Intercept: Profiling Windows Network Device Drivers*

Manuel Mendonça Nuno Neves

University of Lisboa, Faculty of Sciences, LASIGE, Portugal
manuelmendonca@msn.com, nuno@di.fc.ul.pt

Abstract. Device drivers account for a substantial part of the operating system (OS), since they implement the code that interfaces the components connected to a computer system. Unfortunately, in the large majority of cases, hardware vendors do not release their code, making the analysis of failures attributed to device drivers extremely difficult. Although several instrumentation tools exist, most of them are useless to study device drivers as they work at user level. This paper presents Intercept, a tool that profiles Windows Device Drivers (WDD) and logs the driver interactions with the OS core at function level. The tool helps to understand how a WDD works and can provide support for several activities, such as debugging, robustness testing, or reverse engineering. Experiments using Ethernet, Wi-Fi and Bluetooth device drivers show that Intercept is able to record function calls, parameters and return values, with small overheads even when the device driver under test is subject to a heavy workload.

Keywords. Device drivers; profiling; dependability

1 Introduction

Device drivers (DD) play an important role in the computer industry as they are responsible for interfacing the multitude of devices that can be connected to a system. Therefore, their aggregated size can be a substantial part of modern operating systems. Nevertheless, most system administrators, users, and programmers still view them as an obscure and complex section of the operating system, which in part can be explained due to the DD necessity of addressing low level hardware details and OS internals. In the past, DD misbehavior has been pointed out as a prime cause for system crashes [3], and some researchers have showed that faults in DD can have a strong impact in the overall system dependability [4,5,6,7].

The recognized complexity associated with DD is aggravated as most vendors do not release openly the code, or even the hardware specifications. Therefore, the development, testing and analysis of DD becomes a complex task, and typically can only be achieved through the use of reverse engineering techniques and other forms of instrumentation. Although several instrumentation tools exist, most of them work at user level, making them useless to study the device drivers' behavior.

* This work was partially supported by the EC through project FP7-257475(MASSIF), by the FCT through the Multiannual program and project PTDC/EIA-EIA/113729/2009(SITAN).

In the paper we present Intercept, a tool that instruments Windows Device Drivers (WDD) by logging data about the interactions with the OS core. It operates without access to the driver's source code and with no changes to the driver's binary file. As its name indicates, the tool intercepts all function calls between the DD and the OS core, ensuring that various data can be collected, such as the name of the functions that are invoked, their parameters and return values, and the content of particular areas of memory. Although simple in concept, it enables the users to expose a DD behavior and data structures, which provide a practical approach towards its understanding.

In the case of DD involved with communications, which are the focus of the paper, Intercept can be used as a building block of other tools by providing the contents of packets and the context of their arrival/departure. For this purpose, Intercept can log the network traffic information in the format used by Libpcap [17], which can then be analyzed by popular tools such as WireShark [18]. Intercept can be very helpful in debugging processes since it gives a higher level vision of what is happening between the OS core and the driver, and at the same time offering information on the parameter contents and address locations. Combined with debugging tools from Microsoft, such as WinDbg [20], this data is useful to reduce the time for locating functions, OS resources and global variables. Currently, we are using Intercept as a component of a testing tool for DD. Some preliminary results are presented at the end of the paper.

2 Related Work

In the past, several tools have been proposed for various types of code analysis. For example, CodeSurfer [11] can perform program slicing to support a better understanding of the code behavior. BitBlaze [10] combines dynamic and static analysis components to extract information from malware. Other tools like Coverity [12], Path Finder [14] or CoreDet [13] rely either on C or Java language constructs and LLVM compilers [15] to transform the source code to their analysis format. Unfortunately, these tools depend on the existence of the source code. Binary programs have been addressed by RevGen [16], which translates them to LLVM intermediate representations, enabling the code to be checked with off-the-shelf analysis tools.

These tools, although producing valuable information, only reveal a part of the scope of the analysis, which is the static organization of the software. To obtain a vision over the dynamic behavior of the component, it is usually necessary to resort to debuggers or instrumentation tools that are able to trace the execution and record the instructions that were run. SytemTap [21] and Ftrace [22] are examples of existing tracing tools, but they only support the Linux OS.

Detours [1] is a library for intercepting arbitrary Win32 binary functions on x86 machines. The interception code is applied dynamically at runtime by replacing the first few instructions of the target function with an unconditional jump to a user-provided detour function. The removed instructions from the target function are preserved in a trampoline function, which also has an unconditional branch to the remainder of the target function. The detour function can either completely replace the target function or extend its semantics by invoking the target function as a subroutine

through the trampoline. Detours experiments were based on Windows applications and DLLs, but were not applied to device drivers.

