Consequently, most drivers support one or more interaction methods to respond to requests from other drivers or user-space programs. This subsection elucidates six possible methods of communication with Windows kernel drivers, although some are discouraged from use.

- Device I/O Control is the primary mechanism for interacting with Windows kernel drivers. It necessitates the support of the I/O manager (more details later in Section 3.2.3), which manages all user-space calls to `DeviceIoControl`. Drivers create named objects, which serve as the first argument to the I/O control API, either by directly creating a symbolic link with a specific name accessible to user space or by creating a device interface which is also available as a named object. The specifics of how each is accomplished, i.e., which APIs to use, vary slightly across different frameworks. In addition to the handle to a device object and input/output buffers, I/O Control requires an Input Output Control code (more details in Section 3.2.3). This code essentially defines the function that the driver is requested to execute.

- Read or Write Operations to the handle of a device driver operate similarly to Device I/O Control in that the interaction passes through the I/O manager. However, this method is fundamentally simpler because it lacks control codes. Specifically, each read or write operation sends a specific I/O request that either supplies the driver with data or expects to receive data from the driver.

- Windows Management Instrumentation (WMI) provides a standardized framework for managing and instrumenting drivers and devices in a Windows environment. WMI allows drivers to expose management information and operational events that can be consumed by WMI clients, which may include user-space applications or scripts. Drivers register WMI classes and instances with the WMI provider, enabling clients to query and receive notifications about various operational aspects of the driver and the devices it manages. This method facilitates extensive monitoring and management capabilities, allowing administrators and automated systems to interact with the driver for configuration, status checking, and event handling. The use of WMI ensures that drivers can integrate seamlessly into the broader system management infrastructure, enhancing the overall manageability and reliability of the system.

- Memory-mapped I/O is a generally discouraged method for interacting with kernel drivers. In this approach, the caller and the driver must first gain access to the same memory region, typically by invoking a Device I/O Control to the driver to establish this shared memory. Subsequently, the caller and the kernel use this shared memory and some form of access locking to exchange data. This method effectively bypasses the operating system, precluding the OS from managing memory access control, paging, or similar functions.

- Named Pipes are commonly utilized for communication in user space. Drivers can expose or access pipes, but creating them necessitates using undocumented APIs, such as ZwCreateNamedPipeFile/NtCreateNamedPipeFile. These undocumented APIs are subject to change without notice, potentially causing driver failures. In certain cases, opening existing named pipes using ZwCreateFile, i.e., as a consumer from kernel space, may be acceptable. However, employing this method similarly to memory-mapped I/O is inadvisable as it bypasses the operating system and presumes that a driver is a specific consumer

of a user-space program. This assumption is problematic because drivers typically provide services to user-space applications, not the reverse.

- COM Interfaces are also listed here for the sake of completeness. They are used as an interaction method for kernel drivers between Audio Port or miniport drivers and their corresponding port drivers, but not for user-space interaction. An instance of COM Interface usage is Microsoft's printer drivers and their printer plug-ins.

Device Input and Output Control Codes

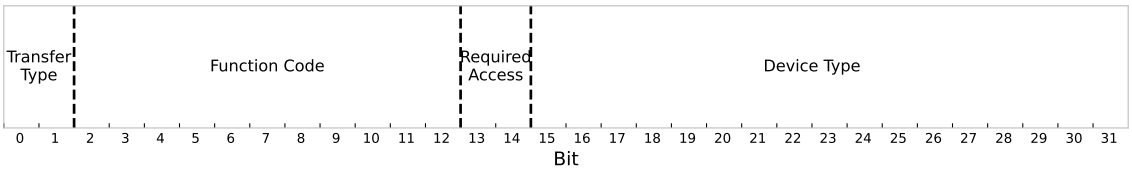


Figure 3.3: Bit field representation of a Windows Input/Output Control Code.

Communication with a kernel driver, whether originating from another driver or a user-space process, often employs the `IoBuildDeviceIoControlRequest` or `DeviceIoControl` functions [41]. In these cases, the sent Device Input and Output Control Code (IOCTL Code) determines the specific functionality requested from the driver. This section serves as a concise reference for the composition of such IOCTL codes, as delineated by Microsoft [40]. Such an IOCTL Code is a 32-bit value comprised of the following bit-fields seen in Figure 3.3

- **Transfer Type** indicates how the I/O Manager provides the driver with the input and output buffers.
- **Function Code** specifies the particular function the driver is expected to execute. Each function code should represent a unique operation.
- **Required Access** bits define the type of access the caller requires to send this IOCTL code to the device handle. The possible access types include:
  - FILE_ANY_ACCESS: Only the device handle itself is required.
  - FILE_READ_DATA: Read access to the device handle is required.
  - FILE_WRITE_DATA: Write access to the device handle is required.
- **Device Type** values below 0x8000 are reserved for Microsoft, whereas third-party device types use values above. The most significant Bit is known as the COMMON bit.

The Windows I/O Manager

This section introduces the I/O manager, an interface that facilitates communication between applications and kernel drivers, and describes its responsibilities in this context. The I/O manager defines a set of standard mandatory driver routines, such as `DriverEntry`, and creates the driver objects for each installed and loaded driver. Additionally, it is responsible for accepting I/O requests, which usually originate from user mode, and creating the corresponding I/O Request Packets (IRPs) for them.

The I/O manager then transfers these interrupt requests as parameters to the appropriate driver's routine responsible for this IRP, along with the address of the created driver object, enabling the driver to respond accordingly. The method by which the I/O manager creates the

IRPs, how the provided input and its length are conveyed to a driver and how the output is returned depends on the I/O request type and the device object flags set by the driver. For read or write requests to the device handle, which are translated into `IRP_MJ_READ` and `IRP_MJ_WRITE` function calls, the device object's flags `DO_BUFFERED_IO` or `DO_DIRECT_IO` determine whether the I/O manager copies the input/output from/to a provided buffer or supplies a reference to the mapped input/output buffers. For device I/O Control requests, the behaviour is dictated by the Transfer Type defined in the IOCTL code of the I/O request [38]. There are four distinct options for any IOCTL code, each being mutually exclusive with exactly one required:

- **Buffered I/O Method**: Typically used for transferring small amounts of data per request in either direction. It is the safest option as the I/O manager initially copies the input data to a temporary buffer and then copies the output data from the same temporary buffer.
- **Direct Input** or **Direct Output**: These methods are sometimes used for higher transfer speeds. The I/O manager copies the input to a temporary buffer and provides a mapping for the user-provided output buffer. Direct Input or Direct Output defines the driver's access rights to the provided mapped output buffer address, read-only or read/write, respectively.
- **Neither**: This is the most insecure option, as the I/O manager provides no system buffers or mappings; it only supplies the driver with the user-mode virtual addresses of the input and output buffers specified in the call.

In conclusion, the I/O manager manages two types of communication with kernel drivers, their setup, and some mandatory driver routines. It is important to note that there are no differences from the I/O manager's perspective, regardless of the framework used to write a driver. Although the definition of the Transfer Type for IOCTL codes can be confusing, the Windows Driver Framework abstracts these details to facilitate easier development.

3.2.4 Access Control for Windows Kernel Drivers

Access control, particularly from user space, is critical to safeguarding the privileged functions that kernel drivers can expose. Within the kernel space, access control is disregarded due to the callers' elevated privilege level; thus, only user-space callers are subject to access control mechanisms. The approach to defining which security principals can access a kernel driver can vary depending on the driver framework used, but it is handled through Access Control Lists (ACLs). The configuration of these ACLs for device objects utilizes the Windows Security Descriptor Definition Language (SDDL) summarized in Section 3.2.4.

There are several methods to secure a device object of a kernel driver. The relevant access controls can be established during driver installation through the setup information (INF) file, set by user-space components using APIs such as `SetKernelObjectSecurity` or configured by the driver during initialization. The first method can be implemented by setting the `HKR Security AddReg` directive for either the device class or the device itself to a valid SDDL string within the INF file. This security string configures the Registry Keys during installation, holding the access control information. At load time, the Plug and Play manager determines the device's initial security descriptor by first checking the relevant registry key. If no entry is found, the device's setup class is checked for a default security descriptor for the device class. In the absence of both, the initial security descriptor is determined by the device type and characteristics, typically defaulting to full access for administrators and read, write, and execute access for all other users for most devices.

The method by which drivers themselves set access controls during initialization depends on the driver framework used:

- For any WDM drivers without a corresponding Security descriptor set in the registry, `IoCreateDeviceSecure` must be used to create a named device with the provided security descriptor. Additionally, WDM drivers have further access control capabilities through the `IoValidateDeviceIoControlAccess` API, which allows for dynamic checks of permitted access and potentially restricts access dynamically after initialization. WDM drivers must also control access to their device namespace if they have one, i.e. if they are accessible by name. Suppose the `FILE_DEVICE_SECURE_OPEN` characteristic is not set during device creation. In that case, files within the driver's namespace can be opened without access control, posing security risks because the device ACL can be bypassed [39].
- For KMDF drivers, the `WdfDeviceInitAssignSDDLString` API is available to set access control during initialization. However, this is again similar to WDM, a last-chance default used if no security descriptor is defined for the device or its device class.

Microsoft provides guidance on controlling device access as part of their driver security checklist [47]. At the same time, the OSR Team offers a more detailed analysis of the nuances of access control for KMDF drivers [63].

In addition to defining ACLs for the driver device handle, each IOCTL code (Section 3.2.3) specifies the access rights, with the `RequiredAccess` bits, that a caller must be granted through the ACL for the I/O manager to forward the request. This is particularly important because opening a device driver without any access rights (i.e., neither read nor write) still allows any IOCTL code defined with `FILE_ANY_ACCESS` to be valid, as the I/O manager sends the IRP for any caller with a valid handle. Therefore, merely being included in the ACL, even with no real access, is sufficient to access potentially privileged functions [61].

All these access controls are specific to obtaining a device handle, which is necessary for sending any read/write or I/O requests. Crucially, they do not apply to other non-standardized methods of interacting with device drivers, such as named pipes. Consequently, these non-standardized methods require additional vigilance in implementing appropriate access controls.

Security Descriptor Definition Language for Device Objects

The Windows Security Descriptor Definition Language (SDDL) is a string-based format for defining security descriptors, which are used to specify the security attributes of objects. In the kernel device drivers' context, SDDL defines the access control policies that determine which users or system components can interact with the driver. Although INF files support the full SDDL, `IoCreateDeviceSecure` only supports the most used subset [52].

D:P(A;;GA;;;SY)(A;;GRGWGX;;;BA)(A;;GR;;;WD)(A;;GR;;;RC)

An SDDL string consists of multiple components that define access control entries (ACEs) for the different security principals. For instance, in the above-given example SDDL string, the local system account (**SY**) has complete access (**GA**) to the driver. In contrast, all built-in administrators (**BA**) have generic read, write, and execute permissions (**GRGWGX**). Everyone else (**WD** = World) including resctricted code (**RC**) have only read permissions (**GR**).

3.2.5 Intrinsics and Windows Kernel Driver APIs

Windows kernel drivers operate at a high privilege level, enabling interaction with both kernel APIs — such as mapping specific physical regions for Memory-Mapped I/O — and hardware itself, for example, to facilitate communication across a physical port with external hardware. Like in standard software development, driver developers have various options depending on the framework and the development objectives. Microsoft provides different libraries that offer APIs for interacting with various kernel components [36], as well as privileged intrinsics that compile

into instructions directly interfacing with specific hardware [48] [56] [37]. This work references a small subset of these APIs and intrinsics, primarily those directly relevant to vulnerable drivers. For completeness, brief definitions of these APIs and intrinsics can be found in Appendix A.5.

3.3 Introduction to Utilized Tools

Portions of this thesis are based on or built upon previous research on automated vulnerability assessment. The following two subsections offer overviews of previously published and utilized tools. These summaries aim to aid readers in understanding the modifications and usage requirements discussed later.

3.3.1 IDAPython Vulnerable Driver Hunting Script

As part of this thesis, modifications were made to the script developed by VMware's Threat Analysis Unit (TAU), which was released alongside the blog post "Hunting Vulnerable Kernel Drivers" [22]. To comprehend the changes applied to the module discussed in Section 5.7, this section summarises the original script.

The previous research employed IDA Pro [67], a powerful yet expensive disassembler and decompiler, particularly utilizing its scripting interface, IDAPython [18], to automatically check a list of drivers for potential vulnerabilities. The created IDAPython script has two operation modes: triage, which functions without the decompiler's user interface, and analysis, which is useful when manually inspecting the driver through the interface. This script is designed to detect potentially vulnerable drivers and assist in the laborious manual validation of them.

It aims to achieve two primary objectives: identifying IOCTL handlers, the entry point of any driver function execution, and tracing execution paths from these handlers to target APIs or instructions while simultaneously resolving any types and propagating meaningful variable names for manual analysis. Although the IDA GUI can find execution paths, this functionality is absent in IDAPython, necessitating TAU's custom implementation.

The method for identifying IOCTL handlers varies depending on the driver type. In WDM drivers, assignments to the `MajorFunction` array member of the `DRIVER_OBJECT` are detected (details about this structure in Appendix A.6), whereas for WDF drivers, identified by an existing call to `WdfVersionBind`, the IOCTL handler is extracted from an argument to the indirect function call `WdfIoQueueCreate`. Extracting the IOCTL handler for WDF drivers is more complex due to version changes in WDF and compilation optimizations altering how the WDF API is compiled into drivers. Additionally, some drivers create function wrappers when calling WDF APIs, altering the final compiled code again.

The IDAPython built-in functionality of traversing a decompiled function as an expression tree in pre-order and post-order is utilized to locate target APIs. Specifically, from any potential IOCTL handler, the expression trees are first traversed in pre-order to fix any IOCTL-related structures, set the types and names of input/output indirect WDF APIs, and recursively propagate function argument names and types into subroutines. During the post-order traversal, paths to any interesting APIs are established when returning up the expression tree.

Windows Driver Framework "Imports"

The WDF framework provides a set of functions for developers. However, these functions are not imported from a library; they are compiled into the driver. Driver developers can utilize these as regular imported functions, but they are called via a function offset table named `WdfFunctions`, as specified in the Windows Driver Framework [35]. An example of this is the `WdfDeviceCreateSymbolicLink` function, illustrated in Figure 3.4a, where the preprocessor macro FORCEINLINE compels the Visual Studio compiler to always inline the table lookup.

```
_Must_inspect_result_
_IRQL_requires_max_(PASSIVE_LEVEL)
NTSTATUS
FORCEINLINE
WdfDeviceCreateSymbolicLink(
    _In_
    WDFDEVICE Device,
    _In_
    PCUNICODE_STRING SymbolicLinkName
    )
{
    return ((PFN_WDFDEVICECREATESYMBOLICLINK) WdfFunctions[WdfDeviceCreateSymbolicLinkTableIndex])
        (WdfDriverGlobals, Device, SymbolicLinkName);
}
```

(a) The `WdfDeviceCreateSymbolicLink` definition from *wdfdevice.h* shows the global function table offset indirect call with inline macro.

```
result = ((__int64 (__fastcall *)(__int64, PVOID, int *))g_WDF_functions.pfnWdfDeviceCreateSymbolicLink)(
           qword_140005E38,
           DeferredContext,
           &v18);
if ( (int)result >= 0 )
{
  memset(&Config, 0, sizeof(Config));
  Config.EvtIoDeviceControl = fn_ioctl_handler_wdf;
  Config.Size = 96;
  Config.PowerManaged = WdfUseDefault;
  Config.DefaultQueue = 1;
  Config.DispatchType = WdfIoQueueDispatchSequential;
  result = ((__int64 (__fastcall *)(__int64, PVOID, WDF_IO_QUEUE_CONFIG *, _QWORD, _QWORD))g_WDF_functions.pfnWdfIoQueueCreate)(
             qword_140005E38,
             DeferredContext,
             &Config,
             0LL,
             0LL);
```

(b) `WdfDeviceCreateSymbolicLink` function IDA decompilation view after script correction for indirect table offset call.

Figure 3.4: Windows Driver Framework functions are compiled as indirect table offset calls. The shown example is with the function `WdfDeviceCreateSymbolicLink`.

This definition has the effect that any compiled and stripped driver contains no references to their initial API names. No references lead to two issues: First, static import analysis cannot identify these functions despite many acting as substitutes for direct kernel imports in WDM. Second, since the IOCTL handler in WDF is defined as an argument to `WdfIoQueueCreate`, one of these indirect functions, initiating propagation requires first identifying this function and its configuration parameter.

Therefore, the original propagation script handled these inlined direct calls for specific cases. To recognize the configuration structure that includes the IOCTL handler function address, the `WdfIoQueueCreate` is first searched, followed by input and output handling functions. It identifies them by verifying any indirect call for a valid offset to the desired function in the global WDF Functions table. Then, it redefines them in the decompiler with the appropriate function header and name. This process results in the much more helpful decompilation view shown in Figure 3.4b.

3.3.2 Intel's kAFL

Numerous fuzz testing [57] frameworks exist, with one of the most prominent implementations in the style of American Fuzzy Lop (AFL) being Intel's kAFL [70]. kAFL is a hardware-assisted feedback fuzzer designed for compiled programs and entire virtual machines operating on Intel's x86 architectures.

The current implementation is based on the kAFL [71] paper, with seven additional papers proposing modifications. Of most use to this thesis are the REDQUEEN [1] and Nyx [72] modifications. kAFL is the name of the entire fuzzing framework, whereas REDQUEEN is an optional feature within kAFL that improves the fuzzing feedback system by exploiting input-to-state correspondence. Input-to-state correspondence works because parts of the input are often

directly influenced by the program state during execution. REDQUEEN enables the fuzzer to bypass hardcoded checks such as magic numbers and checksums more efficiently, which is helpful for driver IOCTL codes and input/output size checks in Windows drivers. Although the Nyx extension to kAFL was initially developed to optimize coverage-guided hypervisor fuzzing, its fast snapshot restoration mechanism also enhances the speed of fuzzing kernel components within virtual machines.

The current kAFL implementation requires hardware support, specifically Intel Virtualization Technology (VT), Intel Page Modification Logging (PML), and Intel Processor Trace (PT). Intel VT facilitates efficient virtualization of the target, Intel PML accelerates the resetting of the target state between fuzzing iterations, and the processor traces extracted from virtual machines via Intel PT guide the fuzzer towards new inputs with improved code coverage. These processor features are available in nearly all modern Intel CPUs, particularly those from the Skylake generation or later.

kAFL adheres to the core principles of AFL: running the target as rapidly as possible with varied inputs to maximize code coverage and detect errors. Feedback for each input is provided as achieved program coverage, which guides the mutation engine towards inputs that yield higher or novel code coverage. Between iterations, the state of the target can either be restored, eliminating side effects from previous iterations but requiring more time, or maintained, in which case side effects might only become apparent during later iterations. Depending on the target, a virtual machine is launched or QEMU's direct kernel boot feature is utilized. Only the former is supported for Windows. A fuzzing iteration involves coordination between a fuzzing agent within the VM and the modified Linux host kernel serving as the hypervisor.

The fuzzing agent is responsible for two tasks. First, it sets up the virtual machine in a starting state from which subsequent fuzzing iterations will commence; it then communicates this to the hypervisor. This setup requires that any targets be loaded, running, and accessible to the fuzzing agent at boot. The second task of the fuzzing agent is to read the fuzzing payload after the hypervisor resets the virtual machine to the starting state during each iteration. To achieve these two tasks, the fuzzing agent has a fuzzing harness, which is the code within the fuzzing agent that sets up and manages the execution of the target program in a controlled manner, or in the case of the Windows driver, the interaction with them.

3.4 Sources for Drivers

This section briefly introduces the three primary sources used to download drivers for later analysis.

VirusTotal

VirusTotal is a widely used online service that aggregates the results of various antivirus engines, website scanners, and other tools to analyze files and URLs for potential malware and other security threats. The key features provided by VirusTotal are the following:

- **File and URL Scanning**: Users can upload files to be scanned by a wide range of antivirus engines and website scanners, providing a robust assessment of potential threats.
- **VirusTotal Intelligence**: This feature allows users to perform advanced searches across the entire VirusTotal database, using various filter criteria such as file type, signature status, filename patterns including wildcards, and specific Windows executable imports.
- **API Access**: VirusTotal offers an API enabling automated service interaction, which allows for programmatic submissions, retrieval of scan results, intelligence queries and integration into other security tools.

