POPKORN: Popping Windows Kernel Drivers At Scale

Anonymous Author(s)

ABSTRACT

External vendors develop a significant percentage of Windows kernel drivers, and Microsoft relies on these vendors to handle all aspects of driver security. Unfortunately, device vendors are not immune to software bugs, which in some cases can be exploited to gain elevated privileges. Testing the security of kernel drivers remains challenging: the lack of source code, the requirement of the presence of a physical device, and the need for a functional kernel execution environment are all factors that can prevent thorough security analysis. As a result, there are no binary analysis tools that can scale and accurately find bugs at the Windows kernel level.

To address these challenges, we introduce POPKORN, a lightweight framework that harnesses the power of taint analysis and targeted symbolic execution to automatically find security bugs in Windows kernel drivers at scale. Our system focuses on a class of bugs that affect security-critical Windows API functions used in privilege-escalation exploits. POPKORN analyzes drivers independently of both the kernel and the device, avoiding the complexity of performing a full-system analysis.

We evaluate our system on a diverse dataset of 212 unique signed Windows kernel drivers. When run against these drivers, POPKORN reported 38 high impact bugs in 27 unique drivers, with manual verification revealing no false positives. Among the bugs we found, 31 were previously unknown vulnerabilities that potentially allow for Elevation of Privilege (EoP). During this research, we have received two CVEs and six acknowledgments from different driver vendors, and we continue to work with vendors to fix the issues that we identified.

1 INTRODUCTION

Device drivers are essential to modern operating systems, serving as the backbone of communication between applications and underlying hardware components. In the Microsoft Windows operating system, device drivers handle critical hardware interactions, such as access to the GPU, BIOS updates, and management of network cards. Since all of these operations require privileged access, drivers are loaded directly into the kernel, and execute at the highest privilege level (i.e., *Ring 0*) [72]. Device vendors develop most of the drivers for Windows using frameworks provided by Microsoft, such as the Windows Driver Model (WDM) [24] and the Windows Driver Framework (WDF) [34]. Unfortunately, device vendors often lack rigorous, security-focused development processes, and as a result, they introduce bugs that can be exploited to gain elevated privileges.

To enforce basic security and performance requirements, Microsoft requires that drivers be signed by the Windows Hardware Quality Labs (WHQL) and be certified with Extended Validation (EV) [35, 70]. While this may appear to be an adequate safeguard, this stamp of approval can also provide a false sense of security, as Microsoft does not thoroughly verify a driver's code before its certification, and in fact, relies on vendors to adequately secure their own drivers.

Indeed, recent attacks have shown that certifying drivers does not provide complete security [53, 58, 75]. A relatively new technique called BYOB ("bring your own bug") makes it possible to load an unsigned driver into the kernel by piggybacking on a signed-but-vulnerable driver [56]. For instance, the LoJax and Slingshot malware families [19, 60] ship a signed-but-vulnerable driver with the malware itself, which allows for loading of the malware into the kernel. This problem also negatively affects game vendors, as players can bypass anti-cheat software running in the kernel by loading their unsigned kernel cheats [14, 63, 64].

Microsoft has taken these issues seriously, and developed several verification tools to support driver developers during the development life cycle. These tools can be roughly divided into *static* verification tools, and *dynamic* verification tools. Static tools, such as Static Driver Verifier (SDV) [46], analyze the driver source code and are targeted towards finding erroneous interactions with the Windows kernel. On the other hand, dynamic tools, such as Driver Verifier (DV) [23], monitor the driver execution to detect bugs that are difficult to find statically (e.g., buffer overruns and use-after-free vulnerabilities). However, DV and SDV only support a set of specific assertions, rules, and tests that focus on the correct usage of Windows APIs. Consequently, the applicability of these tools is restricted to a limited subset of security vulnerabilities.

Given the critical role of drivers, developing third-party tools for the identification of bugs and vulnerabilities in kernel drivers remains a priority. However, third-party analysis of kernel drivers is challenging for several reasons. First of all, certain tools operate only on source code, which is often not available for commercial drivers [46]. Secondly, some tools require the presence of the hardware device managed by the driver, which makes the analysis difficult to scale [15, 61, 62]. Third, some analysis techniques require that the driver be operating in the context of a functional kernel, which makes the analysis resource-intensive and challenging to automate because of the required installation procedure [15, 55, 61, 62].

To address these issues, we designed a lightweight and flexible vulnerability detection framework for Windows kernel drivers called POPKORN. The POPKORN framework focuses on identifying insecure uses of (certain) Windows kernel API functions within drivers. More precisely, POPKORN combines taint analysis techniques and targeted symbolic execution to detect when unsanitized user input (the *source*) can reach functions that provide access to critical kernel resources (the *sinks*). By focusing on a specific pattern of vulnerability, POPKORN can analyze the binary image of the driver without the need for the actual device or a functional kernel. As a result, the POPKORN framework can perform automated analysis at scale on a heterogeneous set of Windows drivers, including (but not limited to): BIOS drivers, GPU drivers, display drivers, printer drivers, and system diagnostic utilities.

In summary, this paper makes the following contributions:

(1) We perform an in-depth analysis of a class of logic bugs in Windows kernel drivers. We focus on API functions commonly seen in the exploitation of bugs that lead to local privilege

escalation, showing how the dispatch interface for driver interactions might allow user-mode input to reach critical sinks without proper checks.

(2) We present the design and implementation of POPKORN, a flexible, lightweight, and extensible framework to detect these API-misuse bugs in a diverse set of Windows drivers using a combination of taint analysis and targeted symbolic execution.

(3) We evaluate POPKORN on a diverse dataset of 212 signed Windows kernel drivers and demonstrate its effectiveness at finding high-severity vulnerabilities in real-world kernel drivers automatically and at scale. In total, POPKORN found 38 unique bugs, manually verified as exploitable. 31 of the 38 exploitable bugs were previously unknown vulnerabilities (0-days) that could potentially allow privilege escalation.

To foster further research in this area, we will release the source code of POPKORN and any other research artifact upon acceptance of this paper and following the rules of responsible disclosure.

2 BACKGROUND

In this section, we provide background information to help understand the runtime environment of a Windows kernel driver. We introduce concepts and terms we will use throughout the rest of the paper.

2.1 Windows Kernel Drivers

Kernel drivers allow code running in user mode (Ring-3) to interact with the operating system kernel and peripheral devices in a complex yet flexible fashion. Microsoft has offered different driver frameworks with different development paradigms over time. Starting with Windows/386, Microsoft introduced VxD kernel drivers to handle concurrent accesses to different system resources. With Windows 98, the Windows Driver Model (WDM) introduced a layered message-passing architectures for Windows drivers, which allowed compiled drivers to be binary-compatible with, and run on, multiple different versions of the Windows operating system. Modern versions of Microsoft Windows also support a newer framework, called the Windows Driver Frameworks (WDF). This framework offers object-oriented driver development options built on top of WDM. Despite the WDF being the framework currently recommended by Microsoft for modern driver development, WDM is still quite prevalent among existing software, as evidenced by the fact that 62% of the drivers collected during this research are WDM drivers. For this reason, in this paper, we focus on WDM drivers[1], since they represent the most widely used and deployed type of drivers.

WDM drivers follow a basic message-passing model: they accept Interrupt Request Packets (IRPs) requests, process them using a series of callback functions (dispatch handlers), potentially interact with a hardware device, and finally return a response (called IOStatus) [28, 29, 32].

Figure 1 provides an example of an interaction between a user-mode application and a kernel-mode driver in the WDM framework. The user-mode application starts by opening a handle to the device driver file, using the `CreateFile` API. The application then calls the function `DeviceIoControl` with the target IoControlCode (0x800 in this example), and two user-space buffers, the input and output buffer. Depending on the driver implementation, these can either be different buffers, or a single buffer can be reused for both. The `DeviceIoControl` function creates an IRP object, containing the IoControlCode and the buffers. Finally, the IRP is passed to the driver's *IRP_MJ_DEVICE_CONTROL* handler. At this point, the driver checks the IoControlCode and processes the desired actions, and writes any result data into the output buffer, before returning an *IOStatus* code indicating whether or not the operation succeeded.

In the rest of this section, we take a closer look at the structure of these messages as well as how they are handled.

2.2 Dispatch Handlers

Device drivers typically declare a set of dispatch handler routines (callbacks) that are invoked depending on the specific requests received from user mode. A driver registers these callbacks to be called by the kernel by storing them into an array of function pointers associated with the device object, known as the *major function table*. For example, the callback at index 14 in the major function table (`IRP_MJ_DEVICE_CONTROL` [27]) is invoked when a user calls the API function `DeviceIoControl` [37]. Most dispatch handlers have a corresponding user-space Windows API function that can be used by user-space programs to interact with the driver.

The right hand side of Figure 1 shows how drivers can register these dispatch handler routines. In this example, the driver leverages the *major function table* to register the dispatch handlers `ReadFunc`, `CloseFunc`, and `DeviceControlDispatch`. These handlers will be invoked when a user-space application calls `ReadFile`, `CloseFile`, and `DeviceIoControl` respectively on a driver file handle. Dispatch handlers are usually referred to by the name of the symbolic constant of their index in the major function table, or by referencing the corresponding user-mode function used to interact with them.