PIN [2] is a software system that performs run-time binary instrumentation of Windows applications. PIN collects data by running the applications in a process-level virtual machine. It intercepts the process execution at the beginning and injects a runtime agent that is similar to a dynamic binary translator. To use PIN, a developer writes a "Pintool" application in C++ using the PIN API consisting of instrumentation, analysis and callback routines. The "Pintool" describes where to insert instrumentation and what it should do. Instrumentation routines walk over the instructions of an application and insert calls to analysis routines. Analysis routines are called when the program executes an instrumented instruction, collecting data about the instruction or analyzing its behavior. Callbacks are invoked when an event occurs, such as a program exit. Several applications were instrumented using PIN, such as Excel and Illustrator. PIN executes in user level ring3, and therefore can only capture user-level code. DynamoRio [24] is an example of dynamic binary translation technique similar to the one used by PIN [2].

NTrace [23] is a dynamic tracing tool for the Windows kernel capable of tracing system calls, including the ones involving drivers. The used technique is based on code modification and injection of branch instructions to jump to tracing functions. It relies on the properties introduced by the Microsoft Hot patching infrastructure, which by definition start with a `mov edi, edi` instruction. NTrace replaces this instruction with a two-byte jump instruction. However, due to the space constraints, the jump cannot direct control into the instrumentation routine. It rather redirects to the padding area preceding the function. The padding area is used as a trampoline into the instrumentation proxy routine.

Intercept uses an alternative approach to instrument device drivers in Windows, which requires no changes to the binary code and supports callbacks. It uses a DD loader to point all imported functions from a driver to its own interception layer. Callback functions registered by the driver are also captured and directed to the interception layer. No extra code needs to be developed for normal operation --- a complete log is generated describing how the driver behaves as a result of the experiments. However, extensibility is achieved by changing the actions performed by the interception layer, allowing more complex operations to be carried out.

3 Device Drivers

DD are extensible parts of the OS, exporting interfaces that support the interactions with the hardware devices. They are called when either the OS requires some action to be carried out by the device or the other way around. Depending on the type of device, the DD can operate in two different ways. In the first one, the DD accesses the device in a periodic fashion (pooling) --- the DD programs a timer with a certain value and whenever the timer expires the device is checked to see if it needs servicing (and proceeds accordingly). In the second way, the device triggers an interrupt to request the processor's attention. Each interrupting device is assigned an identifier

called the interrupt request (IRQ) number. When the processor detects that an interrupt has been generated on an IRQ, it stops the current execution and invokes an interrupt service routine (ISR) registered for the corresponding IRQ to attend to the request of the device. In either case, these critical pieces of code must be quickly executed to prevent the whole system from being stopped.

The rest of this section provides context on the operation of WDD. Some of this information was obtained by reading available literature, while other had to be discovered by reverse engineering the operation of Windows.

3.1 Windows device drivers

The *Windows Driver Model (WDM)* defines a unified approach for all kernel-mode drivers. It supports a layered driver architecture in which every device is serviced by a driver stack. Each driver in this chain isolates some hardware-independent features from the drivers above and beneath it avoiding the need for the drivers to interact directly with each other. The WDM has three types of DD, but only a few driver stacks contain all kinds of drivers:

- *Bus driver* – There is one bus driver for each type of bus in a machine (such as PCI, PnP and USB). Its primary responsibilities include: the identification of all devices connected to the bus; respond to plug and play events; and generically administer the devices on the bus. Typically, these DD are given by Microsoft;
- *Function driver* – It is the main driver for a device. Provides the operational interface for the device, handling the read and write operations. Function drivers are typically written by the device vendor, and they usually depend on a specific bus driver to interact with the hardware;
- *Filter drivers* – It is an optional driver that modifies the behavior of a device. There are several kinds of filter drivers such as: lower-level and upper-level filter drivers that can change input/output requests to a particular device.

The WDM specifies an architecture and design procedures for several types of devices, like display, printers, and interactive input. For network drivers, the *Network Driver Interface Specification (NDIS)* defines the standard interface between the layered network drivers, thereby abstracting lower-level drivers that manage hardware from upper-level drivers implementing standard network transports (e.g., the TCP protocol). Three types of kernel-mode network drivers are supported in Windows:

- *Miniport drivers* - A *Network Interface Card (NIC)* is normally supported by a miniport driver that has two basic functions: manage the NIC hardware, including the transmission and reception of data; interface with higher-level drivers, such as protocol drivers through the NDIS library. The NDIS library encapsulates all operating system routines that a miniport driver must call (functions `NdisMXxx()` and `NdisXxx()`). The miniport driver, in turn, exports a set of entry points (`MPXxx()` routines) that NDIS calls for its own purposes or on behalf of higher-level drivers to send packets.
- *Protocol Drivers* - A transport protocol (e.g. TCP or IP) is implemented as a protocol driver. At its upper edge, a protocol driver usually exports a private interface to its higher-level drivers in the protocol stack. At its lower edge, a proto-

col driver interfaces with miniport drivers or intermediate network drivers. A protocol driver initializes packets, copies data from the application into the packets, and sends the packets to its lower-level drivers by calling `NdisXxx()` functions. It also exports a set of entry points (`ProtocolXxx()` routines) that NDIS calls for its own purposes or on behalf of lower-level drivers to give received packets.