Although VirusTotal's basic file and URL scanning service is free, advanced features like VirusTotal Intelligence and API access require a premium subscription to be usable.

Microsoft Update Catalog

The Microsoft Update Catalog is described "as a one-stop location for finding Microsoft software updates, drivers, and hotfixes" [50]. This free-to-use resource is accessible via a website that provides search functionality to locate and download archives containing updates across different Windows versions. The types of updates supported include important, recommended, and optional updates.

Users can search using various criteria, including driver model, manufacturer, hardware ID, update title, description, applicable products, classifications, and knowledge base articles. However, the catalog does not offer a direct listing of all existing updates or drivers, and search results are limited to the first 1000 entries, requiring specific and combined search terms to narrow the scope and obtain the desired results. In addition to individual downloads, the Microsoft Update Catalog allows updates to be imported into corporate network management applications like Windows Server Update Services (WSUS), System Center Essentials (SCE), and System Center Configuration Manager (SCCM) to facilitate efficient distribution within enterprise environments.

InfoGuard's Cyber Defence Center

The InfoGuard Cyber Defence Center (CDC) provides a comprehensive range of cybersecurity services. These services include outsourced cloud and managed security services and a Security Operations Center (SOC) service. The CDC employs over 80 highly qualified Cyber Security Experts and Analysts who ensure clients' security across the DACH region (Germany, Austria, Switzerland).

The SOC services are based on sensors deployed across the client networks to detect anomalies, investigate suspicious activities, and respond to threats. The value of cooperating with the CDC lies in the drivers they can provide that are present in real-world business networks.

Chapter 4

Design

This chapter elaborates on the challenges of identifying vulnerable kernel drivers (Section 4.1). Subsequently, Section 4.2 discusses the design decisions implemented to address these challenges through an automated pipeline. Finally, Section 4.3 details specific design choices for various modules within the pipeline.

4.1 Challenges in Identifying Vulnerable Kernel Drivers

Identifying vulnerabilities in Windows kernel drivers involves a multistep process, but no single, well-established methodology exists. Each possible step presents numerous challenges, broadly categorized into technical, logistical, and methodological issues. This section delineates the steps and primary challenges encountered throughout this process.

Logistical Challenges

To commence any analysis, obtaining valid driver files is essential. Accessing a wide range of drivers poses a significant logistical challenge for two main reasons. First, a Windows operating system can have Microsoft's original and third-party kernel drivers. Collecting pre-installed drivers from multiple systems is insufficient to search for vulnerabilities or, at the least, very inefficient. Second, because nearly all drivers are proprietary, there is no single central source for the compiled drivers, and their source code cannot be easily accessed because they are proprietary. Lastly, given new or updated kernel drivers are continuously published, any similar checking for vulnerabilities must be done continuously.

Technical Challenges

Upon securing valid driver files, two general types of analysis are possible: static and dynamic. Static analysis involves only examining the driver binaries and metadata, such as their signatures, due to the unavailability of the proprietary source code for thorough source code analysis. Any dynamic analysis, such as behavioural, dynamic, or concolic analysis, on the other hand, requires executing the driver and observing its execution within its environment. These analysis methods present several technical challenges.

- **Environment Setup**: Setting up an isolated environment, typically within virtual machines, is crucial to avoid host system crashes or non-reproducibility if the drivers are executed. Kernel drivers operate at a high privilege level, interacting closely with the operating system and hardware, meaning any intended or unintended side-effect might affect the whole system. These side effects necessitate a controlled and reproducible environment to ensure accurate and safe analysis.

- **Runtime Dependencies**: Executing drivers often requires specific runtime dependencies, especially for drivers designed to interact with particular hardware, such as BIOS drivers. These drivers may not load or may function with reduced capabilities if the expected hardware is unavailable, which is often the case for third-party drivers within virtual machines.
- **Architecture Compatibility**: Another technical challenge is the architecture for which the driver was compiled. While the source code may be the same across different architectures, tools and systems that work with binary files for one architecture may not easily support another or do so in a limited capacity.

Although static binary analysis is generally only affected by the compatibility of the tools with different architectures, it is inherently more complex due to the absence of a working environment to observe driver interactions. Conversely, dynamic analysis is complicated to fully automate in a generalized manner because of the vast array of possible environment setups and hardware configurations. These together make creating a one-size-fits-all automated analysis solution for kernel drivers challenging.

Methodological Challenges

Identifying vulnerabilities in kernel drivers also involves methodological challenges, primarily related to the risk of false positives and false negatives. A single false negative of a severe vulnerability can be enough for malicious actors to compromise the entire operating system. On the other hand, addressing false positives is also crucial, as fixing and publishing a new driver version is time-consuming and costly. Consequently, only definite vulnerabilities are typically addressed promptly. To mitigate these risks, the final step in vulnerability identification often involves developing a proof-of-concept exploit to verify the vulnerability conclusively.

Another methodological challenge is determining which version of a driver to analyze. Given the analysis process's human resource-intensive nature, examining every driver's version is usually impractical. Most drivers exist in numerous versions, even with the same target architecture, so adopting a strategic approach to selection or prioritization is necessary. This challenge is compounded by the fact that vulnerabilities can vary between versions.

Although no well-established methodology exists, any tool or system trying to automate or help with this process for successfully identifying vulnerabilities in Windows kernel drivers must tackle these challenges. The subsequent section will detail the specific methodologies employed to overcome these hurdles in an automated way wherever possible, ensuring a comprehensive and practical initial vulnerability assessment.

4.2 Pipeline Design for Identifying Vulnerable Kernel Drivers

Given that the primary objective is to automate the identification of vulnerable kernel drivers as much as possible, the design was necessarily a pipeline. This section outlines the design and rationale behind the different decisions for this pipeline.

Collection and Storage of Files

Driver files must be gathered and stored effectively to conduct any further analysis. Since no single central authority for acquiring drivers exists, this logistical challenge is addressed modularly to allow for incremental integration of different driver sources. This modularity is achieved by defining a storage interface, which specifies the minimal required information a driver has to have.

Since not all sources might provide only Windows kernel drivers but also executables, libraries, or similar files, an initial step is necessary to determine whether the retrieved file is a Windows kernel driver. The storage interface requires only the file and its origin to centralize this decision process and keep the unified interface simple. Depending on the source, such an origin can be anything from a unique identifier to an origin location description.

The actual storage of files is similarly important. Submitted files, potentially hundreds of megabytes, are stored on disk to avoid inefficiencies associated with database file blobs. Files are named using a sufficiently complex and unique hash. Metadata about the stored files, including the origin filename, file size, type, architecture (if applicable), and multiple hash types, is saved in a database, referencing the file on disk via its unique hash. This metadata facilitates quick identification of duplicate files, which is essential when downloading files is costly. Given the simple storage interface, the small identification step to gather the mentioned metadata is part of the later analysis stage.

To minimize storage footprint, especially since not all files are relevant, a cleanup process removes non-useful files from the disk while retaining their metadata, ensuring that no analysis is repeated if the same file is submitted again. Non-useful files are any that do not contain drivers, such as text files, given that we are only interested in Windows kernel drivers.

Analysis of Files

After collecting a set of driver files, the next step is their analysis. Given the complexity of Windows kernel drivers, this analysis involves multiple steps, most based on existing tools with modifications and categorized into static or dynamic analysis:

- **File Identification (Static)** determines whether a file is a Windows kernel driver. It is crucial as it completes the metadata mentioned above for each file, including calculating hashes and extracting the file type. This module ensures that only valid drivers proceed further in the pipeline.
- **Known Vulnerable Drivers (Static)** cross-references the collected drivers with a list of known vulnerable drivers. By identifying already known vulnerable drivers, this module helps reduce false negatives. It further ensures that any driver previously identified as vulnerable is flagged and handled accordingly.
- **Static Driver Information (Static)** is a task shared between two modules to extract static information from each driver. It looks at the kernel APIs the driver imports and specific strings within the driver. This information can indicate whether a driver has exploitable functionality or if it allows user-space applications to communicate with it, which is essential for many types of exploits.
- **File Extraction (Dynamic)** is not a pure analysis module because after recognizing supported archive formats, such as Cabinet Files (CAB), it extracts their contents and resubmits these files into the pipeline. This process ensures that all potentially valuable files are analyzed, even if initially embedded within other files.
- **Driver Signature Status (Static)** checks verify the digital signature of Windows kernel drivers. Since newer versions of Windows only load signed drivers, this module helps eliminate drivers that cannot be executed in normal circumstances, thereby reducing false positives. This ensures that resources focus on drivers that could be loaded and potentially exploited.
- **Driver API Reachability (Static)** is the most sophisticated static analysis module in the pipeline. It examines if any interesting APIs for exploitability used in the drivers have

user-controlled arguments. If such paths exist, the driver is possibly exploitable, thereby prioritizing it for further analysis.

- **Driver Fuzzing (Dynamic)** This module performs automatic fuzzing of drivers. Given the resource-intensive nature of fuzzing, this module operates a priority queue that ranks drivers based on the results of other analysis modules. This prioritization helps optimize resource use, focusing efforts on the most promising candidates and minimizing time spent on unlikely targets.

Microservices Architecture

Due to the diverse system requirements of these modules, ranging from only running on Windows to Ubuntu kernel version dependencies for fuzzing and the required modularity, a single monolithic system is impractical. Consequently, the pipeline uses a microservices architecture, allowing different modules to operate on varied hardware and operating systems. This architecture also facilitates environment setup and compatibility for dynamic analysis without imposing these requirements on the entire pipeline.

Addressing the methodological challenge of multiple driver versions is still problematic, as it involves recognizing and prioritizing similar drivers or analyzing all versions. Any system applying the former would have to calculate a similarity matrix between all drivers, which involves the inherent complexity of similarity matrix calculations while still not providing any meaningful way to decide which of the similar drivers to check. As such, the current pipeline design runs all modules across all possible drivers, disregarding similarities for now. This approach ensures that even minor changes, which might introduce vulnerabilities, are not overlooked by design.

Integrating Security Researchers

The pipeline design aims to minimize false negatives by analyzing all drivers and rating them by likelihood of vulnerability rather than discarding any outright. Still, it cannot definitively classify drivers as true positives. Definite classification requires a proof-of-concept exploit. While fuzzing crashes provide concrete inputs that reach bugs, these bugs are not always exploitable; hence, these inputs do not count as proof-of-concept. Therefore, an essential component of the pipeline is the integration of security researchers. A dedicated module allows researchers to inspect the analysis results with the most likely deemed vulnerable drivers first, filter results, and provide feedback to the pipeline, including tagging drivers as true or false positives or prioritizing specific drivers for further analysis, such as fuzzing.

4.3 Modular Design of the Pipeline

The overall design of the pipeline utilizes a microservices architectural approach. The eight current modules are illustrated in Figure 4.1. These modules can be categorized into three distinct phases: the gathering and storage phase, the analysis phase, and the evaluation phase. This order of phases generally aligns with the sequence a driver file progresses through within the pipeline.

The further right a module is positioned in the graphic, the more preceding modules a driver file must pass through to utilize that module. The exceptions are the Frontender, which provides the user interface and can present partial results, and the Coordinator, the central communication hub with its API and the database participating in every task.

The first module to handle a file is an Importer, which imports files into the pipeline using the minimally defined storage interface. There are three instances depicted, as different Importer modules exist for various sources: a manual one, one handling VirusTotal API interaction, and the UpdateCataloger, which automatically gathers files from the Microsoft Update Catalog

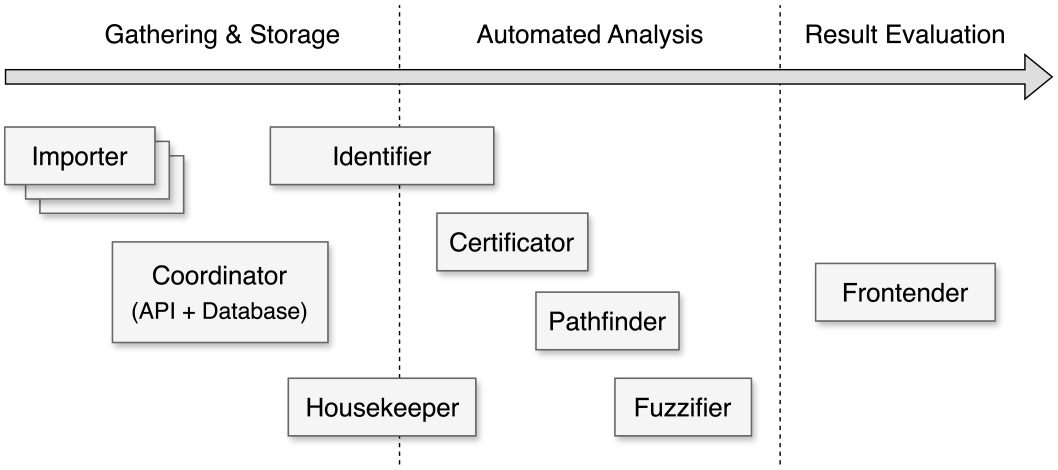


Figure 4.1: A conceptual overview of the implemented modules along the path a valid driver takes in the pipeline. From left to right, each module sequentially processes the file, except for the Coordinator, which serves as the communication channel, and the Frontender, which can also display partial results.

(see Section 3.4). The Importers then send the file to the pipeline's central component, the Coordinator, for storage. The Coordiantor module also provides the API for communication and the metadata database.

Next, the Identifier queries new submissions to identify files and, together with the Coordinator, recognizes files as known vulnerable drivers. The Housekeeper module queries identified files to extract archives and remove unnecessary files to clean up storage.

Following identification, the Certificator module adds information about the signature status to each file. The Pathfinder is the first module exclusively operating on drivers, augmenting them with information on paths to potentially interesting kernel APIs. The final analysis module is the Fuzzifier, which attempts to automatically fuzz test drivers for which the prerequisites were met through previous modules. Currently, the only module clearly in the evaluation phase is the Frontender, which provides a user-friendly interface for searching and viewing the results of other modules.

The following subsections describe specific design decisions for some of the abovementioned modules comprising the pipeline.

4.3.1 Identifying Interesting Functions

The Pathfinder module aims to identify paths from user-controllable inputs to interesting Windows kernel API functions or intrinsics within drivers. Functions or intrinsics that can be exploited to alter privilege levels are categorized as interesting functions and contribute to the heuristic that makes the pipeline deem a driver more likely vulnerable.

The highest interest level of kernel APIs is given because of their potential for simple exploits or weak constraints, such as arbitrary physical memory access via `MmMapIoSpace` and `ZwMapViewOfSection`. Others are crucial for interaction with the kernel driver from user space, such as `IoCreateSymbolicLink` and `WdfDeviceCreateSymbolicLink`. Table 4.1 gives an overview of a few of the most highly valued interesting functions and the origin of where they were found to be discussed previously.

Function	Intrest level	Source
`MmMapIoSpace`	5	TAU
`MmMapIoSpaceEx`	5	TAU
`IoCreateSymbolicLink`	5	KDR
`ZwMapViewOfSection`	4	REW
`WdfDeviceCreateSymbolicLink`	5	DKR
`ZwOpenSection`	3	REW
`IoAllocateMdl`	3	SELF
`SePrivilegeCheck`	3	SELF
`WdfDeviceCreateDeviceInterface`	3	DKR
`IoCreateDevice`	3	SELF
`WdfIoQueueCreate`	3	KDR

Table 4.1: Eleven of the most interesting kernel API functions, with their assigned interest level and source. TAU = original propagation script [22], KDR = WDF driver bug hunting presentation at 44CON [61], REW = blog post about local privilege escalation [68], SELF = identified as interesting by reading Microsoft documentation

Interesting intrinsics and memory operations the Pathfinder module looks for include: [1]

- **I/O Port Read/Write**: Intrinsics like `__indword` and `__outdword` facilitate reading and writing to arbitrary hardware I/O ports. Writing to hardware ports, such as the SPI controller, is required for firmware rootkit exploits.
- **Model Specific Register Read/Write**: Functions like `rdmsr` and `wrmsr` for AMD, and `_ReadStatusReg` and `_WriteStatusReg` for ARM, allow modifications of model-specific registers used for security control in Windows.
- **Memory Operations**: Memory move or copy operations, although not directly exploitable, are necessary when combined with memory mapping functions like `MmMapIoSpace` to construct an exploit.

4.3.2 Fuzzing Payload

The Fuzzifier module employs a fuzzing framework to automatically fuzz a subset of drivers that comply with the framework's constraints. A critical design decision for this module was determining the payload structure received by the framework's fuzzing agent. The challenge lies in the fuzzer mutating this payload to generate new inputs for the driver to achieve greater coverage. Thus, all valid arguments for an IOCTL call must be encoded within the payload to enable their effective mutation.

Of the two payload designs considered, the one depicted in Figure 4.2 was selected. This payload encodes the IOCTL code in little-endian format within the first four bytes of the header (violet), allowing for the discovery of valid IOCTL codes and the exercise of all accessible execution paths. The subsequent four bytes of the header represent the input and output size, which are mutatable because the fuzzing harness generates buffers of this size for the I/O manager. Hence, the fuzzer can pass checks on buffer sizes performed by drivers using this size mutability. Following this header is the dynamically sized input buffer (blue), which can be up to 0xFFFF bytes in size due to the two-byte size value in the header. Directly after the input buffer is the output buffer (light blue), which is also mutatable because of the Direct Input/Output and Neither transfer methods (refer to Section 3.2.3 for details).

[1] Only the I/O port intrinsics were part of the released TAU script, although the accompanying blog post also mentions vulnerabilities in MSR write operations.

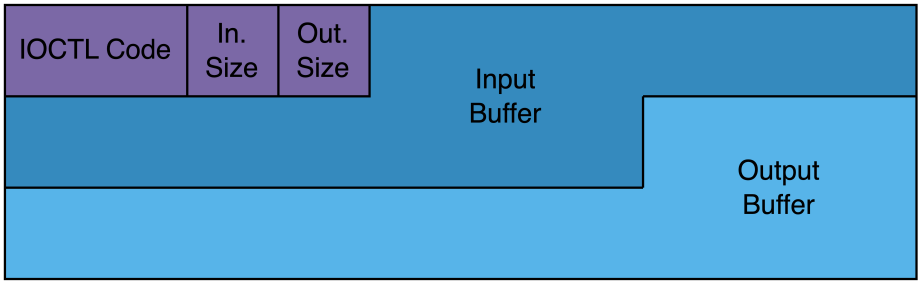


Figure 4.2: Representation of current fuzzing payload layout. The IOCTL Code, input and output size are all encoded within the header (violet), while the rest is the input buffer (blue) directly followed by the output buffer (light blue).

Another payload design featured an outer header with a one-byte value indicating the number of subsequent inner payloads described above. This design allowed a single fuzzing iteration to call multiple IOCTL requests and exercise possible multi-interaction paths. However, due to the significant state explosion of the input, the fuzzer could not crash simple test drivers within a reasonable time with this more complex payload. Hence, this more complex design was rejected for the more straightforward single IOCTL code payload.

Chapter 5

Implementation

The system developed as part of this thesis is structured as a pipeline comprising multiple modules, each responsible for different aspects of the vulnerability discovery process. This section begins with an overview of the implemented pipeline, followed by detailed descriptions of the modules in the subsequent sections.

5.1 Pipeline Implementation Overview

As discussed in Section 4.2, this pipeline is implemented using a microservices architecture to meet the diverse requirements of the various modules. Wherever feasible, the modules are containerised using Docker to ensure reproducibility across different environments. Docker Compose files were created to define their running configuration statefully.

Nearly all modules are written in Python, primarily because previous work has been written in it and the authors have prior experience in it. Other programming languages employed include Go and TypeScript for one of the modules and the front end, respectively, and Rust for a Proof-of-Concept exploit.

The diagram in Figure 5.1 provides an overview of the architecture of the implemented pipeline. It highlights the various modules involved in the pipeline's implemented vulnerability discovery process, with their communication channels depicted with arrows. The modules are grouped into numbered blocks representing the hosts they run on. The docker host ①, the Windows machine ③, and the redirector ④ are virtual machines, where the former two are hosted on internal infrastructure and the latter in the cloud. The manually started importers ⑤ and the fuzzing host ② are physical hosts. The blue modules are containerised, while all others are directly run on the respective hardware. Running modules directly is done through screen sessions on Linux-based systems and the Remote Desktop Protocol on Windows-based ones.