2.3 Interrupt Request Packets (IRPs)

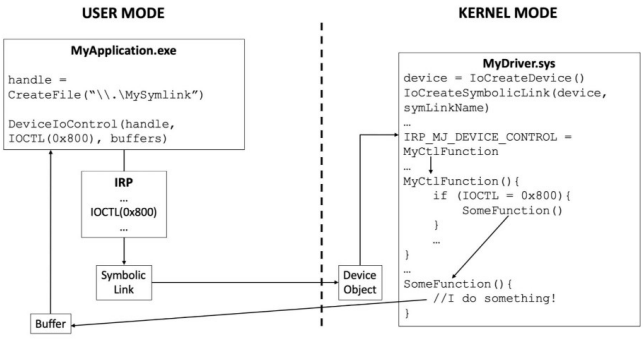
The messages being passed between the operating system, different drivers, and the user-space kernel API are called Interrupt Request Packets (IRPs). The layout of these packets consists of an IRP request header followed by a series of I/O-Stack Locations [25], each representing an operation to execute on a specific driver with a specific set of arguments. In WDM, kernel drivers are a part of a layered architecture [31]; drivers often request actions from other, lower-level, drivers by appending a new I/O-Stack Location IRP and forwarding the IRP to the target driver.

These IRPs are used in most driver interfaces that can be reached from user mode. The driver dispatch handlers will access their corresponding I/O-Stack Location to retrieve the arguments provided by the user-mode program.

3 POPKORN VULNERABILITIES

In this paper, we focus on a specific, known class of logic bugs that often lead to privilege escalation. Specifically, we are looking for a set of "Improper Input Validation"-style bugs, where a kernel driver passes untrusted user input (source) to sensitive Windows Kernel functions (sinks) without sufficient sanitization. Unprivileged users can then "trick" the kernel driver into performing sensitive functionality (e.g., mapping physical memory, accessing other processes' data) on their

[1]For the rest of the paper, unless otherwise specified, we are referring to the Windows Driver Model when referencing driver design or architecture.




```
NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT * Driver,
                               ...)
{
  ...
  struct UNICODE_STRING DevName;
  _DEVICE_OBJECT* Device;
  RtlInitUnicodeString(&DevName, "\\MySymlink");
  IoCreateDevice(Driver, 0, &DevName, ..., &Device);

  Driver->MajorFunction[IRP_MJ_CLOSE] = CloseFunc;
  Driver->MajorFunction[IRP_MJ_READ]  = ReadFunc;
  Driver->MajorFunction[IRP_MJ_DEVICE_CONTROL]
       = DeviceControlDispatch;
  ...
}
```

Figure 1: On the left, an interaction between a user-space program and a driver [16]. On the right, an example of dispatch handler routines (callbacks) being registered during driver initialization.

```
1  NTSTATUS DeviceControlDispatch(_DEVICE_OBJECT *DeviceObject,
2                                 _IRP *Irp)
3  {
4    auto IoStackLocation = Irp->Tail.Overlay.CurrentStackLocation;
5    auto Params = IoStackLocation->Parameters.DeviceIoControl;
6    auto Buf = Irp->AssociatedIrp.SystemBuffer;
7    auto Status = STATUS_UNSUCCESSFUL;
8
9    if ( Params.IoControlCode == 0x9C402530
10     && Params.InputBufferLength >= 8
11     && Params.OutputBufferLength >= 8
12   )
13   {
14     // Map one page as determined by request
15    PHYSICAL_ADDRESS* Address = (PHYSICAL_ADDRESS*)Buf;
16
17    void* res = MmMapIoSpace(*Address, 0x1000, MmNonCached);
18
19    if (res) {
20       *(void**)Buf = res;
21       IofCompleteRequest(Irp, 0);
22       Status = STATUS_SUCCESS;
23    }
24   }
25   return Status;
26 }
```

Listing 1: Motivating Example: Vulnerable Driver

behalf. Depending on which sensitive functions are accessible, the result can range from data corruption and system crashes, to full privilege escalation and device takeover. For the analysis in this paper, we selected common, and frequently exploited, sources and sinks of vulnerabilities to maximize the impact of our work.

Figure 1 presents an example of a driver function vulnerable to the style of bugs that our system aims to find. This function retrieves data from user space—specifically, the `Irp` object (Lines 4–6)—and performs some checks on these parameters (Lines 9–11). Usually, the `IoControlCode` value is used in a large switch-case statement, so the driver performs different operations depending on this value. Moreover, the driver then often performs a series of checks on any user-supplied data (e.g., the input buffer needs to be at least 8 bytes long). Our example, for the sake of simplicity, only contains one valid IoControlCode.

If all checks are successful, the function calls `MmMapIoSpace` (Line 17), which maps a physical page of memory—whose address, in this example, is supplied by the user—into the current virtual address space. In other words, this example shows how a user-space process might trick the kernel driver into mapping physical memory pages of other (potentially higher-privileged) processes into their own address space, without ensuring that the target process is allowed to access these pages.

The core idea behind POPKORN is to symbolically explore driver functions and automatically check whether the parameters of certain critical functions (such as `MmMapIoSpace`) can be directly controlled by a user-space program. In the next section, we will discuss the challenges associated with detecting this type of vulnerabilities, either manually or by using targeted symbolic execution.

3.1 Challenges

Compiled Binaries. Microsoft Windows kernel drivers (".sys" files) are distributed as normal Windows Portable Executable (PE) [73] files. Oftentimes these drivers are released without source code or documentation that might help to understand a driver's internals. Anyone attempting to audit the security of a driver has to first carefully reverse engineer it to understand the attack surface and available functionality. Furthermore, the asynchronous nature of the interactions between the WDM framework and the Windows kernel is particularly challenging for reverse engineering and program reasoning.

Modeling Kernel API Functions. The Microsoft Windows kernel is a complex piece of software that exposes thousands of APIs to user space and kernel drivers. Symbolic execution engines often approximate library or API functions using abstract implementations (summaries). This constrains the symbolic exploration to the code of the program-under-test, in our case the driver, and can help to keep the analysis tractable by significantly reducing path explosion. However, given the large number of API functions, providing these models for every function in the Windows kernel API is infeasible, since such function summaries are usually created manually. Symbolic execution tools, such as `angr` [65], attempt to solve this problem by replacing missing ones with a default abstract summary which simply returns a fresh symbolic variable.

Path Explosion. Generally, the number of paths in a program grows exponentially in terms of the number of conditional branches; a naive analysis based on enumerating all such paths will fail to scale, even for programs of modest size. This problem is commonly known as path explosion. Some specific areas that are prone to path explosion include bug-finding, bounded model checking [4], and formal verification [69].

Therefore, a path-based analysis must limit the number of branches explored, e.g. by deprioritizing or dropping paths indiscriminately or in a targeted manner (e.g., paths that cannot reach a point of interest).

Lastly, the analysis can be limited in scope to reduce the number of paths, e.g., by analyzing only individual functions. Pruning paths that cannot exercise a certain presumed vulnerable program point is known to be possible (with caveats), but generally, determining which paths will exercise any vulnerability is not.

Due to the callback-based nature of Windows drivers, we do not attempt to implement an analysis emulating a complete path through the driver code. Instead, we focus our analysis on the functions most commonly involved in existing exploits and vulnerabilities in order to keep the amount of driver-kernel interaction we need to model tractable.

4 CVE ANALYSIS

To motivate the selection of the sources and sinks used in this paper, we performed an analysis of the *Common Vulnerabilities and Exposures* (CVE) database [50], maintained by MITRE. Using a combination of regular expressions and manual post-processing, we extracted a total of 1390 CVEs likely related to third-party Windows drivers from the years 1999–2021. For this analysis, we used the official CVE database in XML format [51]. Since the CVE description text is often limited to a few sentences, we augmented this information with any text content extracted from all available referenced files and web pages.

In particular, we applied a series of regular expressions and manual filtering on the resulting text, in the following steps:

(1) Identify a set of CVEs that are related to the Windows kernel;
(2) Filter only for driver-related CVEs (i.e., we discard CVEs that are related to the mainline kernel);
(3) Select the CVEs that refer to third-party kernel drivers;
(4) Identify likely mentioned sources, sinks, and vulnerability types.

In the following sections, we analyze the results of this analysis, and study the prevalence of detected sources, sinks, and vulnerability classes individually.

4.1 Sources

To identify common input sources in third-party Windows drivers, we analyzed each augmented CVE text (including all linked references) for keywords that indicate the source of the input triggering the vulnerability. The results showed that the `DeviceIoControl` interface is by far the most common input source for known vulnerabilities, being mentioned in 760 (55%) of our selected CVEs. This is far and ahead of the next most common input sources, being "remote attack" at 21% and "DirectX interfaces" at 15%.

We also performed a manual investigation of 100 randomly sampled CVEs for which no input source was found, to ensure our analysis was not missing any major input sources. Among these 100 CVEs, we found 10 CVEs not related to Windows drivers, 36 related to the mainline Windows kernel, and 54 affecting third-party drivers. For 43 of these third-party driver CVEs, we could not determine an input source manually (e.g., CVE-2010-1592[2]), and of the remaining 11, the most common (4) input source was again the `DeviceIoControl` function. We include a detailed breakdown of this manual investigation in Table 6 in Appendix B.

[2]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1592

We suspect that this over-representation stems from the semantic ambiguity of the `DeviceIoControl` interface. While most interactions between drivers and user-mode applications follow a clear semantics (e.g., `ReadFile` is used to transfer or retrieve data from a kernel driver to user space using standard file APIs), the `DeviceIoControl` interface is designed to be entirely defined by the driver, with no interference from the operating system. This makes it impossible for the kernel to validate the arguments to the `DeviceIoControl` interface on the driver's behalf, leaving the driver itself solely responsible for ensuring that untrusted user input is handled securely.