- *Intermediate Drivers* - These drivers are layered between miniport and protocol drivers, and they are used for instance to translate between different network media. An intermediate driver exports one or more virtual miniports at its upper edge. A protocol driver sends packets to a virtual miniport, which the intermediate driver propagates to an underlying miniport driver. At its lower edge, the intermediate driver appears to be a protocol driver to an underlying miniport driver. When the miniport driver indicates the arrival of packets, the intermediate driver forwards the packets up to the protocol drivers that are bound to its miniport.

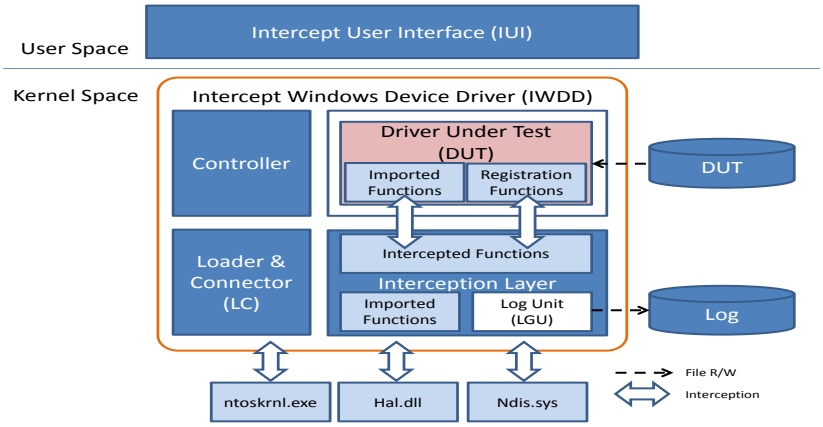
Windows drivers expose functions that provide services to the OS. However, only one function is directly known by the OS, as it is the only one that is retrieved from the binary file when the driver is loaded. By convention, the function name is `DriverEntry()`. This function is called when the OS finishes loading the binary code of the driver, and its role is to initialize all internal structures of the driver and hardware, and indicate to the OS the exported driver functions by calling `NdisMRegisterMiniportDriver()`. Example exported miniport driver functions to NDIS are: `MPInitialize()` and `MPSendPackets()`.

Generically, a packet transmission is accomplished in a few steps with NDIS. The protocol driver sends the packet by calling NDIS function `NdisSendPackets()`, which in turn passes the packet to the miniport driver by invoking `MPSendPackets()` exported by the miniport driver. The miniport driver then forwards the packet to the NIC for transmission by calling the associated `NdisSendPackets()`. On the other way around, when a NIC receives a packet, it can post a hardware interrupt that is handled by NDIS or the NIC's miniport driver. NDIS notifies the NIC's miniport driver by calling the appropriate `MPXxx()` function. The miniport driver sets up the data transfer from the NIC and then indicates the presence of the received packet to higher-level drivers by calling the `NdisMIndicateReceivePacket()`. The upper level protocol driver then calls `NdisReturnPacket()` to retrieve the packet.

3.2 Windows device drivers file structure

Windows normally organizes the information about a DD in several files. Files with the extension ".inf" contain plain text and are divided in several sections. They have relevant context data such as the vendor of the driver, the type and the compatibility with devices, and startup parameter values. They are used during driver installation to match devices with drivers and to find the associated ".sys" files. Files with the extension ".sys" are the binary executable images of the DD, and they are loaded to memory to provide services to the OS. The binary files follow the PEF file format [8], the same format used to represent applications and DLLs.

Fig. 1. Intercept architecture.



The PEF file structure contains binary code and dependencies from other software modules (organized as tables). The binary code is mostly ready to be loaded into memory and run. However, since it can be placed anywhere in memory, there is the need to fix up the relative addresses of the function calls. Functions that refer to external modules are located in the imported functions table. This table contains the names of the external modules (DLLs, .sys, .exe), the function names and the address location in the memory of the running system. The addresses are resolved by the Windows Driver Manager when it loads the driver for execution.

The driver is placed in execution by calling the `DriverEntry()` function. The address of this function is also obtained from the PEF file, and is located in the `AddressOfEntryPoint` field of the Optional Header section.

4 Intercept

Intercept logs information about the interactions between the OS core and the device driver under test (DUT). The data is collected during the whole period of execution, starting when the driver is loaded and ending when it is uninstalled. It includes among others, the list of functions that are used, the order by which they are called, and parameter and return values. This information is quite comprehensive, and it helps not only to understand the driver-OS core interactions, but also to realize how drivers deal with the hardware in terms of programming and access to specific storage areas.