All communication between modules is conducted over HTTP, with only the modules responsible for performing tasks initiating communication. Specifically, whenever an instance of a module is ready to process the next file or driver, it polls the central coordination module for available tasks. Subsequently, the results are sent back using a push-based mechanism upon completion and as such, all modules must be able to reach the Coordinator. When running the containers through Docker Compose within a single VM, Docker's local networks handle reachability. For modules running on the same server but within different VMs, the Coordinator is reached by a hard-coded IP address. The company's in-house server cluster, where the firewall does not permit incoming traffic and the necessary hardware cannot be placed within the same network, requires routing of the Fuzzifier traffic through a cloud redirector.

All file and metadata storage is integrated into the Coordinator module, as described in Section 4.2. Given this pipeline's microservices architecture, any other module requiring a file must download it on demand. While it is technically feasible to give direct access to the files

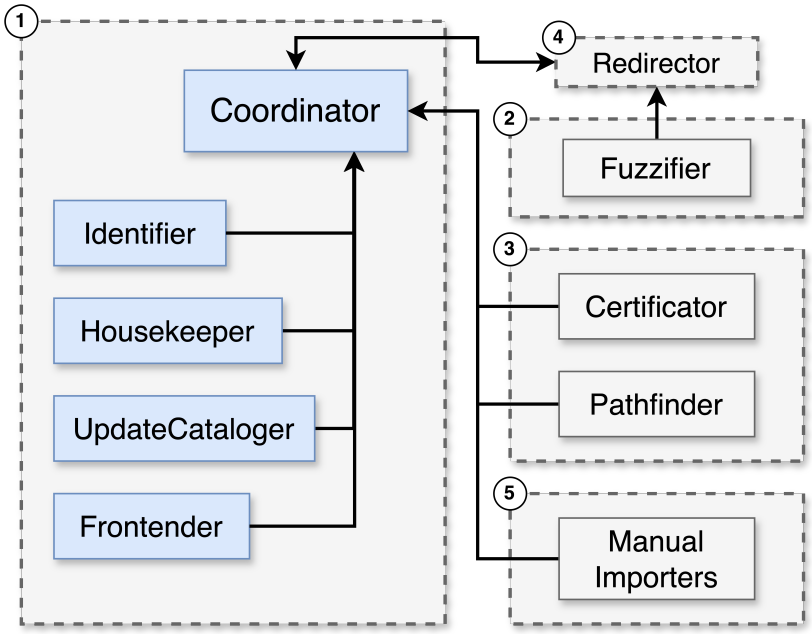


Figure 5.1: Overview of the implemented pipeline modules. Blue-depicted modules are containerised, whereas white modules are executed directly on the host. Communication directions are depicted with arrows, and the numbered squares indicate the host groupings.

to all modules running on the same VM as the Coordinator, this approach would necessitate two different interfaces depending on whether the module is running alongside the Coordinator or not. Given the network's fast transmission speed and because drivers are often only a few megabytes in size, repeatedly downloading on demand is more straightforward.

5.2 Coordinator — The Central Piece

The central part of the pipeline is the so-called Coordinator module. As the name suggests, this module handles all of the coordination of moving files through the different analysis modules and storing them. It consists of two containers: a PostgreSQL database container for storing the metadata and pipeline information and a Python container running a web server to access all of it. Both containers have persistent mounted volumes to store their data. The web server of choice is Flask, with Flask-SQLAlchemy to interact with the database using Object-Relational Mapping (ORM).

5.2.1 Database Schema and Object-Relational Mapping

The Coordinator manages hundreds of thousands of files and accumulates metadata for each. Consequently, the pipeline handles a diverse data set, resulting in a database structure comprising 14 data tables.

Database Markup Language (DBML), "an open-source DSL language designed to define and document database schemas and structures" [14], is used for the database schema. DBML's declarative and database-agnostic nature simplifies the database design process by abstracting complexities irrelevant to the schema design. The database design was developed using the free visualiser dbdiagram.io [24], which was also employed to convert the design into a PostgreSQL database definition.

An Object-Relational Mapping (ORM) framework simplifies interaction with the database. Despite various types of object-relational impedance mismatches [27] — the disparity between how object-oriented programming languages represent the world and how relational databases do — using objects during development is more straightforward. The specific impedance mismatch in this design is manually addressed in the ORM model definition.

Since Python Flask is the chosen web server, Flask-SQLAlchemy [3] was selected as the ORM framework. The DBML database definition is translated using "O! My Models" [83] to arrive at the SQLAlchemy ORM model definition. This translation is incomplete; thus, many-to-many relationships and unique multi-indexes must be manually added. Flask-SQLAlchemy then uses the ORM model to set up the correct tables within PostgreSQL and access data within the Flask endpoints. Because Flask directly creates the schema within the database upon first startup, the list of interesting kernel APIs (Section 4.3.1) and known vulnerable drivers (Section 5.2.2) must be added after.

5.2.2 Identifying Known Vulnerable Drivers

Another function of the Coordinator module is helping to identify known vulnerable driver files. The module relies on a compiled list of known vulnerable drivers to accomplish this. However, due to the absence of a single comprehensive source for such a list, a script aggregates this data from the following three sources:

- The Microsoft recommended driver block rules [51] provide a list of rules to identify third-party drivers deemed exploitable within the Windows ecosystem. The script downloads this XML file from the Microsoft Download Center and parses it, extracting, when available, the filenames and the SHA256 hashes of vulnerable drivers.
- loldrivers.io is another valuable resource, described as "a community-driven project that provides a curated list of all Windows drivers found abused by adversaries" [21]. It provides a JSON API to download all the gathered information, including filenames and hashes.
- The third source is a manually compiled list of filenames and SHA256 hashes of known vulnerable drivers, such as those identified in previous research [22] [86].

Two explicit and one implicit checks are performed to identify newly submitted files as known vulnerable drivers. The first check is implicit: whenever a new origin file is submitted and an identical underlying file, identified by its SHA256 hash, already exists in the database, the latest origin file is linked to the existing one. Consequently, for a known vulnerable driver already in the pipeline, the new origin file is also considered a known vulnerable driver because any file may have multiple origins but only one unique underlying file.

The two explicit checks are done directly after the Identifier returns the list of hashes for a newly submitted file. The first compares the SHA256 hash of the newly submitted file against the list of known vulnerable driver hashes. This method is employed when the above-described sources provide the SHA256 hash. These first two checks do not have false positives, assuming the list of known vulnerable drivers is accurate.

The third check involves comparing the filename of the newly submitted driver with those on the known vulnerable driver list. This check exists because some sources, like the Microsoft blocklist, only provide filenames and no hashes. While this method can produce false positives, such as when a new version of a driver shares the same name as a previously vulnerable one, it helps reduce false negatives by identifying widely known vulnerable drivers. Furthermore, any such potential false positives would concern drivers already blocked by the recommended driver block rules, making them less attractive for further exploration anyway.

5.2.3 Exposed Endpoints for Coordination

The Coordinator module receives HTTP requests from other modules to coordinate various tasks throughout the pipeline. The exposed HTTP endpoints can be categorised according to their relevant tasks. For a detailed list of API endpoints for each category, see Appendix A.7.

- **Note Taking:** Throughout the pipeline, almost every step generates some unstructured notes, including results, logs, or errors. Note-taking API endpoints are designed to return specific notes based on their type or associated file.
- **File Operations:** This category includes endpoints related directly to file operations, such as updating information about a file, searching for a file, or retrieving all files currently unidentified or identified as Windows executables.
- **Origin File Operations:** Operations involving adding or modifying origin files, including adding or searching for them.
- **Extractions:** This category pertains to finding and establishing relationships between archive files and their extracted files.
- **Certificate Operations:** The Certificator uses these operations to update driver signature information and the Frontender to query those results.
- **Pathing Operations:** The Pathfinder utilises these operations to retrieve drivers and update them with its results.
- **Fuzzing Operations:** This category includes endpoints specific to the Fuzzifier module, such as determining which driver to fuzz next and saving the results.
- **Fuzzing Queue:** Since fuzzing is much slower than other steps in the pipeline and configurable, a separate set of endpoints manages a prioritised queue for fuzzing.
- **Filtering Drivers:** Each endpoint in this category returns a list of drivers based on diverse filtering options. These endpoints are designed to reduce the filtering workload and transmission amount for the front end used by security researchers.
- **Driver Operations:** This category contains endpoints that allow searching for drivers or updating them with tags.
- **Miscellaneous:** This category includes any endpoints that do not fit into the above categories, such as the Docker health check endpoint or those providing general information about the database contents.

5.3 File Importers

The first step of the pipeline is gathering drivers. As described in Section 4.2, the storage interface is kept simple in order not to reproduce code between the importers. As such, every importer must only submit a file and its origin as a string. There are many sources of Windows kernel drivers for which importers would make sense, but because of time constraints, only the following were written.

5.3.1 Manual, VirusTotal and CDC Importer

The manual importer recursively searches through a user-provided folder and uploads all files not already present in the pipeline. The user specifies the origin string for these files. For files that already exist, the file is not uploaded again; instead, only the new origin is added and linked to the existing underlying file, provided this exact origin does not exist yet.

Gathering the drivers for submission through this manual importer can be achieved in several ways. One straightforward but labour-intensive method involves manually finding and downloading drivers from online sources. Interesting drivers might have intriguing names or are said to behave in unexpected ways, such as blue-screening Windows. Another manual source of first-party drivers from Microsoft is operational Windows machines. The most straightforward approach to gathering these drivers is to search the C: drive of these systems for `*.sys` files, copying all found files into a single folder for subsequent submission with the manual importer.

A more automated method for gathering drivers, frequently employed in similar research, involves using VirusTotal (see Section 3.4). More specifically, the file download API [81] in conjunction with VirusTotal Intelligence. The powerful filter criteria of the latter can, for example, be used to create the following query to search for signed Windows executables that end in `.sys` and import a specific function:

```
tag:signed type:executable name:*.sys imports:IoCreateSymbolicLink
```

Intelligence queries via the API return metadata, while the files must then be downloaded using the file download API. Although a VirusTotal premium subscription was used and intelligence queries only require one API quota per page of results, the file download API is much more costly, consuming one API quota per file downloaded. The provided subscription only allows 1000 API calls per day, shared among multiple internal teams. Only files not already present in the pipeline database are downloaded for later submission to minimise quota usage. This optimisation is possible because the initial metadata already includes the file hash. Interaction with the VirusTotal API goes through the Python package vt-py [82].

Another source of drivers was the cooperation with the Cyber Defense Center at InfoGuard AG (see Section 3.4) to collect drivers across real-world client networks. This collection was achieved using deployed extended detection and response (XDR) sensors. For this thesis, the sensor was Microsoft Defender XDR, although other XDR solutions should work similarly.

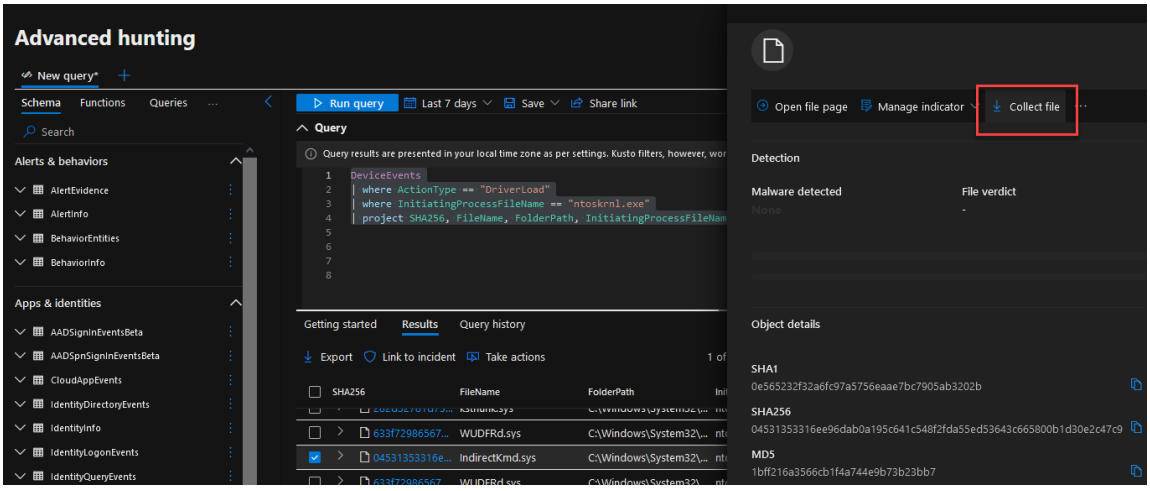


Figure 5.2: Microsoft Defender XDR advanced hunting for Windows Drivers in a client network.

Specifically, the Microsoft Defender XDRs feature called advanced hunting allows querying for all drivers loaded on any endpoints within the client's network. The shown MS XDR query

in Figure 5.2 identifies all drivers by searching for events of type `DriverLoad` by the Windows kernel (ntoskrnl.exe), which are triggered whenever a driver is loaded. This query then returns the SHA256 hash and filename of each loaded driver. Although the UI includes an export function for the list of returned drivers, files can only be downloaded individually through the UI's "Collect file" button, indicated in red.

Accessing the Microsoft Defender XDR API to download files in bulk would necessitate registering an application within each client's network. Due to the development time associated with such an application and the challenge of securing approval from each client to install it on their network, API bulk downloading was deemed impractical within the scope of this thesis. However, manually clicking the collect button hundreds of times in the UI is also infeasible, so a multi-step approach to downloading these drivers was applied. First, the pipeline was queried; if the file existed, it was retrieved from there. Otherwise, the file is downloaded from VirusTotal if it is available there. The process reverted to manual download via the UI for any drivers not found through these two methods.

5.3.2 UpdateCataloger — Fully Automated Importer

The Microsoft Update Catalog (see Section 3.4) is the closest to a complete source available to the public for all Windows drivers. While it does not directly provide a complete list of drivers and has limitations on the number of results for any search query, various tools in different programming languages interact with this website. The UpdateCataloger module leverages one tool, `get_microsoft_updates.py` [7], with some modifications to automatically and periodically search for new drivers.

The modifications include an added caching layer for previously downloaded metadata and logging whenever the search result limit is reached. The caching layer reduces the number of metadata and download requests sent to the update catalog servers when search query results overlap with previously seen ones. Such overlapping results, including those from previous runs of the same query, have already been added to the pipeline. Logging of the search result limit helps identify which search queries need a narrower search scope.

Concretely, the UpdateCataloger periodically uses the modified script to download all new results for a list of search queries. It submits those files to the pipeline, with the origin containing the unique identifier for the update provided by the Microsoft Update Catalog. The list of search queries includes all possible vendor IDs (ranging from 0x0000 to 0xFFFF), a manually curated list of computer component manufacturers, specific keywords to narrow the scope (primarily names of computer devices such as mouse, camera, audio, modem, BIOS, etc.), and the Cartesian product of the manufacturers and these scope-narrowing keywords.

5.4 Identifier — File Classification

Due to the storage interface's minimalist design, the initial analysis stage of the pipeline involves completing any missing metadata for a file, such as its type and possibly filename. This stage also includes determining whether the file is a Windows driver. The Identifier module performs these tasks in cooperation with the Coordinator, which processes the Identifier module's results.

Given the commonality of file identification, various tools have been developed to somewhat accurately determine file types, such as ExifTool [23] and TrID [65]. Additionally, a fully containerised tool utilising these two is available with the Malice File Info Plugin [6], which offers their results as a web service. Consequently, the Identifier module is a minor adaptation of the Malice File Info Plugin, where a polling loop for unidentified files has replaced the original web server.

The Coordinator endpoint stores the identification results and employs the following heuristic approach to determine if the file is a Windows executable. If the file is likely an executable

according to TrID, if its MIME type is PE32 or PE32+, if the filename saved in the metadata ends with `.sys`, and the file architecture is recognised, then the file is assumed to be a Windows executable.

Once identified as a Windows executable, the Coordinator endpoint further examines whether the file is possibly a Windows driver. This is done by inspecting the list of imported functions in the Portable Executable (PE) header. The import list is extracted using the Python library pefile [10], with code adapted from the Malice PExecutable Plugin [5]. Specifically, it checks if the Windows Kernel (ntoskrnl.exe) or the Windows Kernel Mode Driver Framework Loader (wdfldr.sys) are imported. Either indicates that the executable is intended to run in kernel space, suggesting it is a kernel driver.

Suppose the file is identified as a Windows driver. In that case, the endpoint additionally verifies if the driver file contains any of the following three types of strings and, if so, separately stores them as static results: SDDL strings (details provided in Section 3.2.4), the string "PhysicalMemory" itself, or any string that the driver might use for symbolically naming the device (refer to Section 3.2.3 for more information).

5.5 Housekeeper — Storage Cleanup

The Housekeeper module performs two primary functions: extracting known archives and minimising the pipeline's storage requirements. If storage constraints were not an issue, this module would focus solely on extracting known archive formats without deleting files. However, due to limited storage space, this module removes files that do not serve a further purpose.

The process of archive extraction is similar to file identification. Archive files are downloaded from the Coordinator and then extracted. To facilitate this, the Coordinator provides an endpoint that returns only files of a specific type. This module uses this endpoint to download Microsoft Cabinet archive files and possible installer executables. These files are locally extracted using cabextract [9] and 7-Zip [64], respectively, and the extracted files are reintroduced to the pipeline with a reference to the original file. For executables, 7-Zip instead of the dedicated UniExtract2 tool [17] is used because the latter did not work reliably during testing.

In addition to removing successfully extracted archives, the module eliminates files that are deemed unnecessary for the pipeline. Like the extraction process, the Coordinator supplies an endpoint that returns files falling into this category, including their identification results. These results help determine whether a file remains relevant to the pipeline. Specifically, if a file is itself or is likely to contain a driver, it is retained. Text files are generally unlikely to contain drivers and can be safely removed, except INF files, which may contain SDDL strings relevant to driver access. Additionally, any successfully extracted archive is deleted, as any files within were just reintroduced to the pipeline. The removal process only deletes the file from the filesystem while maintaining the database entry, ensuring that the same file is not inspected again.

5.6 Certificator — Signature Inspection

The Certificator module extracts digital signature details from drivers, including the certificate chains and their respective signers. While various implementations for this extraction process exist, the Certificator is a wrapper around the official Microsoft Sysinternal tool, Sigcheck [69]. Despite the drawback of running it on Windows and not containerised along the Coordiantor, Sigcheck offers the advantage of checking local Certificate Revocation Lists (CRL), ensuring the validity of the certificates. Alternative implementations may use the WinVerifyTrust API to achieve similar results. Still, not checking the CRL might erroneously identify drivers as signed, even if their digital signature is invalid. Additionally, utilising a Microsoft-provided tool ensures future compatibility with minimal effort.

```
sigcheck64.exe -accepteula -h -i -e -w <output> <driver>
```

The Certificator executes the above command for each identified driver file, where the first argument (-accepteula) runs Sigcheck without prompting for the EULA, thus avoiding the need for user interaction during execution. The second argument (-h) reports the file's SHA256 hash, serving as a sanity check to ensure the correct file is inspected. The third argument (-i) directs Sigcheck to display the complete signing chain. The fourth argument (-e) tells Sigcheck to disregard file extensions, as the Certificator only processes drivers already verified as executables by the pipeline. The second to last argument (-w) specifies that the output should be written to a file, while the last argument indicates the file to be inspected.

After parsing the Sigcheck output using regular expressions, the results are sent back to the Coordinator. The corresponding endpoint then stores the final signature status along with the digital signature metadata, including the company, description, product name, product version, if available and all signatures and their signers.

5.7 Pathfinder — Static Reachability Analysis

The Pathfinder module serves as an extension and modification of the prior work by VMware TAU [22], as summarised in Section 3.3.1. Similar to other modules, the Coordinator module has corresponding endpoints. These endpoints are used first to obtain a set of drivers that are not inspected by this module but match the correct architecture. Subsequently, they are used to update the database with the results from the Pathfinder module. These results include the driver framework used, addresses of potential IOCTL handler functions, paths to interesting kernel APIs, possibly used IOCTL codes, and, if applicable, a list of WDF functions used by the driver.

The Pathfinder module acts as a wrapper by interacting with the Coordinator, invoking the modified IDAPython script from VMware TAU on each driver, and sending back the results. Running the IDAPython script entails executing IDA Pro from the command line with the script and driver binary as arguments. Thus, this module utilises Python's subprocess functionality. Since the IDAPython script is executed as part of the IDA process rather than directly by the Pathfinder script, structured communication of results is achieved through a result file written to a specified location, which is then read by the wrapper after the IDA process exits.