Unfortunately, driver developers often fail to sanitize these potentially malicious inputs properly. This can lead to security vulnerabilities, for example, when a device driver directly dereferences pointers provided from user space or when it takes privileged actions based on user-space input, as is the case for the vulnerabilities targeted by POPKORN. For POPKORN, we therefore chose the function `DeviceIoControl` as the source for our symbolic analysis.

4.2 Sinks

Similarly to the input source analysis, we attempted to extract possible vulnerability sinks from the augmented CVE descriptions. We selected the 25 most commonly referenced Windows kernel functions and global variables, and categorized them based on the role they play in the description of the vulnerability. The results of this analysis are presented in Table 1. As reported in this table, some of the symbols mentioned in CVEs are vulnerability sinks, while other symbols refer to mitigation functions or exploit targets. We include the full analysis of these functions in Appendix E.

In addition, we analyzed our dataset of 3,094 WDM drivers by creating Control Flow Graphs (CFG) of each driver and statically extracting all the Windows API functions used. The top 25 most frequent functions used throughout the driver database are listed in Figure 3. Note that we pruned instances of functions that were undocumented in the official Microsoft documentation, e.g., `ZwReadVirtualMemory`, and `ZwWriteVirtualMemory`. In addition, we also filtered instances of the corresponding *closing* functions to make the list more diverse, for example, we counted the function `ZwOpenKey`, but pruned the closing function `ZwClose`. Notably, only a few API functions allow manipulation based on the user input, and most of them are used for basic operations (e.g., `RtlInitUnicodeString` and `RtlCopyUnicodeString` are used for initializing and copying Unicode strings).

Based on the results of these analyses, we selected the functions `MmMapIoSpace`, `ZwMapViewOfSection`, and `ZwOpenProcess` (from an existing CVE [57]) as relevant targets for our use case since they can often be trivially exploited for privilege escalation [5]. In the following sections, we describe each function in detail and highlight how unsafe usages of these functions can be exploited for local privilege escalation. In particular, we examine the prototypes of these functions and the arguments that, when controlled by an attacker, can be used to compromise a system.

`MmMapIoSpace`. This kernel routine maps a given physical address range into virtual memory and returns a virtual address pointing to the newly mapped address space. Typical use cases of this function include scanning the physical memory range, retrieving data

Category	Kernel API function and variables
Vulnerable Sink	MmMapIoSpace, RtlCopyMemory, KeBugCheckEx, ZwEnumerateValueKey, ZwQuerySystemInformation
Mitigation	ProbeForWrite, MmHighestUserAddress, ProbeForRead
Exploit target	HalDispatchTable, PsInitialSystemProcess

Table 1: Exploit-relevant Kernel API functions mentioned in CVEs.

stored by UEFI boot loaders, or mapping a PCI device's DMA buffers into kernel space and/or user space [10, 68, 71]. `MmMapIoSpace` is imported by 8% of the drivers in our evaluation dataset (see Table 2). The signature for `MmMapIoSpace` is:

```
PVOID MmMapIoSpace(PHYSICAL_ADDRESS PhysicalAddress,
                   SIZE_T NumberOfBytes,
                   MEMORY_CACHING_TYPE CacheType);
```

For the purposes of exploitation, the arguments `PhysicalAddress` and `NumberOfBytes` are particularly interesting since the parameters together denote the physical address range to be mapped into the virtual memory of the attacker process.

`ZwOpenProcess`. This kernel function is used to retrieve a handle to interact with another process. The Zw version of OpenProcess, as opposed to the Nt version, assumes that the parameters come from a trusted source, i.e., the kernel itself, and it processes the request without validation [33], making the kernel routine a good target for POPKORN's analysis. This function has the following prototype:

```
NTSTATUS ZwOpenProcess(PHANDLE ProcessHandle,
                       ACCESS_MASK DesiredAccess,
                       POBJECT_ATTRIBUTES ObjectAttributes,
                       PCLIENT_ID ClientId);
```

When exploiting `ZwOpenProcess`, an attacker who controls the `ClientId` of this call can retrieve a handle to any arbitrary process running on the system, including higher-privileged ones. The attacker can then, for example, modify the target's process memory or terminate the process.

`ZwMapViewOfSection`. Similar to `ZwOpenProcess`, this routine also assumes that its parameters originate from a trusted source. `ZwMapViewOfSection` creates a view of a Windows section object into the current address space. A *section* in Windows kernel terminology refers to any region of memory designated for shared access [30]. If a kernel driver can be exploited to map sections of shared objects into the current process' virtual memory space, which this process should not have access to (for instance, sections of the file `\Device\PhysicalMemory`), the attacker might be able to modify the memory of privileged processes or the kernel itself. This function is declared as follows:

```
NTSTATUS ZwMapViewOfSection(
        HANDLE SectionHandle,
        HANDLE ProcessHandle,
        PVOID *BaseAddress,
        ULONG_PTR ZeroBits, // 0
        SIZE_T CommitSize,
        PLARGE_INTEGER  SectionOffset, // NULL
        PSIZE_T ViewSize,
        SECTION_INHERIT InheritDisposition, // NULL
        ULONG AllocationType, // sets allocation flags
        ULONG Win32Protect // page access protection
);
```

In this case, the arguments targeted by POPKORN are `SectionHandle`, `ProcessHandle`, `BaseAddr`, `CommitSize` and `ViewSize`. The first parameter, `SectionHandle`, is generated by a successful call to `ZwOpenSection` or `ZwCreateSection`, which points to a privileged memory section from which a view has to be mapped. The parameter `ProcessHandle` refers to the target process into which the kernel will map the created view, and `BaseAddress` refers to the base address of the view. Finally, `CommitSize` and `ViewSize` represent the total number of bytes that need to be mapped in the process. We flag usages of this function as vulnerable if the `SectionHandle` is user-controllable or originates from the `"\Device\PhysicalMemory"` file. Additionally, the user has to control the argument `BaseAddr` or `ProcessHandle` and/or the `CommitSize` and `ViewSize` arguments to map arbitrary pages of `PhysicalMemory`.

5 POPKORN DESIGN

In this section, we elaborate on POPKORN's design and the different stages of the analysis process. A high-level overview of POPKORN is presented in Figure 2. First, POPKORN collects a heterogeneous set of software packages and extracts any driver files (".sys" files). Each driver is then analyzed independently using targeted symbolic execution to automatically discover potential exploit primitives for privilege escalation.

5.1 Retrieving Kernel Drivers

Download Engine. The first task is to create a database of driver files that can be used as input for the analysis. Since no single authoritative source for Windows drivers exists, we created a distributed crawler to scrape popular software repositories on the Internet.

The crawler downloaded around 90,000 compressed software packages from various sources in a period spanning 14 days. These packages are released in different file formats, including executable files (.exe) packaged with different types of installers (such as *InstallAware*, *InstallShield*, etc.) as well as compressed archives (.zip, .rar, .cab, .tar) with driver files (.sys) inside them.

Extraction Engine. Due to a large number of different packaging installers used to create executable files, we used a universal extraction engine called *UniExtract2* [12] to unpack all the compressed files. The tool uses a signature detection mechanism for determining different file types, and then invokes the appropriate extractor for each detected contained file. After running the appropriate extractor, our system saves any extracted driver files (".sys" files) for further processing. This approach successfully unpacks 94% of all downloaded packages across the different file formats, and we extracted (coincidentally) exactly 5,000 Windows drivers.

5.2 Analysis Engine

Our Analysis Engine is built on top of the `angr` framework [65]. After loading a kernel driver, the engine symbolically explores the `DeviceIoControl` dispatch handler until a state reaches a potentially vulnerable Windows kernel API function. As discussed in the previous section, POPKORN supports the detection of vulnerabilities

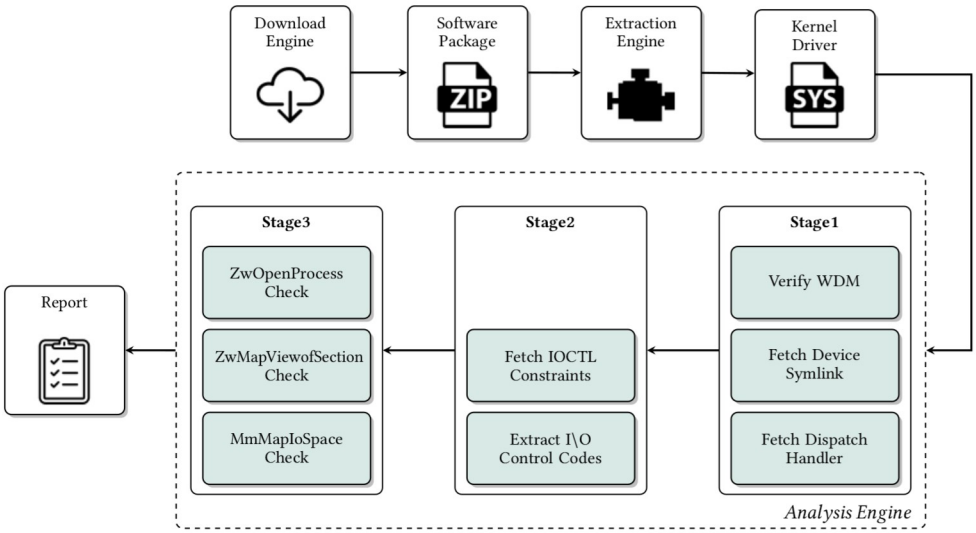


Figure 2: POPKORN System Overview.

related to the usage of the following functions: `MmMapIoSpace` [40], `ZwMapViewOfSection` [42], and `ZwOpenProcess` [43].