4.1 Architecture

The architecture of Intercept is represented in Fig. 1. It can be divided in two main components: the *Intercept Windows Device Driver (IWDD)* and the *Intercept User Interface (IUI)*. The first is a Windows driver that provides all the necessary functions

to load, execute and intercept the DUT. The second is an application that allows users to setup the interception process and control the IWDD activity.

The components of IWDD are the following. The *Controller* provides an interface for the IUI application to control the behavior of the IWDD, allowing for instance the definition of the level of detail of logging and the selection of which functions should be logged. The *Loader & Connector* (LC) is responsible for loading the "DUT.sys" file into the memory space of IWDD. It also links all functions that the DUT calls from external modules to the functions offered by the Interception layer. The *Interception Layer* provides the environment for the DUT to run, and intercepts all calls performed by the OS to the DUT and the other way around. The *Log Unit (LGU)* receives the log entries from the Interception layer and saves them to a file. This is performed in a separate task to decouple the write delays from the remaining processing, and therefore increase the system performance.

Intercept is installed by replacing in the system the DUT with its own driver (the IWDD). When the OS attempts to load the DUT, in fact it ends up loading IWDD. Later on, IWDD brings to memory the DUT for execution. Setting up the interception of a DUT involves the following steps:

1. The user indicates the DUT of interest through the IUI interface, where a list of devices present in the OS is displayed;
2. The IUI locates the "DUT.inf" and "DUT.sys" files, and makes a copy of them to a predefined folder. A copy of the "IWDD.sys" file is also placed in the same folder;
3. The IUI replaces in the "DUT.inf" file all references to "DUT.sys" with "IWDD.sys". The IUI also removes references to the security catalogue, since IWDD is not currently digitally signed. This way, when the OS interprets the "DUT.inf" file, it will install "IWDD.sys" instead;
4. The Windows Device Manager (WDM) is used to uninstall the "DUT.sys", and then it is asked to check for new hardware, to detect that there is a device without a driver. At that time, the location of the predefined folder is provided, and Windows interprets the modified "DUT.inf" file. Since there is a match with the hardware identification of the device, it proceeds to load "IWDD.sys".

4.2 Start up process for interception

After "IWDD.sys" is loaded, the following sequence of actions occurs:

1. The WDM calls the `DriverEntry(DriverObject *drvObj, PUNICODE_STRING RegPath)` function of IWDD, so that it can initialize and register the callback functions. Parameter `*drvObj` is a complex structure where some of the exported callback functions can be registered. Parameter `RegPath` is the path of the Windows Register location where the driver should store information. Since the DD functionality is to be provided by the original DUT implementation, at this stage the control is given to the LC unit to load the DUT's code;
2. The LC unit interprets the "DUT.sys" file contents, relocates the addresses, and goes through the table of imported functions to link them to the Interception layer. Technically this is achieved by having in the Interception layer a table containing

entries with a “name” and an “address” for each function. The “name” is the Windows function name that can be found in the imported table of the DUT and the “address” is a pointer to the code of the function. The address of the function in the Interception layer is placed in the imported function table of the DUT's. In the end, all imported functions of the DUT point to functions in the IWDD.

3. Next, the “DUT.sys” binary is merged and linked to the IWDD. The LC unit also finds the address of the DUT’s `DriverEntry()`, which is then executed. As with any other driver, the DUT has to perform all initializations within this function, including running `NdisMRegisterMiniportDriver()` to register its exported functions to handle packets. However, since the DUT's imported functions were substituted by IWDD functions, a call to `NdisMRegisterMiniportDriver()` in fact corresponds to a call to `_IWDD_NdisMRegisterMiniportDriver()`[1]. In the particular case of this function, the DUT gives as parameters the callback functions to be registered in the NDIS library. In the Interception layer, the implementation of this function swaps the function addresses with its own functions, making the interception effective also for functions that will be called by the OS to the DUT.
4. When the DUT's `DriverEntry()` finishes, it returns a `drvObj` parameter containing potentially also some pointers to callback functions. Therefore, before giving control back to the OS, IWDD replaces all callback entries in `drvObj` with its own intercept functions, which in turn will call the DUT’s routines. This way this type of callback function is also intercepted.

4.3 Tracing the execution of the DUT

The DUT starts to operate normally, but every call performed by the OS to the DUT, and vice versa, is intercepted. The Interception layer traces all execution of the DUT, recording information about which and when functions are called, what parameter values are passed, which return values are produced and when the function exits. The log uses a plain text format and data is recorded to a file.

All functions implemented in the Interception layer make use of routines `_IWDD_DbgPrint()` and `_IWDD_Dump(char *addr, long size)`. The first works like the C language `printf()` function, and is used to write formatted data to the log file, such as strings and other information types. The second function is used to dump into the log file the contents of memory of a certain range of bytes starting at a given memory addresses. Together, these two functions can give a clear insight of the DUT’s and OS’s interaction.