The modifications to the original IDAPython script, described in the rest of this section, include enhancements to recognise and handle a wider variety of drivers, additions of target APIs, speed improvements, and additional features relevant only to the pipeline context.

5.7.1 Expanding Driver Compatibility

One modification to the original script is support for ARM64 drivers. The primary issue with the original script for ARM64 drivers is that IDA Pro 8.4 lacks built-in type information for them, unlike for AMD64 drivers. Consequently, after the driver's initial loading, although disassembly is successful, most kernel driver types are not defined or recognised, including the driver entry arguments or the DRIVER_OBJECT type used for WDM IOCTL handler detection. The modified IDAPython script addresses this by recognising ARM64 drivers and loading the relevant type information libraries. The hardcoded offsets for structs in WDM drivers were also modified to change conditionally based on the driver's architecture.

The relevant type information libraries (TIL) are not readily available online; therefore, multiple small batch scripts using idaclang, a modification of the CLANG compiler by HexRays, were created to extract this information. [1] Specifically, three new type information libraries for

[1] Finding an example or detailed description of how to create new type information libraries (TIL) was much harder than expected. Three helpful sources are [4] [77] [75].

the Windows Native API, Windows Kernel API (NT DDK), and Windows Driver Framework were created by running idaclang with the target architecture ARM64 across the *Windows.h*, a modified *ntddk.h*, and *wdf.h* header files, respectively. The modified NT DDK header was necessary because idaclang failed to compile a single type definition from the original header.

Although these modifications allowed the IDAPython script to run successfully for ARM64 drivers, most WDF drivers were not recognised, necessitating further changes to compensate for errors in the IDA Pro decompilation of some drivers. IDA Pro sometimes fails to recognise the correct fastcall type of the indirect call `WdfIoQueueCreate`, mainly by not identifying the proper number of arguments. This issue is more pronounced on ARM, possibly because of weaker support for ARM by HexRays. Consequently, this case was overlooked in the original script, resulting in the script failing to find the IOCTL handler when the correct `WdfIoQueueCreate` is skipped due to incorrect argument numbers. The fix involves checking if the `WdfIoQueueCreate` call has the expected five arguments, and if not, redefining it with the correct type before redoing the original check. This modification also applies to AMD64 drivers, though the issue occurs less frequently because IDA Pro nearly always recognises the correct number of arguments during testing.

Another modification addresses how the IOCTL handler function for WDM drivers is identified. The original script only checks for simple assignments to the IOCTL handling MajorFunction array member of the DRIVER_OBJECT. This method is insufficient for recognising IOCTL handler functions in drivers that use a single global function to handle all major function codes.

The screenshot of one such example can be seen in Figure 5.3, where a `memset64` function is used to copy the same global handler to all major function codes. Therefore, the modified IDAPython script simultaneously searches for simple assignments and any `memset64` instances where the first argument might be any of the major function offsets, and the second argument is an address to a valid function start.

```
a1->DriverUnload = (PDRIVER_UNLOAD)sub_11580;
memset64(a1->MajorFunction, (unsigned __int64)fn_ioctl_handler_wdm_memset64, 0x1BuLL);
a1->MajorFunction[3] = (PDRIVER_DISPATCH)&sub_11958;
a1->MajorFunction[22] = (PDRIVER_DISPATCH)&sub_11A94;
a1->MajorFunction[27] = (PDRIVER_DISPATCH)sub_11AD8;
fn_11BD8(a1);
```

Figure 5.3: Example of IOCTL handler being set by a `memset64` of all major functions locations.

In addition to using the `memset64` function to set a single global handler, simple loops are sometimes used to achieve the same result. Consequently, another check was added for loop variable assignments representing an increasing major function to a valid function address. However, because IDA Pro decompiles loops into different expression trees, this check cannot handle all encountered loop versions in its current version. Disassembly and decompilation views in Appendix A.4 show examples of supported and non-supported loops.

Although minor, the inclusion of additional deemed interesting functions and, where applicable, setting their types for WDF functions was also added. Adding these functions was straightforward: appending them to a list and defining their correct function types.

5.7.2 Improving Script Execution Time

Further, two intended speed optimisations were implemented, given the pipeline's application across numerous drivers. The first optimisation involved replacing any global variable debug conditions with conditions based on Python's internal `__debug__` variable. Running the modified script with Python's built-in optimisations activated — achieved by setting the environment variable `PYTHONOPTIMIZE = x` when running IDA Pro — results in the Python interpreter

entirely removing them. The second optimisation aims to avoid redundant decompilation expression tree traversals whenever possible. Since pre- and post-order traversals are recursively called on any invoked function, subfunctions could be inspected multiple times. This issue was addressed by caching any pre-order entry to a function and skipping further calls.

5.7.3 Decompilation Context and WDF Import List

Any of the following modifications in the Pathfinder IDAPython script are only relevant to the pipeline and, hence, not part of the original script. One minor addition was obtaining a decompilation context around any found pathing results, inspired by pwndbg's implementation of context retrieval around a decompilation address [13]. This context is returned to the Coordinator for each target API hit, which can enhance the information a security researcher sees when evaluating whether this target API is vulnerable. Two further features were implemented. One of them facilitates the extraction of Input/Output Control Codes from a driver, described in Section 5.7.4, which is later helpful in seeding potential fuzzing efforts.

The other feature involves completing the list of functions a driver uses if written with KMDF. Using the idea of how the original script finds `WdfIoQueueCreate`, described in Section 3.3.1, a feature was implemented in the Pathfinder module to complete the import list saved for a WDF driver by these WDF functions. This results in the import list no longer representing only real PE imports for WDF drivers but rather any functions a driver developer might import during development to use in the driver. Concretely, a new decompilation tree visitor was implemented that similarly checks indirect calls for any valid offset into the global WDF functions table. These valid offsets result from a small script extracting them directly from the Windows Driver Framework struct definition. This tree visitor is called on every expression tree of a driver with at least one reference to the global WDF function table while aggregating all found WDF functions. Additionally, this tree visitor tries to recognise if the WDF driver is compiled in release or debug mode by inspecting if the WDF functions have API wrappers around them, similar to how the original script tries to recognise the API wrapper around `WdfIoQueueCreate`.

5.7.4 Extraction of Input/Output Control Codes

A feature added to the Pathfinder, independent of the original work of VMware TAU, involves extracting Input/Output Control Codes for subsequent use by the fuzzing module. During the input-output propagation phase, any function address utilising input or output APIs is added to a set of functions employed by this IOCTL detection feature. This set of function addresses is then decompiled and examined for potential IOCTL codes. Any identified IOCTL code is forwarded to the Coordinator along with a corresponding comparator. The comparator is used to identify ranges where the driver might expect IOCTL codes. This is necessary because, despite IDA's generally reliable decompilation, compiler optimisations can prevent all I/O control code checks from being recognised as simple switch-case statements, thus preventing them from being able to be recognised as direct comparisons.

Through manual examination of decompilation outputs across numerous drivers, three representations of the IOCTL code checks were identified, each addressed by a separate implementation within the detection feature:

- **Switch Case Statement**: The simplest code flow check is based on a switch-case statement (Figure 5.4a). The Pathfinder can extract this using a regular expression whenever a switch-case statement on an I/O control code is detected. These IOCTL codes have clear code flow paths, so the Pathfinder can confidently assert their validity and save them as equal comparisons.

```
switch ( CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode )
{
  case 0x222664u:
    *(_DWORD *)Irp->AssociatedIrp.SystemBuffer = dword_13110;
    *p_Information_4 = 4LL;
```

(a) Example of IOCTL Code checking through switch case statement.

```
_IoControlCode = CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode;
if ( _IoControlCode <= 0x9C4060D4 )
{
  if ( _IoControlCode == 0x9C4060D4 )
    goto LABEL_21;
  if ( _IoControlCode <= 0x9C40208C )
```

(b) Example of IOCTL Code checking through a series of condition checks.

```
Irp->IoStatus.Information = 0LL;
if ( KeGetCurrentIrql() )
  goto LABEL_74;
__IoControlCode = IoControlCode - 2236708;
if ( __IoControlCode )
{
  if ( __IoControlCode == 4 && outputLen )
```

(c) Example of IOCTL Code checking through arithmetic calculation and simple conditionals.

Figure 5.4: Three different representations of IOCTL code flow checks in IDA decompilation.

- **Hardcoded IOCTL Value Comparisons**: The I/O control code is frequently compared to hardcoded values for various functionalities within the driver (Figure 5.4b). Although these comparisons involve hardcoded values, compiler optimisations result in a mix of equal, less than, and greater than comparisons. A regular expression extracts both the condition values and the types of comparisons, which can later be used to create ranges within which possible IOCTL Codes might be.
- **Incremental Comparison**: The final type of check termed incremental comparison checks (Figure 5.4c), involves compiler optimisations that perform incremental arithmetic operations instead of directly comparing hardcoded values. The I/O control code detection feature must first identify the local variable used for the comparisons, then incrementally perform the arithmetic operations starting from the initial big subtraction and subsequently save each comparison made. These comparisons typically involve non-zero and zero checks followed by further arithmetic operations and checks or comparisons to low values ranging between 4 and 16.

5.8 Fuzzifier — Automated Dynamic Analysis

The Fuzzifier is the last analytical module in the pipeline. It demands the highest computational resources, has slower processing speed, and a dependency on outputs from preceding modules. As its name suggests, this module performs fuzz testing [57] on drivers identified as interesting and feasible for fully automated fuzzing. Fuzz testing is at the core of this module in the form of Intel's kAFL (see Section 3.3.2). This thesis's work includes the wrapper around this tool and its integration into the pipeline, as well as a modified Windows kernel driver fuzzing harness.

Due to the requirements of kAFL, the Fuzzifier module must operate on dedicated hardware that meets specific criteria. As such, it runs separately from other modules. This dedicated hardware lacks direct network connections to the Coordinator; thus, a redirector is established to facilitate communication. This redirector exposes the Coordinator API on a public IP address,

accessible only to allow listed IPs, including the Fuzzifier hardware. Additionally, the Fuzzifier module necessitates a fully functional installation of kAFL, including the modified Linux kernel and a valid kAFL environment (verified through environment variables).

Only one driver is fuzzed concurrently. The Fuzzifier executes the following steps using Python's subprocess library to achieve fuzzing results:

- Retrieving the next driver fuzzing configuration from the fuzzing queue, locally saving the fuzzing payload seeds and the driver and configuring the fuzzing agent accordingly. The configuration of the fuzzing agent involves source code templating, which is then compiled as per kAFL instructions.
- Preparing the virtual machine as kAFL specifies, ensuring the driver under test is within the VM, loaded, and running at VM startup, along with the fuzzing agent.
- Executing `kAFL fuzz` as a subprocess while continuously performing timeout and progress checks via the kAFL built-in GuiData API.
- After fuzzing concludes, collect relevant data returned by kAFL, including logs and final payloads, send this data to the Coordinator, and clean up the host for the next fuzzing target.

Any errors encountered during setup or fuzzing, excluding target crashes, result in premature termination of fuzzing. Such errors are logged, and the corresponding fuzzing configuration is marked as unsuccessful. If no errors occur, the configuration is marked as successful. Either way, the Fuzzifier returns to the initial step to fuzz the next driver.

5.8.1 kAFL Fuzzing Harness Modifications

kAFL includes an example of a Windows kernel driver fuzzing harness, which has been modified for use with the Fuzzifier. These modifications constitute nearly all the customisation done to kAFL within this module, aside from a bug fix addressing QEMU crashes.

Aside from templating the fuzzing agent with the corresponding driver's symbolic name and recompiling it for each driver, the modifications primarily involve how the fuzzing agent processes the fuzzing payload and checks the response code. Specifically, the fuzzing harness decomposes the fuzzing payload defined in Section 4.3.2 into two buffers with specified sizes and the IOCTL code. It catches any payloads that do not adhere to the defined form, such as when the input and output sizes do not sum to the total payload size, which can result from fuzzing mutations or wrong user-defined seed payloads.

The current fuzzing harness then interacts with the driver using `DeviceIoControl` and the defined IOCTL code, subsequently checking the result. An ERROR_NOACCESS returned by `DeviceIoControl` may indicate that the IOCTL code was valid and worked, but memory access was denied. This could suggest that the driver attempted to access memory protected by the kernel, which, if the address was user-defined, is likely a bug or that the driver detected such a wrong address and returned the error code. Given the Fuzzifier is unable to distinguish these two cases, a crash is enforced to save the used payload for later inspection.

5.9 Frontender — Security Reasearcher Interaction

The pipeline collects drivers and attempts to identify likely vulnerable drivers through static and dynamic analysis. During this process, results are generated for each driver. Since the pipeline cannot definitively determine whether a driver is vulnerable, presenting these results in a useful format is essential for security researchers to verify them efficiently. The Frontender module provides a website that displays all results in a human-navigable form while allowing the modification of driver metadata and the manual addition of jobs to the fuzzing queue.

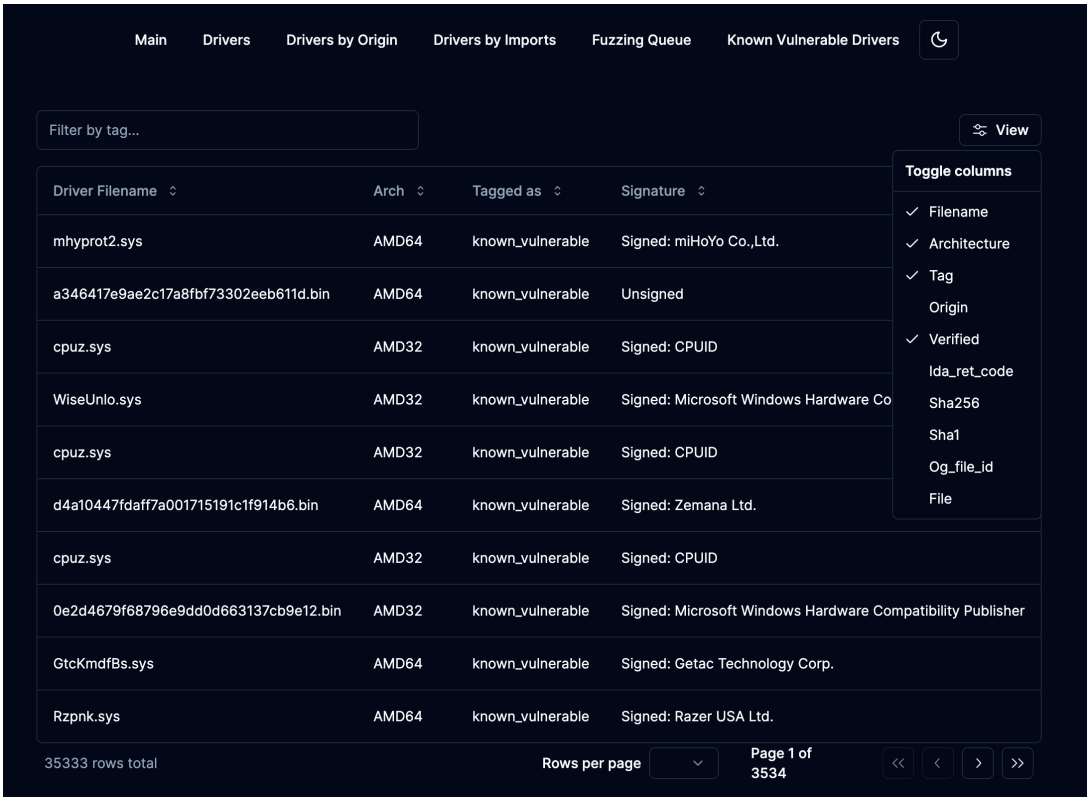


Figure 5.5: A screenshot of the driver overview table showing only a subset of the metadata fields presented by the Frontender website.

From a technical perspective, the Frontender module operates as a containerised Node.js server (version 18) hosting a React website, developed in TypeScript and primarily styled with shadcn/ui components [74]. It employs Vercel's stale-while-revalidate (SWR) [79] implementation for HTTP caching in the front end, enhancing user experience. The shadcn/ui components provide fast development of aesthetically pleasing websites.

The Frontender website provides three types of tabs for displaying pipeline data tailored to different tasks a security researcher might engage in when working with pipeline results. The first tab type offers an overview of the gathered drivers, including metadata and summarised pipeline results per driver. This overview is presented in table form, as illustrated in Figure 5.5. Users can select from various metadata fields, such as filename, driver architecture, driver tags (e.g., known vulnerable), file origin, validity of driver signature, and additional file metadata like hashes. This table allows sorting by most metadata fields and includes pagination for navigating the list of drivers. Three tabs present this table view, each offering different filter options for the drivers they display:

- **Drivers Tab**: Queries all existing drivers in the database with the provided tag. By default, it displays drivers with any tag and searching for substrings is also supported.
- **Drivers by Origin Tab**: Allows users to restrict the displayed drivers to those from a specific origin, useful when cooperating with the CDC. It enables the results to be filtered to a single client if the origin was defined as the client's name during import.
- **Drivers by Imports Tab**: Limits the displayed drivers to those with all functions defined in the search field as part of their imports, including WDF functions.

Each driver shown can be inspected in greater detail by clicking the driver filename. The detailed driver view presents information organised into four sections. The top section displays general file metadata, including the filename, architecture, current tag, and various hashes. Additionally, through a dropdown menu, actions such as modifying the tag and filename, adding the driver to the fuzzing queue and downloading the driver are possible. The second section shows the identification step results, including functions the driver imports and interesting strings found, such as possible access names from user space or SDDL strings. The third section is dedicated to the Certificator results, indicating whether the driver is signed, including signature chains and signers. The Pathfinder results include the driver framework and whether paths to interesting functions were found. Lastly, the Fuzzifier results present statistics on all conducted fuzzing runs, including runtime invested, executions run, and paths and basic blocks covered. The screenshots in Appendix A.3 provide further visual representations.

The other two types of tabs provide information about the current fuzzing queue and the saved list of known vulnerable drivers. The Fuzzing Queue Tab displays three tables showing drivers currently being fuzzed, those that finished or encountered errors, and those still in the fuzzing queue. The Known Vulnerable Drivers Tab shows the list from Section 5.2.2 in a table similar to the driver's overview tabs.

5.10 Provided Hardware and Software

Several hardware resources were provided during the pipeline implementation and to run the modules. These included several virtual machines (VMs) hosted by the company in-house on a server running ESXi 7.0 U2. The VMs collectively offered 24 cores of Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz, 36GB RAM, and 2250GB of disk storage, nearly 2TB of which only for storage of the found files and the metadata database. In addition, another smaller virtual machine was hosted in the cloud to redirect traffic through the company firewall for any modules not running on the in-house server. For the fuzzing tasks targeting AMD64 architecture, an Intel(R) Core(TM) i7-9700K CPU with eight cores and 32GB of memory was utilised. Despite failing to fuzz ARM64 architecture drivers successfully, access was provided to a server equipped with a ThunderX CPU, which included the necessary ETMv4 support.

Specific modules rely on licensed software. HexRays IDA Pro licenses were provided for ARM64 and AMD64 architectures, which are crucial for the Pathfinder module. Additionally, a VirusTotal (VT) subscription was made available, permitting 1000 downloads or queries a day. However, fully automated interaction through VirusTotal, even with limits, was not feasible due to the shared nature of this subscription within the company. Each use, including the expected number of queries, required prior approval.

Chapter 6

Evaluation

This section presents the evaluation results of various components of the developed pipeline. It includes statistics on the collected drivers and files, an analysis of the functions defined as interesting and if they are present in known vulnerable drivers, the effectiveness of storage cleaning by the Housekeeper module, the utility of the features and improvements in the Pathfinder module, and an overview of the current results from the Fuzzifier.

6.1 Gathering of Drivers

This section first presents some metrics of the different origins from which the pipeline gathered the drivers and information about drivers found in collaboration with the CDC. This is followed by a simple historical distribution of when files and drivers were gathered and, lastly, by general statistics about the drivers.

Driver Origin Analysis

Figure 6.1 illustrates the distribution of origins for all drivers the pipeline currently knows about. A total of 48'942 origin files associated with drivers exist, with 43'699 unique drivers identified by hash. Of all drivers, 3'572 are written in WDF, and 9'783 are identified as WDM.