Whenever a sink is discovered, the built-in vulnerability detection mechanism examines the source of the arguments. As discussed in Section 2, users can store data in input buffers and send them to the driver using a `DeviceIoControl` request. During the symbolic exploration, POPKORN flags a driver as vulnerable if the arguments to the sinks are directly or indirectly loaded from the user-mode buffers. When a vulnerability is detected, our system creates a report outlining relevant meta-information about the target, including the `IoControlCode` used to invoke the target and any constraints on the `DeviceIoControl` parameters, such as buffer length checks.

Imports, Avoiding Paths, Device Names. The analysis starts by ensuring that the driver is based on the Windows Driver Model (WDM) by checking if the driver object is initialized using the `IoCreateDevice` WDM API function. When a call to `IoCreateDevice` is encountered, POPKORN extracts the device name from the arguments and reports it to the user so that a Proof-of-Concept (PoC) can easily be created. Then, POPKORN locates possible error conditions in the program, specifically the `KeBugCheckEx` symbol [39], which detects potential system corruption. This information is used during the symbolic exploration to terminate any paths that reach this function. Since this function never returns, this strategy avoids potential false positives and discards uninteresting paths.

`DeviceIoControl` Dispatch Handler. The first step in our analysis is to locate the dispatch handler `DeviceIoControl` and record the global state of the driver after successful initialization. We symbolically execute the driver entry point until it returns with a *STATUS_SUCCESS* return code, indicating successful initialization. We save the recorded state and reuse its concrete memory for the later stages of the analysis in order to preserve any initialized state. Furthermore, we ensure that the dispatch handler is called with the same `DeviceObject` as was returned by the `IoCreateDevice` function during initialization. This is important as drivers can store driver-specific information in the `DriverExtension` field, which can later be accessed by the dispatch handlers.

Fetching IoControlCodes. Next, we have to find the 32-bit integer I/O Control Codes, often called IOCTL codes, supported by the driver. Requests sent from user-mode code to the dispatch handler via `DeviceIoControl` contain the specific `IoControlCode` in the IRP structure. Kernel drivers, after parsing the `IoControlCode` code, often use a switch-case statement to perform the desired functionality. Figure 1 illustrates how a driver processes an IRP to access the user-mode parameters. The driver accesses its I/O-Stack Location [26] by accessing the field *Tail.Overlay.CurrentStackLocation*, allowing access to any user-mode parameters.

To extract the possible values of these IoControlCodes, POPKORN synthesizes an IRP structure with a single I/O-Stack Location with symbolic arguments. After symbolically exploring the `DeviceIoControl` function, it checks, for each path, if the symbolic variable corresponding to the `IoControlCode` code was constrained to a single value, and if so, reports it as a valid `IoControlCode`.

Function Analysis. Finally, as described in Section 5.2, POPKORN symbolically executes the `DeviceIoControl` handler with an integrated vulnerability detection engine for the aforementioned vulnerabilities in the three sink functions. Our system first creates the symbolic expressions corresponding to the `IoControlCode` and the contents and lengths of the input and output buffers. This information is then used to create a symbolic `IRP` structure, which, together with the `DeviceObject` recovered from the successful `DriverEntry` execution, are used as the input parameters to the `DeviceIoControl` handler.

At this point, POPKORN starts to symbolically execute this handler function. Whenever a call to a sink function is encountered during the symbolic exploration, POPKORN checks whether or not the parameters are user-controllable, and a vulnerability is reported for the current call if so. POPKORN then outputs a report indicating which vulnerable functions were triggered, the arguments that the user-mode process can control, as well as the data that should be passed by the user-mode program to reach the vulnerable function. This includes, at a minimum, the `IoControlCode` needed to reach the vulnerability, any constraints placed on the user buffers and their contents and lengths. During our research, we used this meta-information to manually generate a proof-of-concept exploit to verify that the vulnerability exists and is exploitable.

Modeling the Environment. The POPKORN framework is built on top of `angr` [65], an open-source symbolic execution engine. We added support for the Windows kernel driver environment

since angr is targeted towards programs that run in user space. The main component that had to be implemented was a collection of symbolic summaries called SimProcedures [1] for Windows kernel API functions. For our analysis, creating summaries for the following minimal set of functions was sufficient to reach the targeted sink functions: IoCreateDevice, IoCreateSymbolicLink, RtlInitUnicodeString, RtlCopyUnicodeString, ExAllocatePool, ZwOpenSection, and ExAllocatePoolWithTag[3].

Taming Path Explosion. Finally, POPKORN implements a combination of measures to combat path explosion. First, we keep track of the number of generated states and stop the execution when we reach a threshold of 10,000 symbolic states (called SimStates in angr). In addition, we reduce the scope of our analysis to driver functions only by implementing symbolic summaries for kernel functions instead of emulating their concrete implementations. As a result, no states in unrelated kernel code are created, reducing the amount of code to analyze.

6 EVALUATION

To determine the effectiveness of POPKORN, we evaluate it on a dataset of 3,094 WDM kernel drivers. We first look at the ability of POPKORN to detect real-world security bugs in Section 6.1. Then, in Section 6.2, we present an in-depth analysis of some of the detected bugs. We present an analysis of false positives and false negatives in sections 6.3 and 6.4, respectively. Finally, in Section 6.5, we compare our results to Microsoft's Driver Verifier tools.

Analysis Performance. We ran our experiments on an eight-core Intel i7-7700K 4.20GHz CPU and 64GB of RAM. On average, every successful execution of the analysis engine spent about 6.92 seconds analyzing each driver (min 0.40 sec, max 148.72 sec) with 70% of drivers' analysis finishing in less than 24.58 sec, and the average driver size being 381.5 KB (min 1.2 KB, max 52 MB).

Dataset Collection. To evaluate the ability of POPKORN to find bugs in the real world, we first had to assemble a suitable dataset. To this end, as discussed in Section 5, we first downloaded a diverse set of about 90,000 software packages from various sources and extracted 5,000 kernel drivers (".sys" files) for both 32-bit and 64-bit versions of Windows combined. Out of these drivers, 3,094 drivers (62%) were based on the Windows Driver Model (WDM). We further filtered these drivers down to 271 containing at least one of our target sink functions (MmMapIoSpace, ZwMapViewOfSection or ZwOpenProcess). To avoid skewing our results from over represented drivers, we performed a best-effort driver de-duplication based on the driver names, hashes, debug symbol paths, and manual checking. After this, we were left with a set of 212 unique drivers.

The results presented in this section will refer to the set of de-duplicated drivers. Our analysis was run on every single driver in the dataset, but unless otherwise noted, most results were consistent across driver versions. In case we found divergences between the results for different versions of the drivers, we mentioned them separately.

[3]For more details about these functions, refer to: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/

	Using Function		**Vulnerable Usages**	
Kernel Function	Total	Dedup.	Total	Dedup.
MmMapIoSpace	240 (7.76%)	188	24	17
ZwMapViewOfSection	40 (1.29%)	32	17	12
ZwOpenProcess	14 (0.45%)	11	0	0
Total (Unique)	271 (8.76%)	212	38	27

Table 2: Drivers using specific target functions, and drivers found vulnerable by POPKORN, out of a total of 3,094 WDM drivers.

6.1 Bug Finding

The second and third column of Table 2 show a breakdown of the number of drivers that use at least one of the sinks that our analysis targets (MmMapIoSpace, ZwMapViewOfSection, and ZwOpenProcess). Note that certain drivers might use more than one of the three target functions, so the numbers add up to more than 271. The 271 drivers were used as input to POPKORN. The last two columns of Table 2 present a breakdown of the results that POPKORN's analysis delivers: The total number of vulnerable drivers we found for each one of the different sinks.

In total, POPKORN found 38 distinct vulnerable drivers (27 after de-duplication). 24 vulnerabilities were related to improper calls to MmMapIoSpace and 17 to invocations of ZwMapViewOfSection. Our dataset did not contain instances of vulnerable ZwOpenProcess calls. All 38 bugs were verified manually, demonstrating a zero false positive rate. For false negatives, we performed multiple analyses of known-vulnerable drivers with POPKORN. This provided some anecdotal evidence of a relatively low numbers of false negatives (< 33% in both experiments). Of the 38 bugs involving the critical functions, 31 bugs were previously unknown, and most of them potentially allow for Elevation of Privilege (EoP) on the system, because of their similarity with the bugs used in existing EoP exploits. The remaining seven bugs were previously known or already had CVEs assigned to them. All bugs detected by POPKORN have the potential to impact the security and the availability of the system, either via a Denial-of-Service (DoS), leaking sensitive kernel data, or privilege escalation.

Vulnerability Impact and Disclosure. At the time of writing this paper, we have received two CVEs and six acknowledgments for reporting the bugs. Only one of the CVEs has been made public so far and has been assigned a CVSS score of 7.8 HIGH. Table 5 shows the impact of each vulnerability not currently under embargo, along with their respective disclosure status. Notably, POPKORN found two high-impact privilege escalation bugs in AMD's flagship graphic software (AMD Radeon Software Adrenalin) that is shipped pre-installed on AMD machines. In addition, POPKORN found a critical bug in MSI's principal gaming center (MSI Dragon Center), which also comes pre-installed with every MSI device that is released, along with three critical bugs in Intel's flash update utility used by many driver developers themselves.

These findings demonstrate the real-world impact of POPKORN. The remaining bugs are either prohibited from being disclosed or are in the triage process. We are working closely with the vendors' mitigation teams on rolling out bug fixes and disclosing them publicly. Interestingly, despite the fact that some of the bugs that we reported are present in critical software (such as antivirus engines and BIOS update utilities), we did not hear back from certain vendors.