Typically, the Interception layer creates a log entry both when entering and leaving a function. Whenever input parameter values are involved, they are also logged before calling the intended function, either in the DUT’s code or in the OS. Output parameters and return values are saved before the function ends execution. Complex structures, such as `NetBuffers`, `NetBufferLists` or `MDLs` [9], are decomposed by specific routines so that the values in each field of the structure can be stored.

[1] The prefix _IWDD_ is used to identify a function provided by the IWDD.

The interception of functions and the trace of its related information is a time consuming activity that may interfere with the DUT and the overall system performance. To reduce overheads, the storage process is handled by a separate thread. During the IWDD startup process, the LGU unit creates a queue and a dedicated thread (`DThread`), whose task is to take elements from the queue and write them into the log file. The queue acts as a buffer to adapt to the various speeds at which information is produced and consumed by the thread. The access to the queue is protected by a lock mechanism to avoid race conditions. A call to `_IWDD_DbgPrint()` or `_IWDD_Dump()` copies the contents of the memory to the queue, and signals the thread to wake up and store the information.

In the standard mode of operation, the log file is created when the thread is initiated. Each time the thread awakes, the data is removed from the queue and written to the file. When the file reaches a pre-determined value, it is closed and a new one is created. However, in case of a crash, the information in cache can be lost. To cope with this situation, the thread can also be configured to open, write synchronously and close the file each time it consumes data from the queue. However, this comes at the expense of a higher overhead.

5 Experimental Results

The objective of the experiments is twofold. First, we want to get some insights into the overheads introduced by Intercept, while a DD executes a common network task --- a file transfer by FTP. Second, we want to show some of the usage scenarios of the tool, such as determining which functions are imported by the drivers and what interactions occur while a driver runs.

5.1 Test environment

The experiments were performed with three standard drivers, implementing different network protocols, namely Ethernet, Wi-Fi and Bluetooth. Table 1 summarizes the installation files for each DUT.

Table 1. Device drivers under test.

Driver Type	Files
Ethernet	netrtx32.inf, rtlh.sys
Wi-Fi	netathr.inf, atrh.sys
Bluetooth	netbt.inf, btnetdrv.sys

The corresponding hardware devices were connected to a Toshiba Satellite A200-263 Laptop computer. The Ethernet and Wi-Fi cards were built-in into the computer, while the Bluetooth device was a SWEEX Micro Class II Bluetooth peripheral [19] linked by USB. In the tests, we have used Intercept both with Windows Vista and Windows 8.

The overhead experiments were based on the transmission of a file through FTP. The FTP server run in an HP 6730b computer. The FTP client was the Microsoft FTP client application, which was executed in the laptop together with Intercept. Different network connections were established depending on the DUT in use. For the Ethernet driver an Ethernet network of 100Mbps using a TP-Link 8 port 10/100Mbps switch was setup to connect the two systems. For the Wi-Fi and Bluetooth drivers an ad-hoc connection was established.

5.2 Overhead of Intercept

To evaluate the overheads introduced by Intercept, we have run a set of experiments consisting on the transfer of a file of 853548 bytes length between a FTP server and a client. Any file could have been used for the transfer. We selected this file because it was the first log produced by Intercept during the experiments.

For each driver five FTP transfers were performed, and the average results are presented in the tables. Table 2 summarizes the results for the execution time and transfer speeds. Column "Driver ID" represents the DUT, either in Windows Vista (xx_Vista) or in Windows 8 (xx_Win8). The columns under the label "Intercept off" display the average transfer time and average speed when the Intercept tool is not installed in the client system. The columns under label "Intercept on" correspond to the case when the Intercept tool is being used.

The results between Intercept off and on show a performance degradation, which was expected as Intercept records all the activity of the drivers, and performs tasks such as decoding parameter structures and return values of all functions. Nevertheless, these overheads are relatively small: between 2% and 7% for the Ethernet driver, 2% to 3% for the Bluetooth driver and 14% to 15% for the Wi-Fi driver. These observations were more or less expected since the Wi-Fi drivers have more imported functions, are longer in size and require more processing when compared with the other drivers. The same Bluetooth driver was used in both OS which can explain the similarity of the degradation. The differences between the overheads on the Ethernet and Wi-Fi networks can be related to changes in the drivers, since we have used the standard drivers that came with the Windows installation.

Table 2. FTP file transfer time and speed values (Time in seconds; Speed in Kbytes/second)

Driver ID	FTP Transfer				
	Intercept off (average)		Intercept on (average)		Time Overhead
	Time	Speed	Time	Speed	
Eth_Vista	0,198	6238	0,202	6204	2%
Eth_Win8	0,136	6503	0,146	5963	7%
WiFi_Vista	9,300	97	10,650	84	15%
WiFi_Win8	0,276	3076	0,314	2872	14%
Bth_Vista	5,890	145	6,012	142	2%
Bth_Win8	5,612	152	5,760	148	3%

During the experiments we saw that for each transmitted byte, Intercept generated between 9 to 23Kbytes of data. Not surprisingly the Wi-Fi driver was the one that

generated a higher amount of data, which can be interpreted as a synonymous of increased complexity.