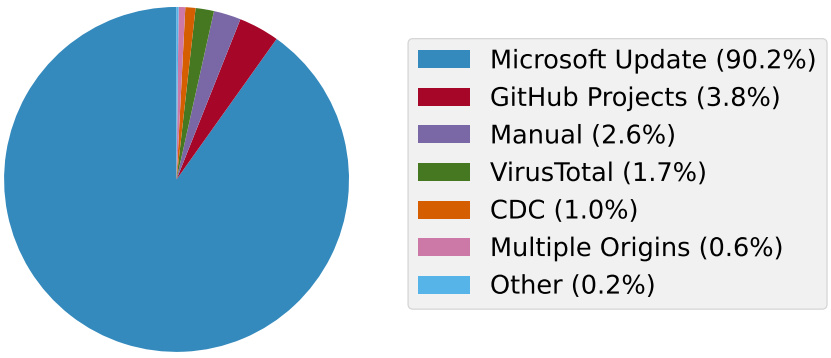


Figure 6.1: Drivers are categorised by their origin. GitHub Projects includes drivers like loldrivers.io, physmem_drivers, and CalendoniaProject. CDC are all Cyber Defence Center found drivers. Manual are any test drivers, and Other encompasses drivers that do not fit into these categories.

The primary source of drivers was the Microsoft Update Catalog, accounting for 90.2% of all drivers. Of these, 16'090 drivers (approximately 41%) were obtained through name searches, while the remaining 59% were acquired by brute-forcing vendor IDs. In total, 1'565 vendor IDs were found to return results. Notably, the restriction of showing only the first 1000 results on the Microsoft Update Catalog was encountered for 78 vendor IDs and 229 searches using words or compound words. Although Microsoft Update Catalog is the primary source of drivers, only 191 drivers are listed among the known vulnerable drivers.

Drivers collected from vulnerability GitHub repositories included 1695 from loldrivers.io, 137 from physmem_drivers, and 100 from CalendoniaProject, totalling 1709 drivers. These collectively cover 91.2% of all drivers in the list of known vulnerable drivers, which is partially based on loldrivers.io explaining this high coverage. Theoretically, all these GitHub project drivers should be on the list of known vulnerable drivers. However, 24 drivers are not in the initial list due to specific cases of the Microsoft recommended blocklist implementing blocking based on driver names for a small number of drivers. These exceptions are a limitation of the current implementation, which can only reference a single underlying file for a known vulnerable driver even if multiple versions exist due to an open block rule based solely on the driver name. Additionally, 22 drivers, all found in the loldrivers.io GitHub repository, are not recognised as vulnerable or part of the loldrivers.io website.[1]

Drivers sourced through VirusTotal queries numbered 817, with 90 identified as known vulnerable. The pipeline also included 1183 drivers added manually and 93 drivers from miscellaneous sources, such as testing drivers, 11 of which are known to have vulnerabilities. Furthermore, drivers sourced from approximately 33'300 endpoints in cooperation with the Cyber Defence Centers (CDC) totalled 593, with 12 distinct known vulnerable drivers and another seven distinct drivers identified by the analysis steps as likely vulnerable. The distribution across the three scanned clients is presented in Table 6.1. Notably, at least one known vulnerable driver was running in each client network.

CDC Client	**Total Drivers**	**Known Vulnerable**	**Likely Vulnerable**
CDC Client 1	108	1	3
CDC Client 2	319	4	1
CDC Client 3	250	7	4

Table 6.1: Vulnerability Status of Drivers Sourced from CDC Clients.

These statistics provide an overview of the distribution of driver sources collected by the pipeline for analysis. The predominant reliance on Microsoft Update as a source underscores its significance in driver acquisition. In contrast, drivers sourced from vulnerability gathering repositories clearly show that they are the primary source for the built-in list of known vulnerable drivers. Additionally, the CDC distribution of vulnerable drivers highlights this pipeline's apparent positive side effect when cooperating with them, as any already known vulnerable drivers can be identified and remediated.

Timeline of Driver Collection

To illustrate the dataset growth of files and drivers within the pipeline, Figure 6.2 presents a timeline of the cumulative number of origin files on the left and drivers on the right. Drivers identified as known vulnerable or manually configured as vulnerable are highlighted in red on the day they were added to the pipeline. The timeline also marks several significant events:

[1]These missing drivers appear to be a bug on loldrivers.io, given that most consumers download the JSON provided by the website instead of cloning the repository.

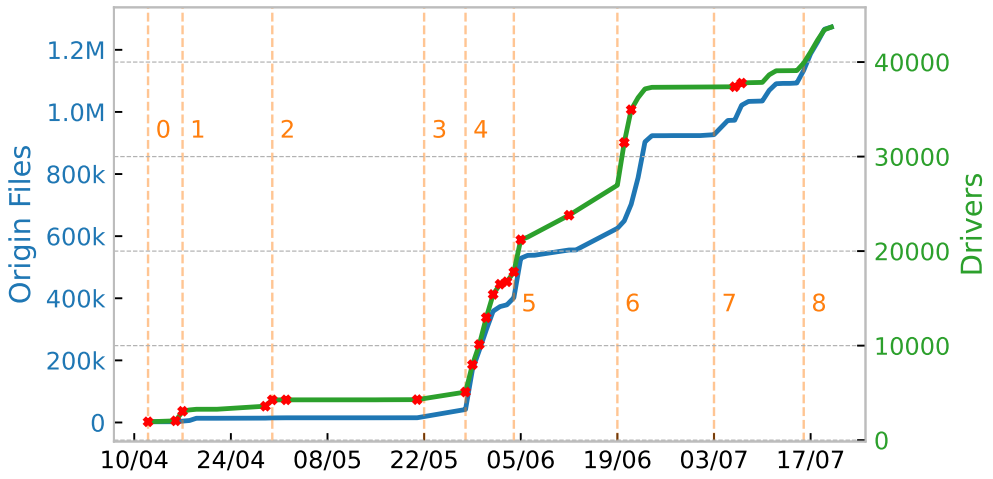


Figure 6.2: Timeline depicting the process of gathering files and drivers within the pipeline, annotated with significant events: (0) Database reset, (1) Manual search for specific drivers, (2) VirusTotal queries, (3) Collection from select GitHub projects, (4) Microsoft Update crawling, (5) Microsoft Update with brute forcing, (6) Storage capacity adjustments, and (7) 7-Zip extraction of executable files. The red dots depict the addition date of vulnerable drivers.

Although challenging to extract from the figure, the current database contained thousands of drivers on its first day (0), resulting from a database redesign and reset, during which all previous drivers were readded. These included the first CDC client, the loldrivers.io repository, and manually gathered drivers during pipeline development.

Initially, the gathering process was not fully automated. The subsequent three events that increased the number of drivers were the addition of default Windows first-party drivers on April 17 (1), the implementation and execution of VirusTotal queries around the end of April (2), and the incorporation of more GitHub projects like physmem_drivers and the CalendoniaProject (3).

Due to the inability to automate VirusTotal queries, a crawler for the Microsoft Update Catalog was added, significantly increasing the number of files and drivers found from May 28 (4). This crawler was enhanced to search for vendor identification through brute force, leading to the discovery of more drivers via the Update Catalog (5).

The Microsoft Update Catalog crawler required more storage than provided to handle tens of thousands of files. Additional storage was allocated on June 19, enabling it to operate at full capacity again (6).

While increasing the number of files, initially extracting executables with 7-Zip (7) did not notably increase the number of drivers found; later, around July 16, many extractions were successful (8).

General Statistics of Pipeline Collection

The different gathering methods yielded a dataset comprising currently of 1'077'879 unique files. These files would require 1.42 TB of storage, but after cleanup, they occupy 951.3 GB. Of those are 54.75 GB driver files. The average file size submitted to the pipeline is 1.38 MB, with a median size of 48.00 KB, the first quartile at 7.58 KB and the third quartile at 249.61 KB. Although the biggest file submitted is 348.92 MB, the 99th percentile is 26.18 MB. The average size for drivers is smaller at 1.28 MB, while they are larger in the median at 113.34 KB and in the first and third quartiles (51.31 KB and 436.5 KB). The architecture distribution of the collected drivers is as follows: 211 drivers are of the discontinued Intel Itanium family, 26'139 are of the Intel x86_64 architecture, and 16'325 of the Intel x86 architecture. There are 967 64-bit ARM drivers, and 57 are of the 32- or 16-bit ARM architectures.

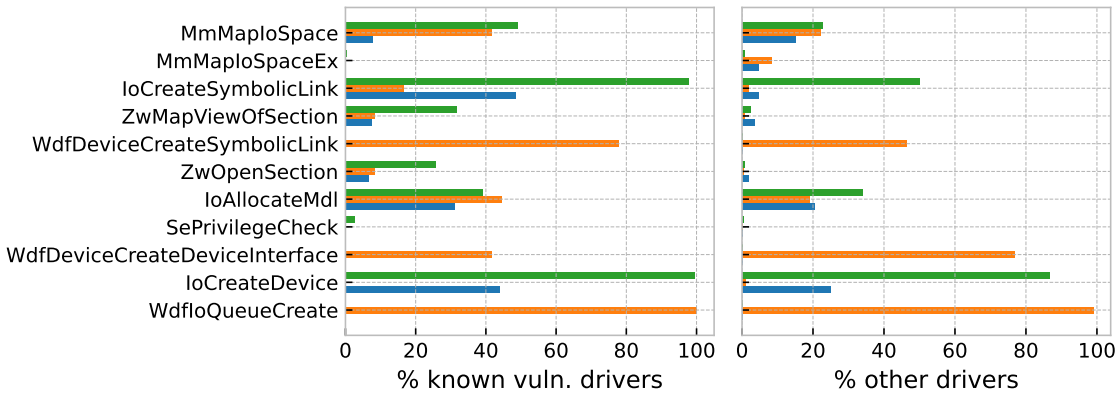


Figure 6.3: Usage of the 11 most interesting functions across known vulnerable and other drivers, differentiated by driver type: green for WDM, orange for WDF, and blue for unrecognised frameworks.

6.2 Tangible Effectiveness of Interesting Functions

This section demonstrates the efficacy of functions defined as interesting (Section 4.3.1) in identifying vulnerable drivers. Figure 6.3 presents two side-by-side graphs for the top 11 interesting functions. The left graph presents the percentage of known vulnerable drivers utilising each function. Conversely, the right graph shows the percentage of other drivers using the same functions. The colour coding distinguishes driver types: green for Windows Driver Model drivers, orange for Windows Driver Framework drivers, and blue for those where the framework is not recognised.

The distribution of these interesting functions indicates a higher presence in known vulnerable drivers than other drivers for most of the top 11. Notable exceptions include the `IoCreateDevice` and `WdfIoQueueCreate` functions, which are used in driver setup and are thus too widely used to be interesting. `SePrivilegeCheck`, used to verify user access rights dynamically, is more prevalent in vulnerable WDM drivers but appears in only 2.6% of all drivers, similarly having limited utility as an indicator.

`MmMapIoSpace`, often exploited for physical memory access, appears in roughly 50% of known vulnerable drivers, compared to less than 20% of non-vulnerable drivers. Interestingly, drivers with unrecognised frameworks show fewer instances of this function among known vulnerable drivers. This discrepancy is challenging to attribute to a specific factor. Still, it may be due to unrecognised framework drivers exhibiting strange behaviour, which also accounts for their functions not being recognised. Conversely, the extended function `MmMapIoSpaceEx` is less prevalent in known vulnerable drivers, possibly because developers with better security awareness use it or because it was excluded from much previous research.

`ZwOpenSection` and `ZwMapViewOfSection`, which enable physical memory manipulation, are more frequently found in known vulnerable drivers. This is likely because improper handling of these functions can lead to exploitation, making them targets for explicit searches. `IoCreateSymbolicLink` and `WdfDeviceCreateSymbolicLink` are heavily represented in known vulnerable drivers due to their role in kernel driver interactions from user-space applications.

`IoAllocateMdl`, used to map memory locations for data transfer, is likely more common in known vulnerable drivers because wrong use allows for arbitrary kernel memory write from user space. Lastly, `WdfDeviceCreateDeviceInterface` is less frequently found in vulnerable drivers, potentially due to the increased complexity of dynamic device interfaces requiring better driver knowledge for exploitation or because this type of driver was not as extensively considered in prior research.

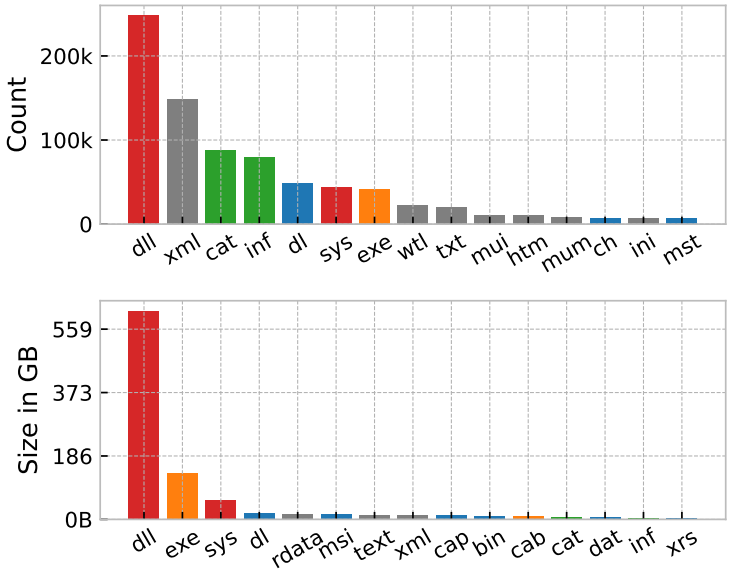


Figure 6.4: Top 15 file extensions ranked by count (top) and cumulative size (bottom). Colours indicate the Housekeeper's handling of these files: green = never delete (potentially valuable information), grey = always delete (no value to the pipeline), red = potentially interesting (should not contain drivers), blue = not yet separately handled (do not contain drivers).

6.3 Benefit of Housekeeper Tasks

The Housekeeper module performs two primary functions: reducing storage requirements by cleaning unnecessary files and extracting known archives. This section discusses the measured benefits these tasks provide to the pipeline.

Figure 6.4 illustrates the top 15 file extensions by the number of files and their cumulative storage size. Notably, there is a significant skew towards the first few file extensions, with 73% of all extensions by count and 64% by size represented in this figure. Short descriptions of the uncommon file extensions mentioned are provided in Appendix A.1. The colours indicate the Housekeeper module's actions for each file type:

- Gray depicted extensions are deleted immediately as they are unlikely to contain drivers or relevant information (e.g., `.xml`, `.txt`, `.htm`).
- Green marked extensions are preserved due to their relevance; for instance, `.inf` files may define SDDL strings for device installation, and `.cat` files contain signatures.
- Red extensions, though not expected to contain drivers, are retained for potential further inspection.
- Orange extensions, such as Cabinet Files and Executables, are extracted using cabextract and 7-Zip, respectively.
- Blue extensions are not specifically handled; for example, some `.dl` files, though generally unrecognised, can sometimes be identified as Cabinet Files and successfully extracted.

1'024'259 unique files were extracted, roughly 134 thousand of them using 7-Zip. These extractions submitted over 90% of all pipeline drivers.

6.4 Pathfinder Modification Evaluation

The modifications to the Pathfinder module can be categorised into those allowing it to handle additional drivers and those intended to speed up execution. This section presents the increased number of handled drivers and then evaluates the speedup improvements.

The pipeline has run the Pathfinder module across 967 ARM64 drivers, representing 3.9% of all drivers and 94.4% of all ARM drivers. Of these, 54 drivers are deemed attractive for vulnerability verification by security researchers. The added features for detecting more IOCTL handlers via loops or `memset64` identified an additional 89 and 1932 drivers, respectively. The low number of drivers detected using loop detection is expected, as such loops are specific to ARM drivers, and not all loop types are yet supported. The WDF functions completion feature identified 270'529 function calls across 3966 drivers. Among the 356 distinct WDF functions, `WdfDeviceCreateSymbolicLink` and `WdfDeviceCreateDeviceInterface` allowing driver interaction from user space were found in 3266 of these drivers.

Two modifications to the original propagation script aimed at speed improvements involved replacing debug conditions and caching trees that had already been visited. We tested 40 randomly sampled drivers from the database with an IOCTL handler, thus exercising the modified code sections. For each driver, IDA execution was run 30 times on a VM with 4 Intel(R) Core(TM) i7-9700K CPUs, with only the operating system and evaluation code running. A maximum timeout of 30 minutes per run was set to ensure timely completion. However, only two out of three WDF drivers completed within this time limit. A Wilcoxon–Mann–Whitney U test was used to evaluate the significance of median execution time improvements per driver, following the procedure outlined in Figure 2 of [78]. Of the 39 drivers, 53.3% showed significant improvement according to the statistical test, with a cumulative speedup of 5.13% compared to the same drivers without modifications.

6.5 Automated Fuzzing Results

Although the Fuzzifier module supports only AMD64 drivers, the fuzzing queue contains drivers for ARM and AMD 64-bit architecture drivers. Specifically, 4047 AMD and 111 ARM drivers were added to this queue by the pipeline. Of those feasible for fuzz testing with kAFL, 5275 configurations were attempted; 5193 failed for various reasons, and 18 successfully fuzzed a driver. This high failure rate is not unexpected, as most drivers require specific environmental setups that cannot be fully automated, particularly when specific hardware is needed. Furthermore, the symbolic name extraction via static analysis is more error-prone than initially expected, as 19.4% of fuzzing configurations contain invalid characters for symbolic names, such as $\%s$ or $\%w$. Despite the large proportion of invalid configurations, they are discarded rapidly, with 75% being rejected within 2.2 minutes.

Six of the 18 successfully fuzzed configurations yielded a timeout and ten crashing payloads. A driver's average and median fuzzing times were 14.1 and 6 hours, respectively. The ten drivers for which crashing payloads were identified are listed in Table 6.2. Four of these were known vulnerable drivers, added to the fuzzing queue before the adjustment to exclude such drivers.

Crashes in the known vulnerable drivers were not investigated, as these drivers should be blocked, and finding further vulnerabilities adds no additional value. Although the Fuzzifier identified three distinct IOCTL codes that crash CtiAIo64 and three that crash two versions of amdfendr, all these payloads result in non-exploitable null dereferences. The fuzzer also found a payload for biontdrv that invokes `HalReturnToFirmware(3LL)`, which crashes the system because it attempts to reboot. The second payload for the biontdrv driver and both for the DDDriver cause a physical memory read failure due to a no access error. Finally, the single crash payload for the NVFLASH driver resulted from accessing an uninitialised memory location within the driver state, which needs initialisation via another IOCTL interaction.

Driver Name	Tag	IOCTL Codes	Crash Payloads
PMXDRV.sys	Known Vulnerable	22	22
AIDA64Driver.sys	Known Vulnerable	4	4
AIDA64Driver.sys	Known Vulnerable	3	3
EBIoDispatch.sys	Known Vulnerable	1	1
CtiAIo64.sys	Not Vulnerable	3	3
amdfendr.sys	Not Vulnerable	3	4
amdfendr.sys	Not Vulnerable	3	3
biontdrv.sys	Vulnerable	2	2
DDDriver.sys	Vulnerable	1	2
NVFLASH64.sys	Vulnerable	1	1

Table 6.2: Table of unique drivers for which fully automated fuzz testing found crashing payloads. The "Not Vulnerable" tagged drivers are non-exploitable bugs.

6.6 Efficiency and Effectiveness of Pipeline

This thesis aimed to enhance the efficiency and effectiveness of Windows kernel driver vulnerability research by developing a system that assists security researchers with repetitive and automatable aspects of their work. The terms efficient and effective have diverse interpretations across different disciplines and are frequently not well-defined [87]. Generally, efficiency refers to how well a process uses resources to achieve its goals. It is typically measured as a ratio between input and output, where time and cost are often the critical resources. Conversely, effectiveness pertains to the extent to which the process achieves its intended goal.

In identifying vulnerable drivers, the most expensive input is the time security researchers spend reviewing drivers. Therefore, increased efficiency can be defined as the ability to verify more vulnerable drivers within the same time frame. Measuring effectiveness is more challenging since neither a customer who can evaluate the result nor a tangible product to evaluate exists. However, the number of vulnerabilities found can serve as a meaningful metric, given the goal of identifying and disclosing as many vulnerabilities as possible.