6.2 Real bugs found by POPKORN

In this section, we explain two vulnerabilities found by POPKORN in real-world drivers.

Case Study I: MmMapIoSpace Abuse. One of the most interesting bugs that POPKORN found was a privilege escalation by `MmMapIoSpace` abuse (this is one of the six acknowledged bugs, and the fix is yet to be released). A simplified version of the vulnerable device driver that uses `MmMapIoSpace` is shown in Listing 2. In `Lines 4–7` the physical address range is mapped into the virtual memory, and a pointer to its base address is returned. However, the mapped address is not available to user mode yet. The base virtual address and size are now used to create a Memory Descriptor List (MDL), which essentially describes the buffer that needs to be allocated via `IoAllocateMdl` [38] (`Line 13`). Finally, a pointer to MDL is passed to `MmMapLockedPages` [41] (`Line 19`). The second argument specifies the *AccessMode* as either UserMode (1) (meaning accessible to user mode) or KernelMode (0) (accessible only to kernel mode), which determines where the MDL will be mapped. Since the drivers are primarily meant to be used by user-space applications to allow interaction with the hardware, the MDL has to be mapped to UserMode (1). Then, the starting address to the mapped range is returned through the output buffer (`Line 20`).

Therefore, since the physical address range is user-controllable, an attacker can ultimately map kernel memory into its process space. To exploit this, an attacker can simply iterate through all physical pages until it reaches critical kernel data structures, such as `EPROCESS` which is the kernel structure used to represent a running processes. The attacker can then locate and overwrite their process token with the security token of a process running with SYSTEM privileges, successfully escalating their privileges to NT Authority/System. Our analysis only detects whether or not the user-controlled physical address range is provided to `MmMapIoSpace`, a determination of actual exploitability is performed manually afterward.

Case Study II: ZwMapViewOfSection Abuse. This section explains a typical case of `ZwMapViewOfSection` abuse that POPKORN detected in multiple drivers. An example of a vulnerable implementation can be seen in Listing 3. The `ProcessHandle` and `CommitSize / ViewSize` are loaded from the input buffer (Lines 4 and 5). Then, `ZwOpenSection` opens a privileged section handle (with r/w permissions) that originates from the `"\\Device\\PhysicalMemory"` (Lines 12–16). This allows an attacker to map an arbitrary number of bytes of physical memory into their own process. The exploitation of this vulnerability is similar to the one previously described for `MmMapIoSpace`. For a user-controlled `SectionHandle`, the same can be achieved by using existing techniques to leak a kernel section handle for the PhysicalMemory device [5].

6.3 False Positives

False positives should be extremely rare with POPKORN as the analysis has the full constraints for a path and the solver should guarantee that the path is possible during the real execution. False positives can arise when unmodeled functions perform side-effects, e.g., writing to arguments or global memory. In such cases, stale data can be erroneously treated as tainted inputs. However, in our evaluation, none of the bugs reported by POPKORN exhibited this behavior, and we manually verified all 38 (27 de-duplicated) bugs as true positives.

```
1  void *__fastcall mmapiospace_func(void *inbuf){
2
3    // mapping addr from input buffer
4    base_addr = MmMapIoSpace(
5        *(void**)(inbuf+0),  // start addr,
6        *(size_t*)(inbuf+8), // number of bytes
7        MmNonCached); // Caching Behavior
8
9    if (base_addr)
10   {
11     // allocating a memory descriptor list
12     // (MDL) to map virtual buffer
13     mdl_ptr = IoAllocateMdl(base_addr,
14                 *(size_t*)(inbuf+8),0,0,0);
15
16     if (mdl_ptr)
17     {
18     // pages are being mapped in UserMode(1)
19     mapped_start_addr = MmMapLockedPages(mdl_ptr, 1);
20     *(void**)outbuf = mapped_start_addr;
21
22     // freeing allocations
23     IoFreeMdl(mdl_ptr);
24     MmUnmapIoSpace(base_addr, inbuf_size);
25     return SUCCESS;
26     }
27   }
28 }
```

Listing 2: MmMapIoSpace Abuse detected by POPKORN.

```
1  void *__fastcall zwmapviewofsection_func(void *inbuf)
2  {
3    void *SectionHandle;
4    HANDLE *ProcessHandle = *(HANDLE *)&inbuf[0];
5    SIZE_T CommitSize = *(size_t*)inbuf[1];
6    PVOID BaseAddress = 0;
7    ULONG_PTR ViewSize = CommitSize;
8    struct _OBJECT_ATTRIBUTES ObjectAttributes;
9    struct _UNICODE_STRING DestinationString;
10
11   //initiating ObjectName attr with "PhysicalMemory"
12   RtlInitUnicodeString(&DestinationString,
13                 "\\Device\\PhysicalMemory");
14   ObjectAttributes.ObjectName = &DestinationString;
15
16   // Mapping the privileged SectionHandle to
17   // "PhysicalMemory"
18   success = ZwOpenSection( &SectionHandle,
19     DesiredAccess, &ObjectAttributes);
20
21   if(success){
22     result = ZwMapViewOfSection( SectionHandle,
23     ProcessHandle, BaseAddress, 0, CommitSize, 0,
24     ViewSize, ViewShare, 0, 0x40u);
25
26     return result;
27   }
28 }
```

Listing 3: ZwMapViewOfSection Abuse detected by POPKORN.

Furthermore, we do provide models for common memory manipulation functions, e.g., `memcpy`, to reduce the chances of stale symbolic data erroneously triggering a vulnerability report. If a new function is identified that could cause a false positive, it is trivial to provide a model for it going forward.

6.4 False Negatives

We analyzed the possibility of false negatives of our analysis in two experiments. Specifically, by evaluating POPKORN on both a public dataset of vulnerable drivers as well as drivers mentioned in public CVEs, we provide an empirical false negative rate.

Vulnerable Drivers. For the first experiment, we used a publicly available dataset of 128 drivers with physical memory mapping vulnerabilities[4]. From this initial dataset, we removed 23 drivers that do not contain any of our sinks. In addition, we removed two drivers that were also included in our crawled dataset.

For the experiment, we ran POPKORN on the remaining 103 drivers, and report our results after driver de-duplication, as this dataset also includes multiple versions of the same driver. There were 30 de-duplicated drivers in the dataset. A summary of the results is presented in Table 3: our system detected 21 vulnerable drivers (68%), timed out while analyzing 6 drivers (19%), and terminated for 4 drivers without finding any vulnerability (13%). Timeouts were largely caused either by long-running concrete loops (e.g., memset on a buffer of size 0x20000), which is a known weakness of `angr`, or by state explosion in very complex drivers. Manual investigation of the 4 drivers where POPKORN reported no results showed that these drivers maintained internal state, and that the vulnerable functions would only be called after a sequence of multiple `DeviceIoControl` calls were made, a behavior not currently supported by POPKORN.

With the exception of one driver (included twice in the results), the results were consistent across driver versions. For the inconsistent driver, the analysis succeeded in detecting the vulnerability for one version, while timing out for another. Manual verification showed that the other versions of the duplicate driver contains the same vulnerability; to be conservative, we counted this driver as *timed out* in our results.

Analysis Result	#Drivers
Reported vulnerability	20 (+1*)
Timed out	6
Analysis terminated	4

Table 3: POPKORN results for `physmem_drivers`

Dataset Drivers with Known CVEs. To assess false negatives from another angle, we separately inspected all drivers in our crawled dataset that are explicitly mentioned in a CVE. This dataset consists of 16 CVEs, 7 of which refer to vulnerabilities POPKORN could find. The other 9 either refer to unsupported bug classes (unchecked pointer dereference, buffer overflow, etc.) or cannot be reached via the `DeviceIoControl` interface. After analyzing the 5 drivers (3 unique) mentioned in these CVEs, POPKORN detected the known vulnerability in 4 out of 5 drivers (2 out of 3 unique drivers). After manual analysis, we discovered that the remaining driver (MODAPI.sys) was not the vulnerable driver mentioned in the CVE, but a different driver sharing the same name.

Summary. Based on these results, we believe that POPKORN achieves both a very low false positive rate, and an acceptable rate of false negatives, positioning it well for automated, large-scale analyses of Windows drivers.

[4]https://github.com/namazso/physmem_drivers

6.5 Comparison with Microsoft's Driver Verifier

In this section, we compare POPKORN against two Microsoft tools, namely Driver Verifier (DV) [23], and Static Driver Verifier (SDV) [46]. We selected these two tools because they are provided by Microsoft and they are purposefully designed for and actively used by driver developers to assess the security and stability of their Windows kernel drivers. Any bugs missed by Driver Verifier and Static Driver Verifier might land in release builds. Unfortunately, since the source code for the drivers in our dataset is not available, we could not directly use SDV. However, as a proxy, we analyzed the assertion rules documented in SDV to analyze if the API-misuse bugs, which are targeted by POPKORN, could be detected.

Driver Verifier (DV). This tool is shipped with the Windows Driver Development Kit (WDK), and it runs a predefined list of checks for performing stress tests on device drivers under heavy load. Microsoft recommends using DV throughout the development cycle of kernel drivers to find design flaws, troubleshoot problems, and to debug crashes early in the development cycle.