Table 3. Top 5 most used functions by each driver.

Function	Eth Vista	WiFi Vista	Bth Vista
NdisMSynchronizeWithInterruptEx	-	69301	-
InterruptHandler	880	33931	-
MiniportInterruptDpc	-	32774	-
NdisAcquireReadWriteLock	-	6345	-
NdisReleaseReadWriteLock	-	6345	-
NdisMIndicateReceiveNetBufferLists	-	-	1032
NdisAllocateMdl	1096	-	-
NdisFreeMdl	1096	-	-
NdisAllocateNetBufferAndNetBufferList	1024	-	-
NdisFreeNetBufferList	1024	-	-
NdisAllocateMemoryWithTagPriority	-	-	520
NdisFreeMemory	-	-	520
MPSendNetBufferLists	-	-	503
NdisMSendNetBufferListsComplete	-	-	503

5.3 Understanding the dynamics of function calls

The dynamics of function calls during a driver's execution is determined by its work load. Intercept can support various kinds of profiling analysis about the usage of functions by a certain device driver under a specific load. For example, in our FTP transfer scenario, Table 3 represents the top 5 most called functions by each DUT from installation and until deactivation (in Windows Vista). Based on the number of function calls it becomes clear that the Wi-Fi driver is the one that shows more activity in the system. Focusing on the top 3 functions from this driver, the `NdisMSynchronizeWithInterruptEx` is the most used function. Drivers must call this function whenever two threads share resources that can be accessed at the same time. On a uniprocessor computer, if one driver function is accessing a shared resource and is interrupted, to allow the execution of another function that runs at a higher priority, the shared resource must be protected to prevent race conditions. On an SMP computer, two threads could be running simultaneously on different processors and attempting to modify the same data. Such accesses must be synchronized.

`InterruptHandler` is the second most executed function. This function runs whenever the hardware interrupts the system execution to notify that attention is required. From the 33931 interrupts, 32774 calls were deferred for later execution with `MiniportInterruptDpc`. By inspecting the remaining functions used by the Wi-Fi driver, which are lock related, it becomes evident that the driver is relying heavily on multithreading and synchronization operations.

Several other metrics can be obtained with Intercept, such as the minimum, average and maximum usage of each individual resource, DMA transfers, restarts, pauses, most used sections of the code, to name only a few. Intercept can also be employed when particular information needs to be collected. As an example, we wanted to find out what data is returned by the FTP server after the client connects. Fig. 2 shows a

call performed by the DUT to the OS notifying NDIS that a new frame has just arrived. In this case it is possible to observe the banner received from the FTP server, i.e., “220-Welcome to Cerberus FTP Server”.

Fig. 2. Looking in detail at a particular packet (excerpt).

```
ENTER - NdisMIndicateReceiveNetBufferLists
MiniportAdapterHandle..:0x895de190
NetBufferLists.........:0x87f86898
PortNumber.............:0x00000000
NumberOfNetBufferLists.:0x00000001
ReceiveFlags...........:0x00000002
NDIS_RECEIVE_FLAGS_RESOURCES
NetBuf.................: 0x87f86898
Dumping mdl.....: 0x882412b0
0x884a6a10  ff ff ff ff ff ff 00 27 13 4f 61 40 08 00 45 00 .......'.Oa@..E.
0x884a6a20  00 4e 01 87 00 00 80 11 24 ba c0 a8 c9 0d c0 a8 .N......$.......
0x884a6a30  c9 ff 00 89 00 89 00 3a f1 ff 91 92 01 10 00 01 .......:........
0x884a6a40  00 00 00 00 00 00 20 46 44 45 4e 46 44 46 50 46 ...... FDENFDFPF
0x884a6a50  44 45 4d 46 41 43 41 43 41 43 41 43 41 43 41 43 DEMFACACACACACAC
0x884a6a60  41 43 41 43 41 41 41 00 00 20 00 01 6f 6e 0d 0a ACACAAA.. ..on..
0x884a6a70  32 32 30 2d 55 4e 52 45 47 49 53 54 45 52 45 44 220-UNREGISTERED
0x884a6a80  0d 0a 32 32 30 2d 57 65 6c 63 6f 6d 65 20 74 6f ..220-Welcome to
0x884a6a90  20 43 65 72 62 65 72 75 73 20 46 54 50 20 53 65  Cerberus FTP Se
0x884a6aa0  72 76 65 72 0d 0a 32 32 30 20 43 72 65 61 74 65 rver..220 Create
0x884a6ab0  64 20 62 79 20 47 72 61 6e 74 20 41 76 65 72 65 d by Grant Avere
0x884a6ac0  74 74 0d 0a 00 56 00 03 00 01 00 00 00 02 00 32 tt...V.........2
0x884a6ad0  00 5c 4d 41 49 4c 53 4c 4f 54 5c 42 52 4f 57 53 .\MAILSLOT\BROWS
```

5.4 Understanding how drivers interact with the hardware

Intercept can also help to understand how specific hardware interactions are performed. The NDIS Library provides a set of I/O functions that a miniport driver calls to access I/O ports. These calls provide a standard portable interface that supports the various operating environments for NDIS drivers. For instance, functions are offered for mapping ports, for claiming I/O resources, and for reading from and writing to the mapped and unmapped I/O ports. Taking the Wi-Fi driver as an example, one can use Intercept to learn how the hardware initialization process happens. It starts when the OS invokes the drivers’ callback function `MPInitializeEx` (see Fig. 3).