Due to time constraints, including other researchers in evaluating this pipeline's efficiency gains was not feasible. Still, the author dedicated approximately 36 hours to manually inspecting drivers identified by the pipeline as likely vulnerable. During this period, around 140 drivers were inspected. Exact numbers were not recorded as the evaluation was not the primary focus. This equates to roughly 15 minutes per driver invested using the pipeline. Hence, searching for drivers, verifying their vulnerability status, and inspecting them for vulnerabilities are reduced to 15 minutes only for the most likely candidates. More security researchers would need to use the pipeline to calculate the efficiency improvement compared to performing these tasks without the pipeline. Before creating the pipeline, though, the author spent about an hour or more per driver manually going through these tasks. [2]

The effectiveness of the pipeline can be evaluated in two ways: the number of new vulnerable drivers discovered and the number of known vulnerable drivers that would have been identified. Out of approximately 140 manually inspected drivers flagged by the pipeline as most likely vulnerable but not previously known to be vulnerable, 16 were found to have at least one vulnerability. Of these, ten drivers were previously unknown. Additionally, since this manual inspection was conducted when roughly 5000 drivers were in the pipeline, and subsequent gathering increased this number to over 40 thousand, more vulnerabilities are likely to be discovered with further manual verification. If the rest of the drivers follow a similar distribution to the inspected drivers, this would indicate over 100 unknown or unblocked vulnerable drivers.

Interestingly, the pipeline did not filter out six known vulnerable drivers, marking them as

[2] Although not scientific, this would equate to a speedup of roughly 4x.

likely vulnerable, and hence, they were manually analysed again. Three versions of the TP-PWRIF driver, part of a Lenovo Power Manager, had publicly disclosed vulnerabilities [33] but were not included in any sources of vulnerable drivers collected by the pipeline, such as the Microsoft recommended driver blocklist. One of these versions was even found running on multiple endpoints within a client network. Two other known vulnerable drivers were RamCaptureDriver64, which allows user-space programs to read physical memory by design, and AsusBSItf, an Asus BIOS flash driver. The Asus driver was only recently published as vulnerable [62], while RamCaptureDriver64 is part of a forensic tool, which might explain why it is not part of the recommended blocklist. The last of these six drivers, WiseHDInfo, is a hard disk analysis tool driver included in the Windows recommended driver blocklist. However, it is blocked via its certificate, which the pipeline does not yet check.

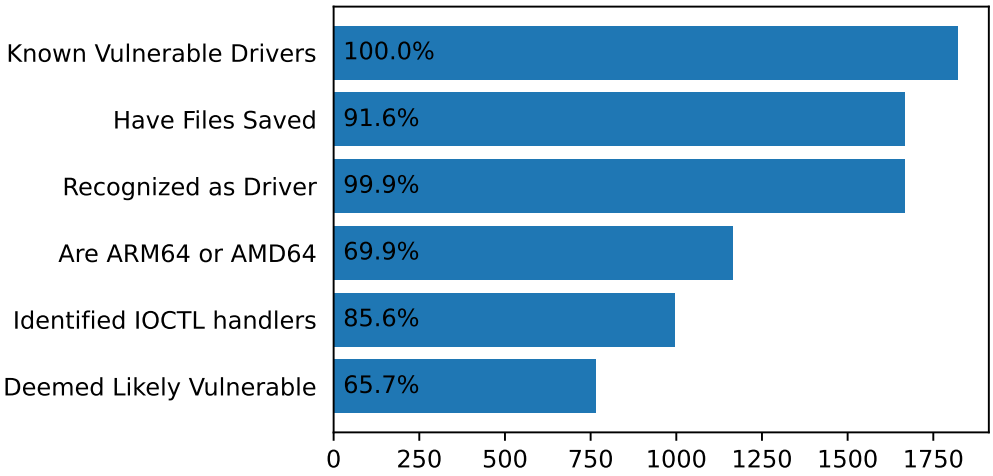


Figure 6.5: The number of drivers known to be vulnerable and the number of those recognised by the pipeline in different stages. The percentage shown is always relative to the amount of drivers directly above.

The second measurement of effectiveness involves assessing how many known vulnerable drivers the pipeline deems likely to be vulnerable and would thus be inspected by a researcher if they were unknown. Figure 6.5 shows at the top the number of drivers collected from known vulnerable driver sources. Each subsequent bar represents the number of drivers as a percentage relative to the preceding one. For instance, the pipeline has the actual files for only 91.6% of the known vulnerable driver's list, as the Microsoft recommended block list provides only hashes without files, and so does loldrivers.io for a subset of its drivers. The pipeline recognises all these files except two as drivers; ExifTool identifies the two outliers as more likely to be icon files. Of the 1668 drivers, 69.9% are either ARM64 or AMD64 architectures, which are the only ones currently supported by the Pathfinder module due to the licensing restrictions and missing support for 32-bit. In those, the Pathfinder can identify a valid IOCTL handler in 85.6%. Consequently, the pipeline deems 765 out of the 1668 known vulnerable drivers as likely vulnerable, resulting in an overall effectiveness of 45% and 65% when considering only 64-bit architectures. While 45% may seem low, most modern drivers are written for 64-bit architectures, making extending the pipeline for 32-bit architectures less worthwhile. Although only 65% of potentially vulnerable drivers were deemed likely vulnerable, security researchers would have prioritised these 765 drivers for inspection, leading to effective results for the invested time.

Driver Name / Product Name	Signature Year	Vulnerabilities
SysInfoDetectorPro / -	2014	rPM, wPM
SysInfoDetectorProX64 / -	2014	rPM, wPM
DVALZ / -	2022	rPM
biontdrv / -	2018	rPM
QQPCHW / QQ Computer Manager	2024	rMSR, wMSR, inB, outB
DellProf / Dell System Analyzer Control	2017	rMSR, wMSR, inB, outB
POADrvr / DPO Control System	2012	rMSR, wMSR
DDDriver / Dell Diagnostics	2014	inB, outB, rPM
LDiagIO / LSI MegaRAID controller	2020	inB, outB, rMSR
NVFLASH64 / NVIDIA NVFlash	2015	inB, outB

Table 6.3: Summary of previously unknown vulnerable drivers discovered using the developed pipeline. Note that not all drivers deemed likely vulnerable have been inspected. (rPM = Read Physical Memory, wPM = Write Physical Memory, rMSR = Arbitrary Read MSR, wMSR = Arbitrary Write MSR, inB = Arbitrary I/O Port Read, outB = Arbitrary I/O Port Write)

6.6.1 Identified New Vulnerable Drivers

Table 6.3 presents the ten drivers that were found to have vulnerabilities and were not previously publicly known to be vulnerable. These drivers serve various intended purposes, but since some were collected through VirusTotal, their associated products are not always obvious. The SysInfoDetector drivers and the two Dell drivers (DellProf, DDDriver) are part of general information-gathering or diagnostic tools, likely used by administrators or support personnel to assist users. The DVALZ, POADrvr, and LDiagIO drivers appear to be specific device drivers for different hardware associated with Dynabook Inc., Dell, and Lenovo computers, respectively. biontdrv is a Paragon Software GmbH's Backup and Recovery tool driver, while NVFLASH64 is a GPU BIOS flashing tool by the NVIDIA Corporation. Lastly, QQPCHW is likely a driver for a Tencent PC management tool, which also includes an anti-virus.

Five drivers permit kernel memory access and allow arbitrary physical memory reads; two even allow arbitrary physical memory writes. Among the ten drivers, three enable reading from and writing to any model-specific register (MSR). Furthermore, five drivers allow any user-space process to send bytes to any I/O port.

As a result, the five drivers allowing physical memory or MSR writes present a risk of local privilege escalation under the condition that they are active and that mitigations are bypassed in the case of MSR writes. For the 64-bit version of the SysInfoDetector, a Proof-of-Concept local privilege escalation exploit using the well-known Token Stealing technique was written (see Appendix A.2 for details). Similarly, those five allowing for physical memory read are prime targets for the Bring Your Own Vulnerable Driver (BYOVD) technique [76] to bypass defences. The five drivers permitting arbitrary bus communication (inB, outB) can be combined with memory-mapped IO to install firmware rootkits. However, this requires interaction with Serial Peripheral Interface (SPI) controllers, which demands a significantly higher level of attacker sophistication than other exploitation techniques like token swapping.

A coordinated vulnerability disclosure process was initiated for these ten drivers with new vulnerabilities. The initial interaction with vendors included the following elements: an explanation for inspecting these drivers, i.e., this master thesis, a list of each identified vulnerability along with the potential impact on a Windows system, and a recommendation to incorporate the driver into Microsoft's recommended driver block rules. Additionally, it was indicated that if the report were not acknowledged, the authors would report the driver to Microsoft Security Intelligence [26] for inclusion in the block rules.

Chapter 7

Discussion

In this chapter, we begin by discussing the difficulties encountered while working with the underlying tools during implementation. We then examine the general and inherited limitations of the current pipeline implementation. Subsequently, we address the delays in the coordinated vulnerability disclosure process and consider the implications of this work for companies striving to defend against sophisticated attackers. The chapter concludes with a section outlining potential avenues for future work.

7.1 Encountered Difficulties During Implementation

This section discusses the difficulties encountered during the pipeline implementation, providing insights for future work to avoid similar issues and consider essential factors early on.

Implementation Design

Although it is commonly known, it is often overlooked how quickly implementations can become unmanageable and incomprehensible, even to the original developers, if a straightforward design is not followed from the outset. Despite an early definition of the pipeline design, the source code for the Coordinator has become difficult to read due to varying requirements for additional modules and extensions. Although Python is a simple language and allows rapid development, it is not ideal for maintaining such a central pipeline component, especially when the API definition is consolidated into a single source file.

Another aspect to consider during the design phase is the difficulty of dockerizing tools within a Microsoft Windows environment while running on virtualized hardware. This is primarily due to Docker for Windows requiring hardware virtualization, which the provided VMware ESXi does not support.

Database Schema to Implementation

Tools like Database Markup Language (DBML) [14] are available to design a database schema. While DBML is helpful for initial schema design, it is less effective when used with an object-relational mapping system like SQLAlchemy, even with translation tools. This is particularly true for non-trivial SQL features not supported by the translation due to their complexity. As such, part of the pipeline DBML requires manual translation to the ORM, thereby reducing its usability beyond the initial design phase.

After deciding on a database schema implementation, another crucial decision is the choice of the database backend. The principle of getting it to work first and then making it robust does not apply well to the DB backend. Initially, SQLite was used for its simplicity, but once all modules were implemented, it could not handle all concurrent requests. Migrating to

PostgreSQL was more labour-intensive than starting with it directly, especially considering how easy it is to set up a PostgreSQL container.

Complications for Fuzzifier

The Fuzzifier module was planned to have an AMD64 and an ARM64 instance. While the ARM64 version is not functional, the AMD64 version's performance is disappointing, especially considering the substantial computing resources it requires.

Although kAFL, the fuzzing framework used in the Fuzzifier, supports Windows kernel drivers, it is challenging to use and automate. The provided Ansible scripts are helpful for manual use but do not consistently succeed. They sometimes fail due to unchecked environmental changes, such as operating system updates or imperfect cleanup of previous fuzzing targets. Additionally, kAFL required a bug fix to avoid heap overflows during fuzzing.

Despite these issues, kAFL is a viable framework for fully automated AMD kernel fuzzing, whereas ARM fuzzing frameworks are much less advanced. Although fuzzing on ARM is of interest, particularly for embedded devices and smartphones, most frameworks work only within their specific domains, and none include Windows. The closest project to ARM kernel driver fuzzing is ARMored CoreSight [59], which implements the kAFL hardware tracing concept using ARM CoreSight. However, CoreSight's reliance on optional ETMv4 [34] features means very specific hardware is required, posing a significant challenge. Even with the correct hardware, existing examples could not be reproduced. Moreover, ARMored CoreSight lacks support for ARM instructions used in kernel space, limiting its applicability to kernel-level fuzzing.

Regardless of the availability of fuzzing frameworks, fully automated fuzzing of kernel drivers has additional challenges due to the complexity of interacting with them. Drivers are typically designed to interact with their accompanying user-space programs, so they do not adhere to a robust public interface. Finding the correct symbolic name, the proper layout for input/output buffers and managing multiple interactions with a driver towards a single goal are only some of the complexities an automated system has to find, while the accompanying programs do not. This complexity likely contributed to the limited success of the Fuzzifier module.

7.2 Limitations of Current Pipeline

The current implementation of the pipeline exhibits several limitations, primarily due to the constraints of the underlying tools, such as a lack of support for specific architectures and the limitations imposed by licensing requirements. Each module has further specific limitations inherited from their underlying tools, listed in Table 7.1.

7.3 Friction in Coordinated Vulnerability Disclosure Process

The coordinated vulnerability disclosure process for the ten unknown vulnerable drivers listed in Section 6.6.1 was delayed multiple times but began on the 10th of July, 2024. This postponement was partially due to the pipeline's ongoing development. While identifying additional drivers at later stages allowed for a more streamlined acceptance process within InfoGuard AG, internal company bureaucracy also contributed to a slight delay. While aggregating the drivers and reporting them collectively proved advantageous for the company's acceptance processes, immediate reporting upon discovery would have been more beneficial to the broader community as the total delay would have likely been shorter for the initially found drivers.

Although all vulnerabilities are for Windows third-party kernel drivers, vulnerability reporting differs between vendors. Prominent vendors such as Tencent, Dell, and NVIDIA prefer researchers create accounts on bug bounty websites to submit reports. Such websites understandably streamline the coordinated vulnerability disclosure process but require additional

Module	Description of Limitation
Importers	Difficulty in locating niche drivers not distributed via Windows updates. They are unlikely to be promptly added to VirusTotal; hence, they would not be gathered from there.
Housekeeper	Restricted to extracting Cabinet Files and executables with 7-Zip-compatible compressions because UniExtract2, which theoretically supports extraction from various installers, was not possible to get working.
Identifier	Files neither recognized by ExifTool as a Windows Executable nor having a filename ending in `.sys` are not recognized as drivers.
Certificator	The sigcheck tool fails to recognize the revocation of a small set of drivers. There is no documentation for these specific cases or for manual use of the WinVerifyTrust API to implement the missing checks.
Pathfinder	Whenever IDA Pro fails to decompile drivers correctly, the modified TAU script will produce inaccurate or incomplete results. Already encountered issues were incomplete results when ARM64 functions are decompiled with incorrect arguments or not recognized memory functions such as `RtlMemoryCopy` or `RtlMemoryMove` due to the absence of FLIRT signatures [66] in IDA Pro.
Fuzzifier	Its most significant limitations are already mentioned in Section 7.1, which restrict its functionality to only AMD64 kernel drivers and those for which a valid symbolic name string was extracted.

Table 7.1: Per module summary of limitations in the current pipeline implementation.

overhead for researchers to set it up for the different vendors. Of the five reports submitted through websites, four responded within two weeks. On the other hand, of the four companies contacted by mail, only two responded to the disclosure after three weeks. Both acknowledged the receipt of the report and mentioned an internal case was opened. One of the vendors that responded closed the case with "Not Applicable", citing the obsolescence of the reported drivers. This highlights a further issue within the current state of Windows security, as these outdated drivers remain exploitable and will not be fixed.

7.4 Implications of Results

The developed pipeline and its findings illustrate broader issues within the current state of Windows kernel driver security, with significant implications for companies defending against highly sophisticated adversaries. These issues include the blocklisting approach to vulnerable drivers, reports being disregarded because of driver obsolescence, the low prioritization of publicly disclosed vulnerabilities, and the lack of centralized access to all drivers for third-party verification.

As expected with a blocklisting approach, some drivers inevitably bypass the blocklist due to oversight or because they are newly released. This problem is exacerbated when blocking is based on driver hash or filename, like for the recommended driver blocklist, as minor changes, including renaming, generate a new hash and bypass the blocklist.

A related issue is the low prioritization of driver vulnerabilities due to the low impact scores assigned to them during publication. This results in delayed fixing and them not being included in the Microsoft recommended driver blocklist. The five drivers highlighted in Section 6.6, which were unrecognized by the pipeline because they are not blocked despite being known to be vulnerable, underscore this problem. Even such publicly known vulnerabilities remain exploitable if not blocked, attracting threat actors' interest.

The implication for companies defending against sophisticated threat actors on the level of nation-states (see Section 3.1 for categorization) is clear: the current blocklisting approach is ineffective. It should be replaced with an allowlisting approach. This could involve signer allowlisting for well-known vendors, like Microsoft, and more selective hash-based allowlisting for all third-party drivers. The latter would be closely curated and inspected for disclosed vulnerabilities regardless of their associated impact score.

Regardless of the blocking method, checking new and existing drivers for vulnerabilities is essential. Although Microsoft claims to perform this automatically and offers a submission form for reporting vulnerabilities, these measures seem insufficient. Additionally, third parties lack straightforward access to all drivers or notifications of new releases to assist in vulnerability checks. Therefore, this work emphasizes the need for Microsoft to provide such a service or for the community to develop a system allowing security researchers affordable access to such a list of drivers.

7.5 Future Work

This initial pipeline implementation demonstrates its potential utility in Windows kernel driver vulnerability research, though it has much room for improvement.

Future work on the pipeline could involve broadening the scope of driver collection. This could be achieved through running the VirusTotal gathering permanently and the inclusion of other sources such as leaked malware developer driver lists [11] or possibly even Windows Anti-Cheat blocking lists like those of Valorant's Vanguard [85]. Supporting more extraction tools, such as through UniExtract2 or more specialized ones like lessmsi [73] would also be beneficial.

Modules such as the Pathfinder could be improved to further enhance driver analysis by extending ARM loop detection and incorporating missing FLIRT signatures. Additionally, integrating more modules based on previous work, such as IOCTLance [86], could further improve the automated analysis of the pipeline.

Although extending the Fuzzifier module to support ARM64 could allow for dynamic analysis of ARM drivers, the limited results from the AMD64 version suggest that this should be a lower priority than other improvements. Other potential future modules include testing drivers under XDRs to identify further dynamic blocking conditions and a monitoring module to maintain the pipeline's continuous operation by detecting errors and failures.

Regarding the collected driver set, future work involves further inspecting the likely deemed vulnerable drivers of the pipeline and utilizing the already calculated ssdeep hash with techniques described in ssdeeper [28] to create a similarity matrix. This matrix should be able to identify drivers closely related to known vulnerable drivers.

Lastly, developing a public API for the compiled driver list could allow anyone to check their installed drivers and enable security researchers to utilize it in their analyses without repeating the entire data collection process.

Chapter 8

Conclusion

This thesis presents the design and implementation of a comprehensive pipeline consisting of eight modules aimed at automating the process of gathering and analyzing Windows kernel drivers for vulnerabilities. The pipeline collects drivers from various sources, utilizes existing vulnerability detection tools, and extends their applicability to include ARM64 drivers and more AMD64 drivers. The results are aggregated and presented in a user-friendly interface, enabling security researchers to identify and prioritize potentially vulnerable drivers efficiently.

The development of this pipeline revealed that while most tools function as intended, their diverse requirements — ranging from specific operating systems to necessary hardware features for fuzzing — pose challenges for a unified implementation. This is particularly evident in the emerging Windows ARM64 space, where existing tools often cannot be directly applied. Secondly, the absence of a single, comprehensive source for Windows drivers necessitates the integration of multiple data sources to get a set of drivers for analysis. Despite leveraging only three sources and inspecting only roughly 7% of the likely vulnerable drivers, it was possible to identify previously unknown vulnerabilities in ten unique drivers, for one of which a usable exploit was written.

The successful identification of these vulnerabilities led to the initiation of coordinated disclosure processes, emphasizing the pipeline's practical utility. These findings, coupled with Microsoft's current blocklist approach, suggest that companies defending against highly sophisticated threat actors may need to adopt an allowlist approach, instead despite the significant workload involved with it.

In conclusion, this thesis demonstrates the potential of automated pipelines unifying previous work on identifying vulnerabilities in Windows kernel drivers. Addressing the challenges and limitations identified and pursuing the suggested future directions can enhance the security of the Windows kernel landscape, contributing to a safer and more secure business environment.

Bibliography

[1] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence". In: 2019. URL: https://api.semanticscholar.org/CorpusID:85546717.

[2] Mantvydas Baranauskas. *Token Abuse for Privilege Escalation in Kernel*. 2020. URL: https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/how-kernel-exploits-abuse-tokens-for-privilege-escalation (visited on 07/08/2024).

[3] Michael Bayer. "SQLAlchemy". In: *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. Ed. by Amy Brown and Greg Wilson. aosabook.org, 2012. URL: http://aosabook.org/en/sqlalchemy.html.