Driver Verifier triggers specific bug checks (via Blue Screen of Death (BSoD)) [48] if it detects a violation during stress testing. The bug checks help driver developers in debugging the root cause responsible for the crash, such as Bug Check 0x3B [44] (triggers when an exception occurs while transitioning from non-privileged to privileged code), Bug Check 0xBE [45] (triggers when a driver attempts to write read-only memory), and Bug Check 0xC4 [47] (triggers when any routine executes at incorrect interrupt request level IRQL).

Nevertheless, these checks are based on generic rules that are applicable to every driver. These generic assertions prevent DV from verifying driver-specific interfaces and code paths that require taint-style analysis, such as the `IRP_MJ_DEVICE_CONTROL` interface. They are primarily geared towards detecting irregular API usage violations but cannot analyze the input source for the API functions. Therefore, this makes DV unsuitable for finding the privilege escalation bugs targeted by POPKORN. In addition, DV requires that the drivers be already installed in the Windows kernel, and it checks each installed driver only during the OS startup process. Overall, this increases the manual effort required for the analysis and limits DV from scaling to analyze a database of thousands of drivers.

Table 4 describes our results when evaluating Driver Verifier and POPKORN over a set of 20 known vulnerable WDM drivers (taken from already known CVEs) and 20 WDM drivers taken randomly from our database (that have at least one of the target functions). We tested Driver Verifier on the Windows 10 20H2 build, and measured an average execution time of each driver of 246 seconds (min 106 sec, max 438 sec). Since Driver Verifier can only analyze already-installed drivers while the system boots, it took us 40 system restarts to complete the analysis. Out of the 20 vulnerable drivers, DV invoked bug checks in four drivers, while POPKORN found a vulnerability in ten drivers, for the remaining ten drivers the analysis did not terminate with a result due to state explosion. Moreover, out of the 20 new drivers, DV triggered no bug checks, while POPKORN detected one `MmMapIoSpace` abuse bug.

Static Driver Verifier (SDV). This tool is a code verification utility for driver developers shipped with the Windows Driver Development Kit (WDK) [46]. SDV analyzes the driver's code using a predefined set of rules and assertions to detect Windows API usage

Driver Dataset	Bugs found by POPKORN	Crashes found by DV (BugCheck)
Known vulnerable drivers (out of 20)	8-MmMapIoSpace 2-ZwMapView	2 (0x3B) 1 (0xBE) 1 (0xC4)
Random Drivers (out of 20)	1-MmMapIoSpace	0

Table 4: Bug comparison between POPKORN and Driver Verifier.

bugs and driver design issues. However, SDV does not have rules for detecting improper access to physical memory. In general, SDV is not geared towards finding defects that require taint analysis of Windows Kernel API functions.

7 FUTURE WORK

Supported Driver Frameworks. POPKORN currently analyses drivers implemented using the Windows Driver Model (WDM) [24]. Although the majority (62%) of drivers from our dataset were based on WDM, Windows Driver Frameworks (WDF) is the currently recommended driver architecture by Microsoft. With increasing adoption, support for WDF drivers becomes more important, and we plan to extend POPKORN by adding this support in future work. This requires a much more extensive model of symbolic objects and is considered orthogonal to this paper, since the identified class of vulnerabilities and the important sources and sinks should remain the same.

Exploit Generation. While responsibly disclosing the vulnerabilities, it is critical to provide working Proofs-of-Concept (*PoCs*) to driver vendors to prove a vulnerability's impact. However, creating *POC*s manually for every bug report is a laborious task. To expedite this process, we plan on automating the creation of *PoC*s for detected vulnerabilities.

8 RELATED WORK

This section discusses the state of the art in current research in kernel-level bug finding and analysis. We begin by discussing the closest competitors to POPKORN, followed by a discussion of other general advances made in symbolic execution, static analysis, and fuzzing on kernel drivers. Finally, we attempt to highlight current research directions and areas where improvements in research are necessary.

For fuzzing Windows kernel drivers, tools such as ioctlbf [74], ioctlfuzzer [54], iofuzz [22] and IoAttack by Microsoft are available. These tools are fairly basic, with no coverage guidance or insight into kernel state beyond the return values of the kernel interfaces. It is notable that ioctlbf is a Windows driver fuzzer designed to fuzz the DeviceIoControl interface specifically and can recover valid `IoControlCodes` through fuzzing by analyzing differences in kernel API result codes between valid and invalid `IoControlCodes`. However, it is not designed to detect the taint-style bugs POPKORN targets, aiming to find memory-corruption bugs instead. More recently, BSOD[21] was proposed to fuzz NVIDIA GPU drivers on multiple operating systems. Unfortunately, this requires record & replay of real hardware interactions, making it infeasible for large-scale analysis as performed in this paper.

Static analysis is another popular technique for finding bugs in OS kernel drivers. However, recently, static analysis has been heavily focused on UNIX-based systems, where the source code of the kernel or the drivers is often used [11, 13, 20, 59, 66, 67]. However, static analysis is also used to aid in dynamic analysis tasks, e.g. NTFuzz [9] uses static analysis on user-space code to find type-information of system calls to effectively fuzz the Windows kernel binary code. However, POPKORN does not rely on the type information of system call arguments, and can directly analyze the kernel driver binaries symbolically at scale. Notably, other significant static analysis research on the Windows kernel and drivers dates back more than a decade, e.g., SLAM [3], Device Driver Analyzer (DDA) [2], and Bounded Address Tracking in Jakstab [17].

Lastly, kernel code poses many challenges for symbolic execution due to explicit and implicit interactions with the environment that must be correctly modeled. Nevertheless, symbolic execution has been used successfully for kernel and kernel driver bug-finding. For Windows kernel drivers, Kuznetsov et al. proposed DDT [18], a symbolic execution framework based on the KLEE [6] symbolic execution engine paired with a QEMU runtime that found several memory corruption bugs in device drivers at the time. However, DDT requires significant manual effort to provide PCI device information, install the driver in the VM, and configure the Microsoft Driver Verifier. - S^2E [8] performs selective symbolic execution using full-system emulation and enables symbolic execution not only in the device drivers but anywhere in the kernel. However, due to the full-system emulation, S^2E is not well-suited for large-scale analysis. Recently, Cao et al. [7] have proposed differential replay of full Windows kernel executions to find bugs in the Windows kernel itself. However, replay systems require emulating a full Windows operating system, where any drivers to be analyzed have to be installed, set up, and configured correctly. In contrast, solutions like POPKORN (which examine drivers in isolation) can analyze drivers outside of their native environment and constrain any state explosion to the behavior of the kernel driver itself instead of being affected by unrelated kernel behavior.

9 CONCLUSION

Windows kernel drivers pose an inherent risk to the security of a system since they execute with kernel-level privileges. This risk is exacerbated by the fact that vendors are responsible for the drivers' security. In this paper, we introduced POPKORN, a symbolic-execution-based bug-finding system to help vendors and third-parties find critical vulnerabilities in their kernel drivers. POPKORN is a lightweight framework analysis that can download, extract, and find critical vulnerabilities in Windows kernel drivers in an automated fashion. We show that our technique is effective in recovering the device names, `IoControlCodes`, function arguments, and the buffer data needed to reach a vulnerability. Based on this information, we could effectively generate proof-of-concepts by hand. We conducted a thorough evaluation of POPKORN over a diverse set of Windows kernel drivers and compared its performance against Microsoft's analysis tool, Driver Verifier (DV).

POPKORN found a total of 38 bugs, 31 of which were previously unknown. We have received two CVEs and six acknowledgments from vendors so far. We are open-sourcing POPKORN to provide an automated vulnerability detection framework for securing Windows kernel drivers and fostering further research in this field.

REFERENCES

[1] Angr. 2018. *Programming SimProcedures.* angr. https://docs.angr.io/extending-angr/simprocedures

[2] Gogul Balakrishnan and Thomas Reps. 2008. Analyzing stripped device-driver executables. In *Tools and algorithms for the construction and analysis of systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–140.

[3] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram Rajamani. 2004. *SLAM and static driver verifier: Technology transfer of formal methods inside microsoft.* Technical Report MSR-TR-2004-08. Springer-Verlag. 22 pages. https://www.microsoft.com/en-us/research/publication/slam-and-static-driver-verifier-technology-transfer-of-formal-methods-inside-microsoft/

[4] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. 2009. Bounded Model Checking. *Handbook of satisfiability* 185, 99 (2009), 457–481.

[5] Ruben Boonen. 2017. *Part 19: Kernel Exploitation -> Logic bugs in Razer rzpnk.sys.* FuzzySecurity. https://www.fuzzysecurity.com/tutorials/expDev/23.html

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *17th USENIX Security Symposium (USENIX Security 08)* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[7] Mengchen Cao, Xiantong Hou, Tao Wang, Hunter Qu, Yajin Zhou, Xiaolong Bai, and Fuwei Wang. 2019. Different is Good: Detecting the Use of Uninitialized Variables through Differential Replay. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, London United Kingdom, 1883–1897. https://doi.org/10.1145/3319535.3345654

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-Vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems (ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396 521 citations (Semantic Scholar/DOI) [2022-06-06] Number of pages: 14 Place: Newport Beach, California, USA.

[9] J. Choi, K. Kim, D. Lee, and S. Cha. 2021. NTFUZZ: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 677–693. https://doi.org/10.1109/SP40001.2021.00114

[10] Jonathan Corbet. 2018. *Direct Memory Access and Bus Mastering.* O'Reilly Media, Inc. https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch13s04.html

[11] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3133956.3134069 event-place: Dallas, Texas, USA.