Fig. 3. Call to MPInitializeEx to initialize the hardware (excerpt).

```
ENTER - NewMPInitializeEx
MiniportAdapterHandle.....: 0x8a16b0e8
MiniportDriverContext.....: 0x00000000
MiniportInitParameters....: 0x8c35b6d8
Header.Revision.....: 0x00000001
Header.Size.....: 0x00000028
Header.Type.....: 0x00000081
Flags.....: 0x00000000
IMDeviceInstanceContext.....: 0x00000000
MiniportAddDeviceContext.....: 0x00000000
IfIndex.....: 0x00000041
NetLuid.....: 0x00000000
NetLuid.Info.....: 0x00000000
NetLuid.Value.....: 0x00000000
AllocatedResources.....: 0x8a1e269c
AllocatedResources->Version.....: 0x00000001
AllocatedResources->Revision.....: 0x00000001
AllocatedResources->Count.....: 0x00000003
AllocatedResources->PartialDescriptors[00000000].Type.....: 0x00000003
AllocatedResources->PartialDescriptors[00000000].ShareDisposition.....: 0x00000001
AllocatedResources->PartialDescriptors[00000000].Flags.....: 0x00000080
CmResourceTypeMemory
AllocatedResources->PartialDescriptors[00000000].u.Memory.Start.....: 0xd4000000
AllocatedResources->PartialDescriptors[00000000].u.Memory.Length.....: 0x00010000
```

The OS passes several parameters to this function. One of them is the `MiniportAdapterHandle` so that whenever there is the need for the driver to call for some function, the OS is able to know which hardware the driver is referencing to (in this case, the reference is 0x8a16b0e8). All subsequent functions related with this driver will use this reference.

Another parameter is the resources allocated for the hardware. This allocation was performed automatically by the system according to the PCI standard, which releases the programmers from doing it. However, the driver only gets to know it when this function is called. In this example some of resources assigned to the Wi-Fi hardware were: Memory start: 0xd4000000 and Memory length: 0x00010000.

5.5 Understanding particular (complex) interactions with the OS

Intercept can be used to comprehend how certain complex operations are performed by the driver. For example, in Windows, a driver can remain installed but disabled. By analyzing the log produced by Intercept during the disabling process, it is possible to observe that the OS first calls the drivers' `MiniportPause` to stop the flow of data through the device. Second, the OS calls `MiniportHalt` to obtain the resources that were being utilized. Both these two functions were registered during the initialization process, at the time using the `NdisMRegisterMiniportDriver` function. Finally, the OS calls the `Unload` function to notify the driver that is about to be unload. The `Unload` function was also registered by the driver in the OS when the `DriverEntry` routine returned, by setting the address of this function in the `DriverUnload` field of the `Driver_Object` structure. As soon as the `Unload` function starts it is possible to observe in the log that the driver calls the `MPDriverUnload` callback function. When this function ends the unload process ends and the driver is disabled.

Another example corresponds to uninstalling the driver. With the information logged by Intercept, it was found that there is no difference between disabling and uninstalling a driver, except from the fact that uninstalling the driver removes it from the system.

The detailed information stored by Intercept in the log also helps to determine if all resources allocated by the driver are returned to the OS core. This can assist for instance to detect drivers with bugs. Table 4 represents the use of five resources utilized by the Wi-Fi driver. It is possible to observe a match between the number of resource allocations and releases, which gives evidence that the driver released all those allocated resources.

Table 4. Top 5 allocation/release resources functions.

Allocation function	#Calls	Release function	#Calls
_IWDD_NdisFreeIoWorkItem	1158	_IWDD_NdisAllocateIoWorkItem	1158
_IWDD_NdisMAllocateNetBufferSGList	1041	_IWDD_NdisMFreeNetBufferSGList	1041
_IWDD_NdisMAllocateSharedMemory	803	_IWDD_NdisMFreeSharedMemory	803
_IWDD_NdisAllocateNetBuffer	256	_IWDD_NdisFreeNetBuffer	256
_IWDD_NdisAllocateNetBufferList	256	_IWDD_NdisFreeNetBufferList	256

6 Using Intercept as a Component of a Testing Tool

Currently, we are developing a testing tool that uses Intercept as a building block. Due to its detailed logs, the tester can fully understand the driver's dynamics, and thus plan and design tests that target specific and elaborate conditions. The new tool uses a file that describes the test pattern. Whenever a function is intercepted by the Interception Layer, it calls the `_Inject_decison` function to evaluate the conditions and execute the test accordingly.