[4] Biswapriyo. *How to make type libraries from Windows 10 SDK and DDK?* July 12, 2018. URL: https://reverseengineering.stackexchange.com/a/18785 (visited on 04/12/2024).

[5] blacktop. *Malice PExecutable Plugin*. 2018. URL: https://github.com/malice-plugins/fileinfo (visited on 06/21/2024).

[6] blacktop, jakewarren, and Tomer Frid. *Malice File Info Plugin*. 2018. URL: https://github.com/malice-plugins/pescan/blob/master/malice/__init__.py (visited on 06/21/2024).

[7] Jordan Borean. *get_microsoft_updates.py*. Dec. 9, 2019. URL: https://gist.github.com/jborean93/8bd09a3314311c78fe2939b88bb82f4f (visited on 06/05/2024).

[8] Zac Bowden and Rebecca Spear. *pefile*. June 24, 2024. URL: https://www.windowscentral.com/hardware/laptops/best-windows-laptops-with-arm-processor (visited on 07/18/2024).

[9] Stuart Caie et al. *cabextract*. Mar. 3, 2023. URL: https://www.cabextract.org.uk/ (visited on 06/22/2024).

[10] Ero Carrera. *pefile*. 2023. URL: https://pypi.org/project/pefile/ (visited on 06/21/2024).

[11] Donncha O Cearbhaill. *Mirror of the Shadow Brokers dump*. Apr. 14, 2017. URL: https://github.com/DonnchaC/shadowbrokers-exploits/blob/master/windows/Resources/Ep/drv_list.txt (visited on 07/11/2024).

[12] Wikipedia contributors. *Phishing — Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Phishing&oldid=1235247621 (visited on 07/18/2024).

[13] Dominik Czarnota, NyaMisty, and Bet4. *pwndbg - decompile_context*. July 19, 2019. URL: https://github.com/pwndbg/pwndbg/blob/7dbc5/ida_script.py#L131 (visited on 06/24/2024).

[14] *DBML - Database Markup Language*. 2024. URL: https://github.com/holistics/dbml (visited on 06/20/2024).

[15] debasishm89. *iofuzz*. Mar. 16, 2014. URL: https://github.com/debasishm89/iofuzz (visited on 07/02/2024).

[16] eclypsium. *Screwed-Drivers*. Aug. 14, 2019. URL: https://github.com/eclypsium/Screwed-Drivers (visited on 03/21/2024).

[17] William Engelmann. *Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETM4.6*. Sept. 19, 2023. URL: https://github.com/Bioruebe/UniExtract2 (visited on 07/10/2024).

[18] Gergely Erdelyi and Elias Bachaalany. *IDAPython*. Feb. 2024. URL: https://github.com/idapython/src (visited on 06/28/2024).

[19] Google. *ioctlfuzzer*. 2009. URL: https://code.google.com/archive/p/ioctlfuzzer/ (visited on 04/23/2024).

[20] Rajat Gupta et al. "POPKORN: Popping Windows Kernel Drivers At Scale". In: *Proceedings of the 38th Annual Computer Security Applications Conference*. ACSAC '22. Austin, TX, USA: Association for Computing Machinery, 2022, pp. 854–868. ISBN: 9781450397599. DOI: 10.1145/3564625.3564631. URL: https://doi.org/10.1145/3564625.3564631.

[21] Michael Haag, Jose Hernandez, and Nasreddine Bencherchali. *Living Off The Land Drivers*. 2023. URL: https://www.loldrivers.io/ (visited on 06/20/2024).

[22] Takahiro Haruyama. *Hunting Vulnerable Kernel Drivers*. Oct. 31, 2023. URL: https://blogs.vmware.com/security/2023/10/hunting-vulnerable-kernel-drivers.html (visited on 06/25/2024).

[23] Phil Harvey. *ExifTool*. 2024. URL: https://exiftool.org/#ack (visited on 06/21/2024).

[24] holistics. *Draw Entity-Relationship Diagrams, Painlessly*. 2024. URL: https://dbdiagram.io/ (visited on 06/20/2024).

[25] IBM. *Types of threat actors*. 2024. URL: https://www.ibm.com/topics/threat-actor (visited on 06/05/2024).

[26] Microsoft Security Intelligence. *Submit a driver for analysis*. URL: https://www.microsoft.com/en-us/wdsi/driversubmission (visited on 07/08/2024).

[27] Christopher Ireland et al. "A Classification of Object-Relational Impedance Mismatch". In: *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications*. 2009, pp. 36–43. DOI: 10.1109/DBKDA.2009.11.

[28] Carlo Jakobs, Martin Lambertz, and Jan-Niclas Hilgert. "ssdeeper: Evaluating and improving ssdeep". In: *Forensic Science International: Digital Investigation* 42 (2022). Proceedings of the Twenty-Second Annual DFRWS USA, p. 301402. ISSN: 2666-2817. DOI: https://doi.org/10.1016/j.fsidi.2022.301402. URL: https://www.sciencedirect.com/science/article/pii/S266628172200083X.

[29] Mohamed Khalifa and Mona Albadawy. "Using artificial intelligence in academic writing and research: An essential productivity tool". In: *Computer Methods and Programs in Biomedicine Update* 5 (2024), p. 100145. ISSN: 2666-9900. DOI: https://doi.org/10.1016/j.cmpbup.2024.100145. URL: https://www.sciencedirect.com/science/article/pii/S2666990024000120.

[30] Andreas Klopsch and Matt Wixey. *It'll be back: Attackers still abusing Terminator tool and variants*. Mar. 4, 2024. URL: https://news.sophos.com/en-us/2024/03/04/itll-be-back-attackers-still-abusing-terminator-tool-and-variants/ (visited on 07/18/2024).

[31] Loren Kohnfelder and Praerit Garg. *The threats to our products*. 1999. URL: https://www.first.org/global/sigs/cti/curriculum/The-Threats-To-Our-Products.docx (visited on 06/25/2024).

[32] koutto. *ioctlbf*. Mar. 13, 2017. URL: https://github.com/koutto/ioctlbf (visited on 07/12/2024).

[33] Lenovo. *Memory Leakage and Denial of Service Vulnerabilities Identified in Power Manager, Lenovo Settings Dependency Package and ThinkPad Settings Dependency*. Apr. 16, 2016. URL: https://support.lenovo.com/jp/en/solutions/ps500037-memory-leakage-and-denial-of-service-vulnerabilities-identified-in-power-manager-lenovo-settings-dependency-package-and-thinkpad-settings-dependency (visited on 07/06/2024).

[34] Arm Limited. *Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETM4.6*. Sept. 19, 2023. URL: https://ricercasecurity.blogspot.com/2021/11/armored-coresight-towards-efficient.html (visited on 07/10/2024).

[35] Alex Margarit. *Windows Driver Framework - WDFFUNCTIONS Definition*. Mar. 18, 2015. URL: https://github.com/microsoft/Windows-Driver-Frameworks/blob/3b9780e847cf68d6199dafe0f87650cf1f9c227f/src/framework/kmdf/inc/private/fxdynamics.h#L23 (visited on 06/24/2024).

[36] Microsoft. *API reference docs for Windows Driver Kit (WDK)*. Apr. 3, 2023. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ (visited on 06/05/2024).

[37] Microsoft. *ARM64 intrinsics*. Dec. 12, 2021. URL: https://learn.microsoft.com/en-us/cpp/intrinsics/arm64-intrinsics?view=msvc-170 (visited on 06/05/2024).

[38] Microsoft. *Buffer Descriptions for I/O Control Codes*. Dec. 15, 2021. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/buffer-descriptions-for-i-o-control-codes (visited on 07/21/2024).

[39] Microsoft. *Controlling Device Namespace Access*. Dec. 15, 2021. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/controlling-device-namespace-access (visited on 06/10/2024).

[40] Microsoft. *Defining I/O Control Codes*. Dec. 15, 2021. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/defining-i-o-control-codes (visited on 06/12/2024).

[41] Microsoft. *DeviceIoControl function (ioapiset.h)*. 2022. URL: https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol (visited on 07/10/2024).

[42] Microsoft. *DFfuzz*. URL: https://learn.microsoft.com/en-us/windows-hardware/test/hlk/testref/236b8ad5-0ba1-4075-80a6-ae9dafb71c94 (visited on 03/22/2024).

[43] Microsoft. *Driver Module Framework (DMF)*. 2018. URL: https://github.com/microsoft/DMF (visited on 07/10/2024).

[44] Microsoft. *Introduction to WDM*. Dec. 15, 2021. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-wdm (visited on 07/10/2024).

[45] Microsoft. *IoAttack*. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/ioattack (visited on 06/15/2024).

[46] Microsoft. *IRP Major Function Codes*. Mar. 14, 2023. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-major-function-codes (visited on 06/06/2024).

[47] Microsoft. *Manage driver access control*. Dec. 11, 2023. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/driversecurity/driver-security-checklist#manage-driver-access-control (visited on 06/10/2024).

[48] Microsoft. *memset, wmemset*. Feb. 12, 2022. URL: https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/memset-wmemset?view=msvc-170 (visited on 06/05/2024).

[49] Microsoft. *Microsoft Security Servicing Criteria for Windows*. URL: https://www.microsoft.com/en-us/msrc/windows-security-servicing-criteria#Non-boundaries (visited on 07/12/2024).

[50] Microsoft. *Microsoft Update Catalog - Frequently Asked Question*. 2024. URL: https://www.catalog.update.microsoft.com/faq.aspx (visited on 06/21/2024).

[51] Microsoft. *Recommended Driver Block Rules*. 2023. URL: https://learn.microsoft.com/en-us/windows/security/application-security/application-control/windows-defender-application-control/design/microsoft-recommended-driver-block-rules (visited on 07/02/2024).

[52] Microsoft. *SDDL for Device Objects*. Dec. 15, 2021. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/sddl-for-device-objects (visited on 06/10/2024).

[53] Microsoft. *Security Threat Models*. 2024. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/designing-with-security-threat-models (visited on 07/10/2024).

[54] Microsoft. *Threat modeling for drivers*. 2024. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/driversecurity/threat-modeling-for-drivers (visited on 07/10/2024).

[55] Microsoft. *Windows Driver Framework (WDF)*. Dec. 4, 2023. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/_wdf/ (visited on 06/05/2024).

[56] Microsoft. *wrmsr (Write MSR)*. Feb. 4, 2024. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/wrmsr--write-msr- (visited on 06/05/2024).

[57] Bart Miller. *Project List*. Tech. rep. COMPUTER SCIENCES DEPARTMENT UNIVERSITY OF WISCONSIN-MADISON, 1988. URL: https://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf (visited on 06/28/2024).

[58] Elizabeth Montalbano. *Kasseika Ransomware Linked to BlackMatter in BYOVD Attack*. Jan. 24, 2024. URL: https://www.darkreading.com/endpoint-security/kasseika-ransomware-linked-blackmatter-byovd-attack (visited on 07/18/2024).

[59] Akira Moroo and Yuichi Sugiyama. *ARMored CoreSight: Towards Efficient Binary-only Fuzzing*. Nov. 18, 2021. URL: https://ricercasecurity.blogspot.com/2021/11/armored-coresight-towards-efficient.html (visited on 07/10/2024).

[60] Tao Ni et al. "High-Coverage Security Testing for Windows Kernel Drivers". In: *2012 Fourth International Conference on Multimedia Information Networking and Security*. 2012, pp. 905–908. DOI: 10.1109/MINES.2012.117.

[61] Enrique Nissim. *Reverse Engineering and Bug Hunting on KMDF Drivers*. 2018. URL: https://www.youtube.com/watch?v=puNkbSTQtXY (visited on 06/10/2024).

[62] NIST. *CVE-2024-33221 Detail*. May 24, 2024. URL: https://nvd.nist.gov/vuln/detail/CVE-2024-33221 (visited on 07/06/2024).

[63] Inc. Open Systems Resources. *Making Device Objects Accessible ... and Safe*. Aug. 10, 2020. URL: https://www.osr.com/nt-insider/2017-issue1/making-device-objects-accessible-safe/ (visited on 06/10/2024).

[64] Igor Pavlov. *7-Zip*. June 19, 2024. URL: https://www.7-zip.org/ (visited on 07/21/2024).

[65] Marco Pontello. *TrID - File Identifier*. 2017. URL: https://mark0.net/soft-trid-e.html (visited on 06/21/2024).

[66] Hex Rays. *IDA F.L.I.R.T. Technology: In-Depth*. 2020. URL: https://hex-rays.com/products/ida/tech/flirt/in_depth/ (visited on 07/29/2024).

[67] Hex Rays. *IDA Pro*. 2024. URL: https://hex-rays.com/ida-pro/ (visited on 06/28/2024).

[68] ReWolf. *MSI ntiolib.sys/winio.sys local privilege escalation*. Sept. 26, 2016. URL: http://blog.rewolf.pl/blog/?p=1630 (visited on 07/10/2024).

[69] Mark Russinovich. *Sigcheck v2.90*. July 19, 2022. URL: https://learn.microsoft.com/en-us/sysinternals/downloads/sigcheck (visited on 06/21/2024).

[70] Steffen Schulz and Mathieu Tarral. *HW-assisted Feedback Fuzzer for x86 VMs*. 2024. URL: https://github.com/IntelLabs/kAFL (visited on 06/28/2024).

[71] Sergej Schumilo et al. "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels". In: *USENIX Security Symposium*. 2017. URL: https://api.semanticscholar.org/CorpusID:12778185.

[72] Sergej Schumilo et al. "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo.

[73] Willeke Scott. *lessmsi*. 2018. URL: https://github.com/activescott/lessmsi/releases (visited on 07/11/2024).

[74] shadcn. *shadcn/ui*. 2023. URL: https://github.com/shadcn-ui/ui (visited on 06/21/2024).

[75] Igor Skochinsky. *Igor's tip of the week #60: Type libraries*. Oct. 15, 2021. URL: https://hex-rays.com/blog/igors-tip-of-the-week-60-type-libraries/ (visited on 04/12/2024).

[76] Sangfor Technologies. *What is BYOVD? - BYOVD Attacks in 2023*. Sept. 15, 2023. URL: https://www.sangfor.com/farsight-labs-threat-intelligence/cybersecurity/what-is-byovd-attacks-2023 (visited on 07/18/2024).

[77] Maxime Thiebaut. *Generating IDA Type Information Libraries from Windows Type Libraries*. Nov. 7, 2023. URL: https://blog.nviso.eu/2023/11/07/generating-ida-type-information-libraries-from-windows-type-libraries/ (visited on 04/12/2024).

[78] Sid-Ahmed-Ali Touati, Julien Worms, and Sébastien Briais. "The Speedup-Test: a statistical methodology for programme speedup analysis and computation". In: *Concurrency and Computation: Practice and Experience* 25.10 (2013), pp. 1410–1426. DOI: https://doi.org/10.1002/cpe.2939. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.2939. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.2939.

[79] Vercel. *SWR - React Hooks for Data Fetching*. June 28, 2023. URL: https://swr.vercel.app/ (visited on 06/22/2024).

[80] VergiliusProject. *VERGILIUS _EPROCESS*. June 24, 2024. URL: https://www.vergiliusproject.com/kernels/x64/windows-11/21h2/_EPROCESS (visited on 07/21/2024).

[81] VirusTotal. *Files download through API*. 2023. URL: https://developers.virustotal.com/reference/files-download (visited on 07/10/2024).

[82] VirusTotal. *vt-py*. 2023. URL: https://github.com/VirusTotal/vt-py (visited on 05/18/2024).

[83] Iuliia Volkova. *O! My Models*. 2024. URL: https://github.com/xnuinside/omymodels (visited on 06/20/2024).

[84] Ryan Warns and Tim Harrison. *Device Driver Debauchery and MSR Madness*. 2020. URL: https://slideplayer.com/slide/17845751/ (visited on 07/16/2024).

[85] whatacoolwitch. *What is Vanguard?* Jan. 5, 2024. URL: https://support-valorant.riotgames.com/hc/en-us/articles/360046160933-What-is-Vanguard (visited on 07/11/2024).

[86] Zeze. *IOCTLance*. Nov. 9, 2023. URL: https://github.com/zeze-zeze/ioctlance (visited on 07/11/2024).

[87] Youcef J-T Zidane and Nils Olsson. "Defining project efficiency, effectiveness and efficacy". In: *International Journal of Managing Projects in Business* 10 (June 2017). DOI: 10.1108/IJMPB-10-2016-0085.

A

Appendix A

A.1 Uncommon File Extensions

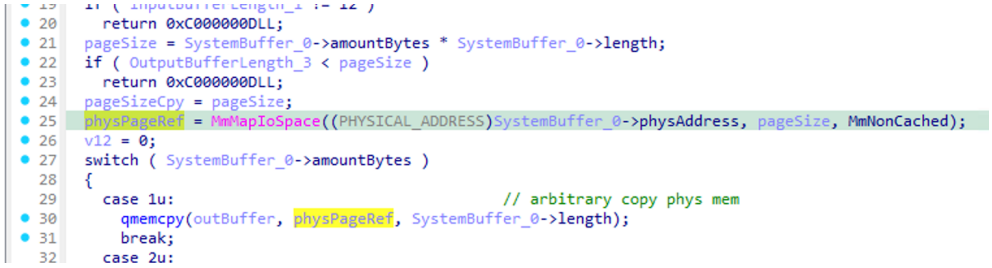
Some of the top 15 file extensions by count and size shown in Section 6.3 might not be known by everyone. As such, Table A.1 gives short descriptions of each.

Extensions	Short Description
`bin`	Generally binary data; some drivers are named `<MD5>.bin`
`cab`	Cabinet File provided by Windows Update Catalog.
`cap`	Multiple possible file types, most likely ASUS BIOS firmware.
`cat`	Microsoft Security Catalog; digital signatures for a file collection.
`chm`	Microsoft Windows HtmlHelp Data files.
`ch`	Microsoft SZDD compressed files containing HTML documents.
`data`	7-Zip extraction result for executables.
`dat`	Generic extension for data files.
`dl`	Microsoft SZDD compressed files; not useful when extracted with 7-Zip, though some are recognized as cab files and extracted.
`inf`	Text file containing information for device installation.
`mui`	Language extension files for Windows.
`mum`	Part of the Windows Update Package and generic XML files.
`rdata`	7-Zip extraction result for executables.
`text`	7-Zip extraction result for executables.
`wtl`	XML files defining some user interaction elements in Windows.
`xrs`	Not a known extension, all of them are PE executables.

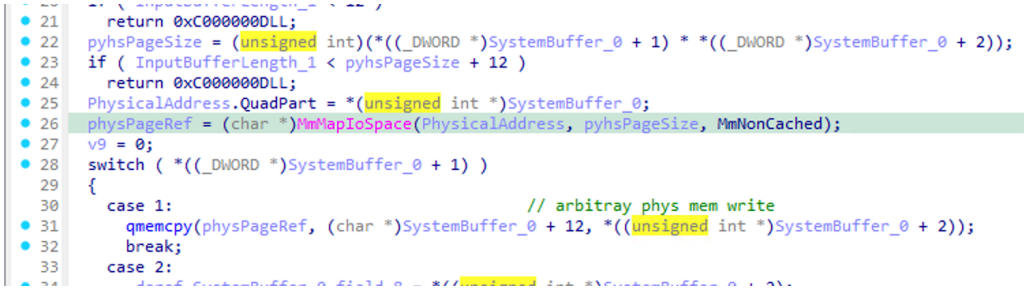
Table A.1: This table briefly describes uncommon file extensions in the top 15 files by count and size within the pipeline.

A.2 Token Stealing Exploit

This section provides details on the Proof-of-Concept exploit developed for the SysInfoDetectorPro driver. The well-known Token Stealing technique [2] is employed, as this driver enables arbitrary physical memory read and write operations accessible to any user-space application. This technique takes the security token saved in each process' `_EPROCESS` [80] structure from a privileged process and copies it into a user-space process, giving that process the same privilege level. The IDA Pro decompilation view of the two required operations within the driver can be seen in Figure A.1. The exploit was implemented in Rust to evaluate the compatibility of the language with Windows exploitation code and for its execution performance.



(a) Physical memory read vulnerability via memory-mapped I/O space at a user-defined address, followed by copying a user-defined size from this location to the output buffer.



(b) Physical memory writes vulnerability via mapping memory I/O space at a user-defined address and copying a user-defined size from the input buffer to this location.