[12] William Engelmann. 2015. Universal Extractor 2. https://github.com/Bioruebe/UniExtract2/

[13] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, 1–15. https://doi.org/10.14722/ndss.2018.23326

[14] Owen S. Good. 2018. *Counter-Strike pro get caught cheating during a major esports tournament.* Polygon, Vox Media. https://www.polygon.com/2018/10/21/18006358/counter-strike-esports-cheating-shanghai-video

[15] Google. 2021. google/syzkaller. https://github.com/google/syzkaller original-date: 2015-10-12T06:05:05Z.

[16] Matt Hand. 2020. *Methodology for Static Reverse Engineering of Windows Kernel Drivers.* Specter Ops, Inc. https://posts.specterops.io/methodology-for-static-reverse-engineering-of-windows-kernel-drivers-3115b2efed83

[17] Johannes Kinder and Helmut Veith. 2010. Precise static analysis of untrusted driver binaries. In *Formal methods in computer aided design*. FMCAD Association, Lugano, Switzerland, 43–50.

[18] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. 2010. Testing Closed-Source Binary Device Drivers with DDT. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 12.

[19] Kaspersky Lab. 2018. *The Slingshot APT.* Kaspersky. https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/09133534/The-Slingshot-APT_report_ENG_final.pdf

[20] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. 2017. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1007–1024. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry

[21] Dominik Maier and Fabian Toepfer. 2021. *BSOD: Binary-Only Scalable Fuzzing Of Device Drivers.* Association for Computing Machinery, New York, NY, USA, 48–61. https://doi.org/10.1145/3471621.3471863

[22] Debasish Mandal. 2021. debasishm89/iofuzz. https://github.com/debasishm89/iofuzz original-date: 2014-03-16T17:17:46Z.

[23] Microsoft. 2017. *Driver Verifier.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/driver-verifier

[24] Microsoft. 2017. *Introduction to WDM.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-wdm

[25] Microsoft. 2017. *I/O Stack Locations.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/i-o-stack-locations

[26] Microsoft. 2017. *IO_STACK_LOCATION structure (wdm.h).* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_io_stack_location

[27] Microsoft. 2017. *IRP_MJ_DEVICE_CONTROL.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/irp-mj-device-control

[28] Microsoft. 2017. *Kernel-Mode Driver Architecture Design Guide.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/

[29] Microsoft. 2017. *Overview of Windows Components.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/overview-of-windows-components

[30] Microsoft. 2017. *Section Objects and Views.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/section-objects-and-views

[31] Microsoft. 2017. *Types of WDM Drivers.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/types-of-wdm-drivers#possible-driver-layers

[32] Microsoft. 2017. *User mode and kernel mode.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode

[33] Microsoft. 2017. *Using Nt and Zw Versions of the Native System Services Routines.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/using-nt-and-zw-versions-of-the-native-system-services-routines?redirectedfrom=MSDN

[34] Microsoft. 2017. *Using WDF to Develop a Driver.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/wdf/using-the-framework-to-develop-a-driver

[35] Microsoft. 2017. *WHQL Release Signature.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/install/whql-release-signature

[36] Microsoft. 2018. *Device Input and Output Control (IOCTL).* Microsoft. https://docs.microsoft.com/en-us/windows/win32/devio/device-input-and-output-control-ioctl-

[37] Microsoft. 2018. *DeviceIoControl function (ioapiset.h).* Microsoft. https://docs.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol

[38] Microsoft. 2018. *IoAllocateMdl function (wdm.h).* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-ioallocatemdl

[39] Microsoft. 2018. *KeBugCheckEx function (wdm.h).* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-kebugcheckex

[40] Microsoft. 2018. *MmMapIoSpace function (wdm.h).* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmmapiospace

[41] Microsoft. 2018. *MmMapLockedPages function (wdm.h).* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-mmmaplockedpages

[42] Microsoft. 2018. *ZwMapViewOfSection function (wdm.h).* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-zwmapviewofsection

[43] Microsoft. 2018. *ZwOpenProcess function (ntddk.h).* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-zwopenprocess

[44] Microsoft. 2019. *Bug Check 0x3B: SYSTEM_SERVICE_EXCEPTION.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/bug-check-0x3b--system-service-exception

[45] Microsoft. 2019. *Bug Check 0xBE: ATTEMPTED_WRITE_TO_READONLY_MEMORY.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/bug-check-0xbe--attempted-write-to-readonly-memory

[46] Microsoft. 2019. *Static Driver Verifier.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier

[47] Microsoft. 2020. *Bug Check 0xC4: DRIVER_VERIFIER_DETECTED_VIOLATION.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/bug-check-0xc4--driver-verifier-detected-violation

[48] Microsoft. 2020. *Bug Check Code Reference.* Microsoft. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/bug-check-code-reference2

[49] Microsoft. 2021. *memmove function from C runtime library.* Microsoft. https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/memmove-wmemmove?view=msvc-160

[50] MITRE. 1999. CVE - CVE. https://cve.mitre.org/

[51] MITRE. 1999. CVE - Download CVE List. https://cve.mitre.org/data/downloads/index.html

[52] CVE MITRE. 2021. *CVE-2021-21551*. MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-21551

[53] Sophos News. 2020. *Living off another land: Ransomware borrows vulnerable driver to remove security software*. Sophos. https://news.sophos.com/en-us/2020/02/06/living-off-another-land-ransomware-borrows-vulnerable-driver-to-remove-security-software/

[54] Dmytro Oleksiuk. 2021. Cr4sh/ioctlfuzzer. https://github.com/Cr4sh/ioctlfuzzer original-date: 2015-06-06T12:45:14Z.

[55] Hui Peng and Mathias Payer. 2020. USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Online, 2559–2575. https://www.usenix.org/conference/usenixsecurity20/presentation/peng

[56] Threat Post. 2020. *BYO-Bug Tactic Attacks Windows Kernel with Outdated Driver*. Thread Post. https://threatpost.com/byo-bug-windows-kernel-outdated-driver/152762/

[57] rzpnk.sys driver ZwOpenProcess Razer Synapse. 2017. *CVE-2017-9769*. MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9769

[58] The Register. 2020. *Windows kernel vulnerability disclosed by Google's Project Zero after bug exploited in the wild by hackers*. The Register. https://www.theregister.com/2020/10/30/windows_kernel_zeroday/

[59] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. 2012. SymDrive: Testing Drivers without Devices. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 279–292. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/renzelmann

[60] ESET Research. 2018. *LoJax: First UEFI rootkit found in the wild, courtesy of the Sednit group*. ESET. https://www.welivesecurity.com/2018/09/27/lojax-first-uefi-rootkit-found-wild-courtesy-sednit-group/

[61] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[62] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. 2020. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Online, 2541–2557. https://www.usenix.org/conference/usenixsecurity20/presentation/song

[63] Phillip Tinner. 2020. *Valorant Anti-Cheat Exploit Discoveries Can Net Players $100,000*. ScreenRant. https://screenrant.com/valorant-anti-cheat-exploit-hacker-cracker/

[64] The Verge. 2020. *The World's Biggest PC Games are fighting a new surge of Cheaters and Hackers*. The Verge. https://www.theverge.com/2020/5/6/21246229/pc-gaming-cheating-aimbots-wallhacks-hacking-tools-developer-response-problem

[65] Fish Wang and Yan Shoshitaishvili. 2017. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, IEEE, Cambridge, MA, USA, 8–9.

[66] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Toronto Canada, 1899–1913. https://doi.org/10.1145/3243734.3243844

[67] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Improving integer security for systems with {KINT}. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. USENIX, Hollywood, Los Angeles, CA, USA, 163–177.

[68] Ubuntu Wiki. 2019. *EFI/UEFI Boot Loaders*. Ubuntu. https://wiki.ubuntu.com/EFIBootLoaders

[69] Wikipedia. 2017. *Formal verification*. Wikipedia. https://en.wikipedia.org/wiki/Formal_verification

[70] Wikipedia. 2018. *Extended Validation Certificate*. Wikipedia. https://en.wikipedia.org/wiki/Extended_Validation_Certificate

[71] Wikipedia. 2018. *Peripheral Component Interconnect*. Wikipedia. https://en.wikipedia.org/wiki/Peripheral_Component_Interconnect

[72] Wikipedia. 2018. *Protection ring*. Wikipedia. https://en.wikipedia.org/wiki/Protection_ring

[73] Wikipedia. 2021. *Portable Executable*. Wikipedia. http://web.archive.org/web/20210604044347/https://en.wikipedia.org/wiki/Portable_Executable

[74] xst3nz. 2012. ioctlbf: Scanning IOCTLs & Fuzzing Windows kernel drivers. https://code.google.com/archive/p/ioctlbf/

[75] ZDNET. 2020. *Ransomware installs Gigabyte driver to kill antivirus products*. ZDNET. https://www.zdnet.com/article/ransomware-installs-gigabyte-driver-to-kill-antivirus-products/

A IMPACT OF VULNERABILITIES FOUND BY POPKORN

Vendor	Bugs **CVSS-7.8 (HIGH)**	Disclosure
Intel: One Boot Flash Utility (version blinded)	Privilege Escalation via `MmMapIoSpace` in `FLASHUD.sys`	CVE assigned
	Privilege Escalation via `MmMapIoSpace` in `IBSMUtil.sys`	Fixed (same CVE)
	Privilege Escalation via `MmMapIoSpace` in `imbdrv.sys`	Fixed (duplicate)
MSI: Dragon Center (version blinded)	Privilege Escalation via `MmMapIoSpace` in `MODAPI.sys`	CVE assigned
AMD: Radeon Software Adrenalin (version blinded)	Privilege Escalation via `MmMapIoSpace` in `atdcm64a.sys`	Fixed (internally)
	Privilege Escalation via `MmMapIoSpace` in `atdcm64a.sys`	Fixed (internally)
Total	6	

Table 5: Impact of vulnerabilities found per vendor (only bugs not under embargo are included).

B MANUALLY DETERMINED INPUT SOURCES IN SAMPLED CVES

Input source	Count
Bug in Windows (not in third-party driver)	36
Not a Windows driver bug	10
Unknown	43
DeviceIoControl	4
File System	1
Network	1
Custom Shader	1
DirectX	1
Named Pipe	1
Serial Line	1
SSDT hook	1

Table 6: Manually determined input sources in the 100 sampled CVEs where the automatic analysis did not detect an input source

C MOST FREQUENT WINDOWS API FUNCTIONS

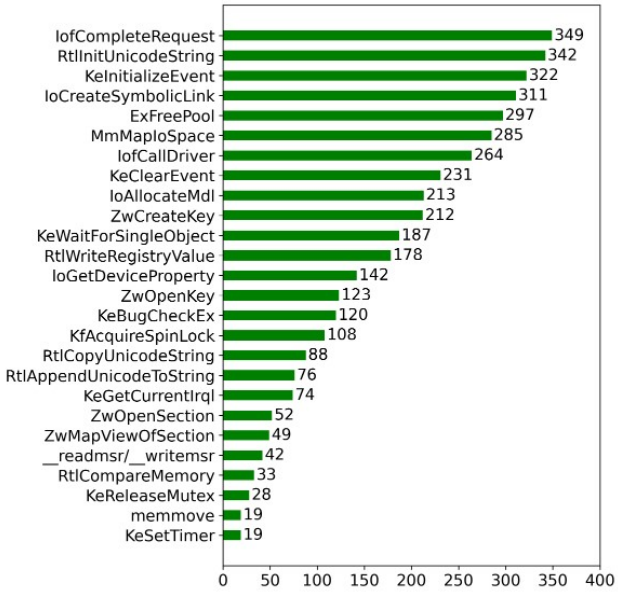


Figure 3: Top 25 most frequent Win-API functions extracted from our dataset of 3,094 drivers.

D ADDITIONAL CASE STUDIES

With the empirical evaluation done, in the following section, we present two additional case studies. The first demonstrates how POPKORN can be used to re-discover and verify the existence of known bugs in drivers, while the second discusses how POPKORN can be extended to support other critical sinks.

Case Study I: ZwOpenProcess Abuse. This case study elaborates on the bug described in CVE-2017-9769, which POPKORN was able to automatically discover, given the vulnerable driver file. Unfortunately, our driver dataset did not contain instances of this vulnerability in the wild, however POPKORN was able to rediscover the existing vulnerability in under four seconds. The decompiled driver source we refer to is shown in Listing 4. In this case, the `ClientId` is loaded from the user input and the `DesiredAccess` is hard-coded to allow reading and writing of process memory (constant `0x2000000`, Line 18). As a result, it becomes trivial for an attacker to take over any arbitrary process running on the system.

To implement this correctly, the driver developer has to i) not allow the user to control the process being opened, or ii) specify either the `OBJ_FORCE_ACCESS_CHECK` or the `OBJ_KERNEL_HANDLE` flags on the `ObjectAttributes`. This way, the kernel would verify that the calling user-mode process has the correct permissions to access the desired process or that only the kernel driver could use the handle. Then, even though the user-mode process successfully opens a privileged handle to the process, it cannot perform any malicious actions. However, since the driver developer left the `ObjectAttributes` uninitialized, the calling process can use the handle opened by the driver to inject malicious code into any privileged process [57].

```c
void *__fastcall zwopenprocess_func(void *inbuf)
{
  NTSTATUS status;
  struct _CLIENT_ID ClientId;
  struct _OBJECT_ATTRIBUTES object_attr;
  void *p_handle;
  // USER INPUT
  ClientId.UniqueProcess = *(void**)inbuf;

  object_attr.Length = 0x30;
  object_attr.RootDirectory = 0;
  object_attr.Attributes = 0;
  object_attr.ObjectName = 0;
  p_handle = 0;

  status = ZwOpenProcess(
           &p_handle, // process handle
           0x2000000, // r/w access
           &object_attr, //typical process attrs
           &ClientId); // process id
  if ( status < 0 )
  {
    printf("Failed to open process");
  }
  return p_handle;
}
```

Listing 4: ZwOpenProcess Abuse

Case Study II: Dell Privilege Escalation Bug. In this section, we demonstrate the extensibility of POPKORN by adding support for a recent bug found in a driver module from Dell's BIOS update utility. The example presented in this section is CVE-2021-21551 [52], which was disclosed (and patched) recently. The bug stems from the vulnerable usage of the `memmove` library function [49]. `memmove` is used to move a fixed number of bytes of memory from a source location to a destination memory block, and it has the following protoype:

```c
void *memmove(void *dest, const void *src, size_t count);
```

In this version of the vulnerable Dell driver, all of `memmove`'s three arguments (`destination`, `source`, and `count`) can be controlled by the user. In other words, insufficient access control checks result in user-provided buffers being directly used in the `memmove` routine. This is a classic write-what-where vulnerability, which lets an attacker overwrite any arbitrary pointer in kernel memory with user-supplied data.

The decompilation of the vulnerable driver code-snippet can be seen in Listing 5. In order to reach the `memmove` routine, a specific `IoControlCode` (`0x9B0C1EC8`) [36] needs to be supplied, and a four-field user buffer is used to fill all the parameters of `memmove`. A `DeviceIoControl` request serves as the entry point to exploit the vulnerability. After satisfying the `IoControlCode` checks, the `source`, `destination`, and `size` parameters are loaded from user buffer fields (Lines 23–25). These parameters are then passed to `memmove` directly, creating an arbitrary write vulnerability. This bug can be potentially exploited to escalate privileges to NT Authority/System [52].

Due to the flexible nature of POPKORN, adding support for detecting this bug was straightforward. We just had to add `memmove` to the list of critical sink functions that POPKORN targets. In particular, we added a check function that gets triggered whenever an invocation of `memmove` is encountered. This function examines the parameters of `memmove` and determines whether they are loaded from user-supplied buffers.

```c
1  // CVE-2021-21551
2  __int64 __fastcall vuln(DEVICE_OBJECT *device_object,
3                          IRP *irp)
4  {
5    // initialize buffers
6    .....
7    // entry in the I/O stack
8    iostacklocation = irp->Tail
9                    .Overlay.CurrentStackLocation;
10
11   // I/O Control Code
12   ioctl = iostacklocation->
13   Parameters.DeviceIoControl.IoControlCode;
14
15   // load buffer data from user input
16   *inbuf = &irp->AssociatedIrp.MasterIrp->Type;
17
18   // vulnerable ioctl
19   if (ioctl == 0x9B0C1EC8)
20   {
21     Dest = (void *)((*inbuf)[1] +
22             (*inbuf)[2]); // dest
23     Src = *inbuf + 3;      // ptr to source block
24     size = *inbuf - 0x18;  // bytes to move
25
26     // user controllable data is accepted as is.
27     memmove(Dest, Src, size); // Write-What-Where Bug
28     ......
29   }
30 }
```

Listing 5: Decompiled driver containing CVE-2021-21551.

With this new check in place, we run our system against the vulnerable driver: POPKORN automatically found the location of `IRP_MJ_DEVICE_CONTROL`—which reaches the vulnerable routine—and detected the bug in less than three seconds. Moreover, our analysis system reported the `IoControlCode` and buffer data that needs to be supplied to reach (and exploit) the vulnerable `memmove` routine.

E KERNEL API FUNCTIONS IN CVES

Kernel API Function	Count	Category	Why mentioned?
ProbeForWrite	26	mitigation	checks user-specified address
MmMapIoSpace	23	vulnerable sink	see Section 4.2
MmHighestUserAddress	22	mitigation	distinguish between usermode and kernel address
ProbeForRead	22	mitigation	checks user-specified address
IofCallDriver	18		stacktraces
RtlCopyMemory	16	vulnerable sink	buffer overflow if size controlled
HalDispatchTable	14	exploit target	syscall table, overwriting gives code execution
ZwDeviceIoControlFile	14		exploit code and stacktraces
ExAllocatePoolWithTag	12		driver code, integer overflows
KeBugCheckEx	11	vulnerable sink	assertion failure, often after memory corruption
IoGetRequestorSessionId	10		10 related CVEs, triggers crash after corruption
KiDeliverApc	8		stacktraces
IoCreateDevice	7		driver code
KeInsertQueueApc	7		stacktraces
RtlInitUnicodeString	7		driver code
DbgPrint	6		driver code
IoCompleteRequest	6		stacktraces, driver code
ExFreePoolWithTag	5		driver code
IofCompleteRequest	5		5 related CVEs, can bugcheck before Win7
PsLoadedModuleList	5		WinGDB output
ZwClose	5		driver code
ZwEnumerateValueKey	4	vulnerable sink	performs arbitrary write
ZwQuerySystemInformation	4	vulnerable sink	performs arbitrary write
ExAllocatePool	3		driver code, integer overflows
IoIs32bitProcess	3		3 related CVEs can only trigger in 32-bit driver
KeQueryPerformanceCounter	3		unrelated changelog, crypto seed
MmUnmapIoSpace	3		driver code
PsInitialSystemProcess	3	exploit target	leaked to locate privileged access token
ZwMapViewOfSection	3	vulnerable sink	see Section 4.2

Table 7: Kernel API functions mentioned by CVEs.