Figure A.1: IDA Pro decompilation screenshots of the SysInfoDetectorPro vulnerabilities.

The exploit comprises four straightforward steps, with the initial step enhancing the process's efficiency:

1. The `Hardware\\ResourceMap\\System Resources\\Physical Memory` registry key is read to determine the memory ranges mapped into the operating system.
2. The driver is accessed via the symbolic name it establishes upon loading. [1]
3. The arbitrary read vulnerability is exploited to perform a linear search across all mapped memory regions for the *wininit.exe* and *cmd.exe* `_EPROCESS` structures in memory. This is achieved by searching for the four bytes writing *Proc*, followed by checking the string of the process name at a specific offset against the searched process name.
4. At a specific offset within this `_EPROCESS` structure, the relevant token is located and calculated using `let token_offset = (eprocess_offset + 0x4b8) as usize;`. This token is then copied from the *wininit.exe* structure to the *cmd.exe*, utilizing the driver's arbitrary write capability.

[1] Note that this exploit only elevates privilege if the driver is preloaded; however, it can escalate to NT/Authority System privileges even when administrative rights are granted, but the Sysinternals tools to elevate to NT/Authority System are restricted.

A.3 Frontender Website Screenshots

The Frontender module provides the user interface for interacting with the pipeline's results. It offers multiple views tailored to different tasks. The overview tabs displaying all or filtered drivers can be seen in Figure 5.5, while this section presents more views. Figure A.2 shows the driver card, presenting detailed information about a specific driver.

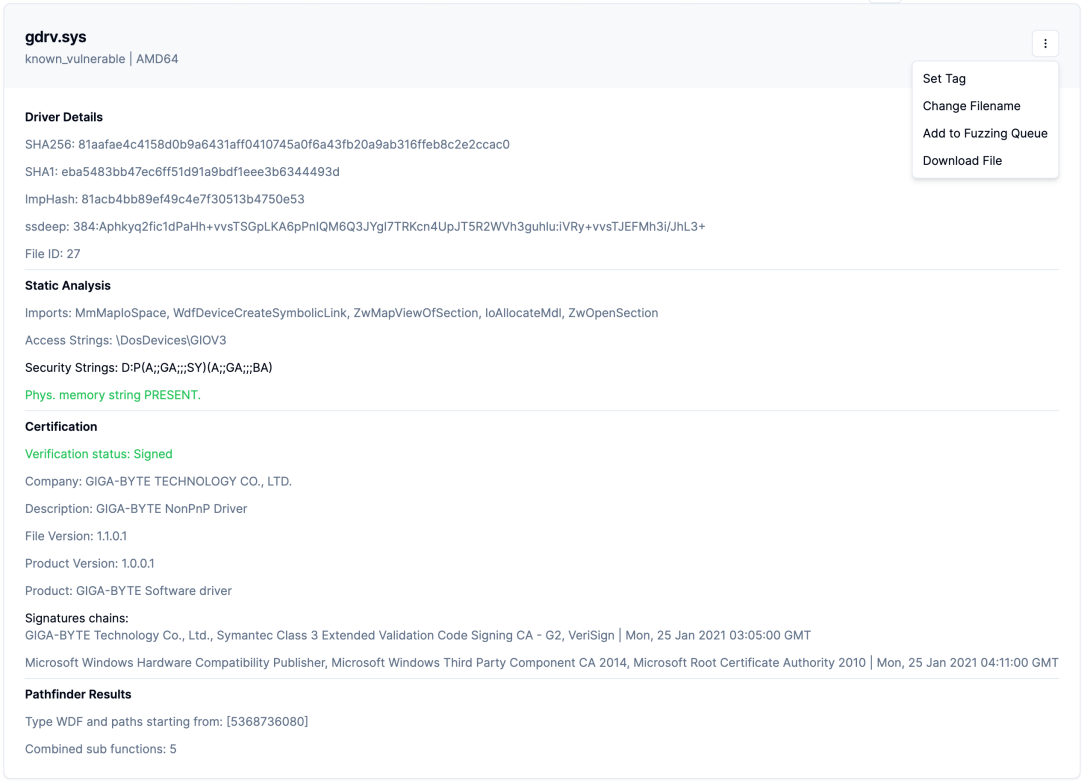


Figure A.2: The light-themed front-end view displays the analysis results of a single driver. This includes nearly all the information needed to decide whether a further manual inspection is interesting.

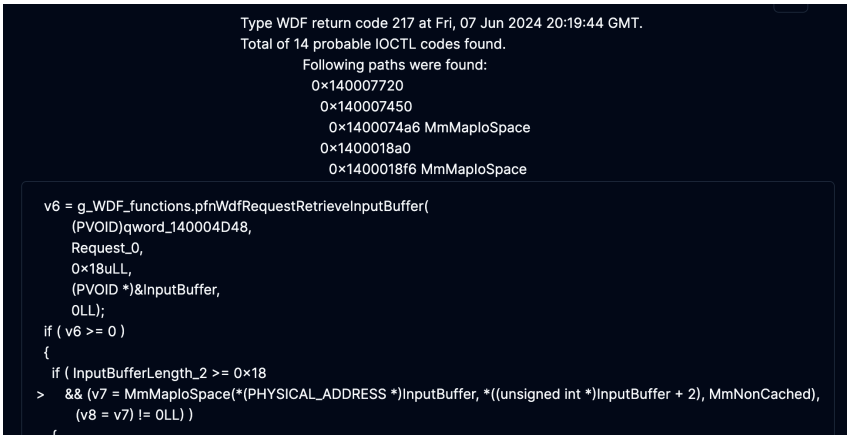


Figure A.3: The Pathfinder results are represented as a tree of addresses pointing to interesting functions. When hovered over, each function reveals the decompilation context around it within the driver, as shown in the centre of this screenshot.

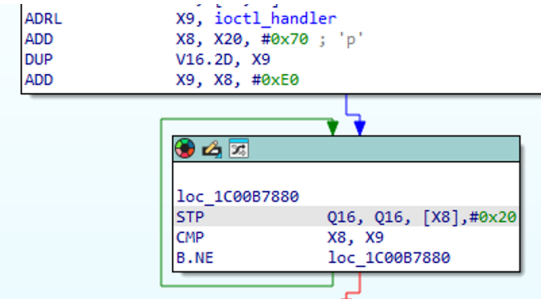
The subpage illustrating the specific results of the Pathfinder for a particular driver is shown in Figure A.3, accessible by selecting the Pathfinder results section within the driver card details.

A.4 Additional Pathfinder Screenshots

Not all screenshots are included in the main thesis body to maintain brevity. This section provides additional screenshots related to the Pathfinder module's ARM loop detection for IOCTL handlers. Specifically, Figure A.4 shows a loop assigning a single function address to all major functions.

```
MajorFunction = (_DRIVER_OBJECT *)DriverObject_0->MajorFunction;
do
{
  *(_QWORD *)&MajorFunction->Type = fn_ioctl_handler_wdm_arm_loop;
  MajorFunction->DeviceObject = (PDEVICE_OBJECT)fn_ioctl_handler_wdm_arm_loop;
  MajorFunction = (_DRIVER_OBJECT *)((char *)MajorFunction + 16);
}
while ( MajorFunction != &DriverObject_0[1] );
```

(a) Screenshot of IDA decompilation of ARM example for IOCTL handler setting through looping over all major function locations.



(b) Same IOCTL handler setting of ARM driver in disassembly view.

Figure A.4: Screenshots showing a loop recognized by the Pathfinder module for setting the IOCTL handler in ARM-based drivers.

```
MajorFunction = (int64x2_t *)v3->MajorFunction;
v14 = vdupq_n_s64((unsigned __int64)ioctl_handler);
do
{
  *MajorFunction = v14;
  MajorFunction[1] = v14;
  MajorFunction += 2;
}
while ( MajorFunction != (int64x2_t *)&v3[1] );
```

Figure A.5: Decompilation of ARM IOCTL handler set by looping over all major function locations currently not recognized by the Pathfinder.

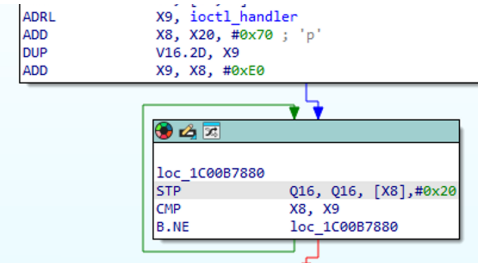


Figure A.6: Same IOCTL handler setting of ARM driver in disassembly view.

Figure A.7: A loop for setting the IOCTL handler in ARM-based drivers not yet detected by the Pathfinder module.

Despite modifications and improvements over the original VMware TAU script, the Pathfinder module does not incorporate all possible enhancements. For instance, the identification code for when a single IOCTL handler function was assigned to most major functions through a loop cannot handle all cases. The current implementation detects some loop types through the decompilation view instead of the disassembly view. As shown in Figure A.6, the disassembly resembles Figure A.4b, but the IDA decompilation view in Figure A.5 differs from Figure A.4a, preventing the current implementation from recognizing both. Therefore, extending ARM loop detection to handle cases like those in Figure A.7 or revising it to use only the disassembly view may enhance the current implementation to handle ARM drivers better.

A.5 Definitions of Key APIs and Intrinsics

This appendix provides brief definitions for APIs and intrinsics referenced in the main text. These definitions offer a quick reference for the specific functions and their relevance to discussing vulnerabilities in Windows kernel drivers.

- **IoAllocateMdl**: Allocates a memory descriptor list (MDL) to map a buffer, given the buffer's starting address and length.
- **IoCreateSymbolicLink**: Sets up a symbolic link between a device object name and a user-visible name for the device, which is required for user-space processes to communicate with the device.
- **MmMapIoSpace/MmMapIoSpaceEx**: Maps a given physical address range to non-paged system space and returns the base virtual address of this mapped range.
- **SeSinglePrivilegeCheck**: Checks the caller's security level, sometimes used to add custom security checks in a driver.
- **WdfDeviceCreateDeviceInterface**: Creates a device interface for this KMDF driver, through which user-space applications can interact with this device object.
- **WdfDeviceCreateSymbolicLink**: Creates a symbolic link to a specified device, which can be used for user-space applications to interact with the device object.
- **WdmlibIoCreateDeviceSecure/IoCreateDeviceSecure**: Uses an SDDL string to secure a device and create a named object.
- **ZwMapViewOfSection**: Maps a view of a section at a virtual base address into the virtual address space. Used in combination with ZwOpenSection to directly interact with memory.
- **ZwOpenSection**: Opens a handle for an existing section object with the desired access; an example of such a section object is `\\Device\\PhysicalMemory`.
- **__indword, __inbyte, __inword**: Generates the instruction to receive a double word/-word/byte data from a port.
- **__outdword, __outbyte, __outword**: Generates the out instruction to send a double word/word/byte data to a port.
- **rdmsr / wrmsr**: Intrinsics to read or write Model-Specific Register (MSR) on x86 architectures.
- **_ReadStatusReg / _WriteStatusReg**: Intrinsics to read or write special-purpose register on ARM, which includes ALU flags, security controls and others.

The following functions are built-ins of the C++/C language as the MSC compiler compiles them. The *w** variants are utilized for wide character strings (*wchar_t **).

- **memcopy/wmemcopy**: Copies a memory block from one location to another.
- **memmove/wmemmove**: Moves a memory block from one location to another.
- **memset/wmemset**: Fills a memory block with a specified value.
- **qmemcpy**: A human-readable name IDA gives to a single-byte copy operation.

A.6 Interrupt Request Packet Types in Windows Kernel Drivers

Windows kernel drivers must communicate with other drivers or the operating system itself. This communication is facilitated through I/O request packets (IRPs), which contain a type in their header that defines the purpose of the IRP. Specifically, the 28 major function request types shown in Table A.2 are defined in the Windows *wdm.h* header file for any IRP a driver might receive.

Some of these types, such as `IRP_MJ_CREATE`, must be implemented by all kernel-level drivers, as the operating system sends this request to the driver whenever something tries to open a handle to a file or device object. Others, like `IRP_MJ_DEVICE_CONTROL`, are necessary for any driver implementing a device type if a set of system-defined I/O control codes exists. For more detailed information on the different types, refer to the Microsoft documentation on IRP Major Function Codes [46].

IRP_MJ_CREATE	0x00
IRP_MJ_CREATE_NAMED_PIPE	0x01
IRP_MJ_CLOSE	0x02
IRP_MJ_READ	0x03
IRP_MJ_WRITE	0x04
IRP_MJ_QUERY_INFORMATION	0x05
IRP_MJ_SET_INFORMATION	0x06
IRP_MJ_QUERY_EA	0x07
IRP_MJ_SET_EA	0x08
IRP_MJ_FLUSH_BUFFERS	0x09
IRP_MJ_QUERY_VOLUME_INFORMATION	0x0a
IRP_MJ_SET_VOLUME_INFORMATION	0x0b
IRP_MJ_DIRECTORY_CONTROL	0x0c
IRP_MJ_FILE_SYSTEM_CONTROL	0x0d
IRP_MJ_DEVICE_CONTROL	0x0e
IRP_MJ_INTERNAL_DEVICE_CONTROL	0x0f
IRP_MJ_SHUTDOWN	0x10
IRP_MJ_LOCK_CONTROL	0x11
IRP_MJ_CLEANUP	0x12
IRP_MJ_CREATE_MAILSLOT	0x13
IRP_MJ_QUERY_SECURITY	0x14
IRP_MJ_SET_SECURITY	0x15
IRP_MJ_POWER	0x16
IRP_MJ_SYSTEM_CONTROL	0x17
IRP_MJ_DEVICE_CHANGE	0x18
IRP_MJ_QUERY_QUOTA	0x19
IRP_MJ_SET_QUOTA	0x1a
IRP_MJ_PNP	0x1b
IRP_MJ_MAXIMUM_FUNCTION	0x1b

Table A.2: All possible Major Function Codes for an IRP.

A.7 Coordinator API in Detail

This section of the appendix provides a detailed definition of the Coordinator API. The following table outlines the various endpoints available and their respective functions, split into categories as described in Section 5.2. The endpoints cover a wide range of operations, from fetching notes and general information about the database to specific file information and managing drivers and their associated data.

/notes-filter/<title>	Returns all notes with the given title.
/notes	Returns all notes.
/notes/<int:file_id>	Returns all notes related to a specific file ID.
/identification-notes/<int:file_id>	Returns all notes related to identification for a specific file.
/existing-files-info/<int:page>	Get all files on disk, with their identification notes, by pagination.
/unidentified-files-info	Get all files that have not been identified yet.
/windows-executables/<int:page>	Get all files identified as Windows executables by pagination.
/files/<int:file_id>	Update the specified file identification.
/files/<int:file_id>	Delete the specified file from disk, but keep its metadata.
/files/<int:file_id>	Update the specified file with the given information.
/files/<int:file_id>	Get a file by its ID.
/file-id/<file_hash>	Returns the file ID for the given SHA1 or SHA256 hash, if it exists.
/ogfile/<file_hash>	Get the origin file information and the underlying file ID.
/ogfile/<file_hash>	Add another origin for this file.
/ogfile/<ogfile_id>	Updates the extraction status of an origin file.
/ogfile	Add an origin file to the database, including the actual file.
/ogfile	Return a form to manually add an origin file.
/ogfiles-info	Get all origin files information (i.e. their IDs).
/ogfiles/<int:ogfile_id>	Get an origin file by its ID.
/ogfiles-to-extract/<type>/<int:page>	Get all origin files that need to be extracted, by pagination.
/extractions/<int:ogfile_id>	Get all extractions of an origin file.
/extractions	Adds the new origin file to the previous extracted origin file as an extraction.
/todo-signatures	Get all drivers that do not have a signature result.
/driver-signature/<int:driver_id>	Update the specified driver with the signature results.
/driver-signature/<int:driver_id>	Get the signature results of a driver.

/todo-paths/<arch>	Get a set of drivers that do not have path results and are of the given architecture.
/driver-paths/<int:driver_id>	Delete the path results of the specified driver.
/driver-paths/<int:driver_id>	Update the specified driver with the path results.
/driver-paths/<int:driver_id>	Get the path results of a driver.

/todo-fuzzing/<arch>	Get the next driver in the list to fuzz for that architecture.
/driver-fuzzing/<int:fuzz_queue_id>	Update the DB fuzzing state with the actual fuzzing state.
/driver-fuzzing/<int:driver_id>	Update the specified driver with the fuzzing results.
/driver-fuzzing/<int:driver_id>	Get the fuzzing results of a driver.
/fuzzing-notes/<int:driver_id>/<log_type>	Receives fuzzing logs for drivers and saves them to the correct notes.

/fuzzing-queue/<int:driver_id>	Get the fuzzing queue for a specific driver.
/fuzzing-queue	Get the entire fuzzing queue.
/fuzzing-queue	Clear the entire fuzzing queue (a debug API).
/fuzzing-queue-add-interesting	Add all unknown drivers 'interesting' to the fuzzing queue.
/fuzzing-queue	Add a specific driver to the fuzzing queue.

/drivers	Get all drivers.
/drivers-filter/origin/<origin>	Get all drivers where at least one relevant origin file is like the origin.
/drivers-filter/imports/<imp>	Get all drivers that import the given function.
/ogfile-drivers/<int:ogfile_id>	Get all drivers of an ogfile.
/ogfiles-filter/origin/<origin>	Get all origin files with the given origin.

/driver-id/<driver_hash>	Returns the driver ID for the given SHA1 or SHA256 hash.
/drivers/<int:driver_id>	Get a driver by its ID, with all its results.
/driver-tags/<int:driver_id>	Update the specified driver with the specified tag.

/origins	Get a list of unique origins in origin files.
/driver-tags	Get all possible driver tags.
/files-info/¡limit¿	Get information about all files (i.e. IDs) limited to that amount.
/functions	Returns all functions currently in the database, with the count of how often they are seen, ordered by the 'interesting' value.
/known-vulnerable-list	Returns all known vulnerable drivers.
/db-stats	Returns some statistics about the current database.
/help	Print all defined routes and their endpoint docstrings, i.e. this table.
/health	An endpoint for the Docker Healthcheck.

Declaration of use of AI-based tools

Table A.3 outlines the application and extent of AI-based tools utilized throughout this thesis. The table is modelled after the framework presented in "Assessing the Potential of AI-based Tools for Scientific Writing" [29].

AI-Based Tool	Use Case	Scope	Remarks
MS365 Copilot Version Jun 10, 2024	Code generation	Implementation	Utilized via VS Code integration
ChatGPT-4 Version Jul 25, 2024	Code modifications	Evaluation	PostgreSQL query modifications
ScholarGPT-4 Version Jul 25, 2024	Text revision for conciseness and academic English	Entire work	Below query was often used.
ChatGPT-3 Version Jul 10, 2024	Text revision for conciseness and academic English	Entire work	Used when GPT-4 free license is capped
Grammarly Version Jul 25, 2024	Grammar and style correction	Entire work	Client or driver names were stripped for this.

Table A.3: Declaration of AI-based tool usage throughout this thesis.

The following ScholarGPT query, occasionally adjusted for additional context, was utilized to rephrase the authors' words into more refined academic English: "I am writing a section for my master thesis and need you to help me rephrase the following text into academic English. Try to be as concise as possible while rephrasing and to keep the latex commands in."

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten schriftlichen Arbeit. Eine der folgenden drei Optionen ist in Absprache mit der verantwortlichen Betreuungsperson verbindlich auszuwählen:

- ○ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Es wurden keine Technologien der generativen künstlichen Intelligenz[1] verwendet.
- ◉ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz[2] verwendet und gekennzeichnet.
- ○ Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz[3] verwendet. Der Einsatz wurde, in Absprache mit der Betreuungsperson, nicht gekennzeichnet.

Titel der Arbeit:

Efficient Pipelining for WindowsDriver Vulnerability Research

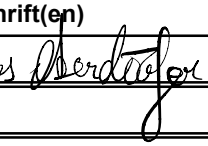
Verfasst von:
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Oberdörfer	Tobias

Ich bestätige mit meiner Unterschrift:

- Ich habe mich an die Regeln des «Zitierleitfadens» gehalten.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu und vollständig dokumentiert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Eigenständigkeit überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
31.07.2024	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie grundsätzlich gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

[1] z. B. ChatGPT, DALL E 2, Google Bard
[2] z. B. ChatGPT, DALL E 2, Google Bard
[3] z. B. ChatGPT, DALL E 2, Google Bard