As a demonstration of the results of this ongoing work, an experiment was performed during the initialization of the Wi-Fi driver in Windows 8. The test targeted the `NdisMMapIoSpace` function that maps a given bus-relative "physical" range of device RAM. When successful, this function returns `NDIS_STATUS_SUCCESS` and the value of the output parameter `VirtualAddress` contains the start of the memory map. Other outcomes are exceptions that should be handled quietly.

Four test scenarios were planned by returning to the driver three possible exceptional values (as described in the Microsoft documentation) `NDIS_STATUS_RESOURCE_CONFLICT`, `NDIS_STATUS_RESOURCES`, `NDIS_STATUS_FAILURE` and one unspecified value `(NDIS_STATUS_FAILURE+1)`, while maintaining `VirtualAddress` equal to `NULL`. The DUT handled correctly the tests and ended quietly, and appropriately deallocated all resources, as confirmed by the Intercept logs.

Four additional test scenarios were performed with the same return values but assigning a specific value to `VirtualAddress`. These tests all resulted in a crash with the DUT being the culprit. It was concluded that the driver is using the value of `VirtualAddress` before checking the return value, which is worrisome in case Windows does not clear the `VirtualAddressis` field.

7 Conclusions

The paper presents Intercept, a tool that instruments WDD by logging the driver interactions with the OS at function level. It uses an approach where the WDD binary is in full control and the execution traced to a file recording all function calls, parameter and return values. The trace is directly generated in clear text with all the involved data structures.

An experiment with three network drivers was used to demonstrate some of the instrumentation capabilities of Intercept. The performance of the tool was also evaluated in a FTP file transfer scenario, and the observed overheads were small given the amount of information that is logged, all below 15%.

As is, Intercept gives a clear picture of the dynamics of the driver, which can help in debugging and reverse engineering processes with low performance degradation. Intercept is also currently being used as a building block of a testing tool. Preliminary results show the ability to identify bugs in drivers, by executing tests based on the knowledge obtained from the driver's dynamics.

8 References

1. G. Hunt, D. Brubacher, “Detours: Binary Interception of Win32 Functions”, In Proc. of the Conf. of USENIX Windows NT Symposium, 1999
2. A. Skaletsky, T. Devor, N. Chachmon, R. Cohn, K. Hazelwood, V. Vladimirov, M. Bach, “Dynamic Program Analysis of Microsoft Windows Applications”, In Proc. of the Int. Symp. on Performance Analysis of Systems & Software, 2010
3. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An Empirical Study of Operating System Errors”, In Proc. of the Symp. on Operating Systems Principles, October 2001.
4. M. Mendonça and N. Neves, “Robustness testing of the Windows DDK”, In Proc. of the Int. Conf. on Dependable Systems and Networks, June 2007
5. A. Albinet, J. Arlat, and J.-C. Fabre, “Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel”, In Proc. of the Int. Conf. on Dependable Systems and Networks, June 2004
6. J. Durães and H. Madeira, “Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation”, In Proc. of the Pacific Rim Int. Symp. of Dependable Computing, December 2002
7. A. Johansson, N. Suri, “Error Propagation Profiling of Operating Systems”, In Proc. of the Int. Conf. on Dependable Systems and Networks, July 2005
8. Microsoft, “Microsoft Portable Executable and Common Object File Format Specification”, February 2005
9. WDK 8.0, http://msdn.microsoft.com/en-US/windows/hardware/hh852362, July 2012.
10. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam and P. Saxena, “Bitblaze: A new approach to computer security via binary analysis”, In Proc. of the Int. Conf. on Information Systems Security, December 2008
11. G. Balakrishnan, T. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R.Gruian, S. Yong, C. H. Chen, and T. Teitelbaum. “Model checking x86 executables with CodeSurfer/x86 and WPDS++”, Computer Aided Verification, 2005
12. C. Henri-Gros, A. Kamsky, S. McPeak, D. Engler, “A few billion lines of code later: using static analysis to find bugs in the real world”, In Communications of the ACM, 53(2), 2010
13. T. Bergan, O. Anderson, J.Devietti, L.Ceze, and D.Grossman, “CoreDet: a compiler and runtime system for deterministic multi-threaded execution”, In Proc of the Int. Conf. on Architectural Support for Programming Languages and Operating Systems, March 2010
14. C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, M. Pape. “Combining unit-level symbolic execution and system-level concrete execution for testing NASA software”, In Proc. of the Int. Symp. on Software Testing and Analysis, July 2008
15. The LLVM Compiler Infrastructure, http://llvm.org/, Feb. 2013
16. V. Chipounov and G. Candea , “Enabling Sophisticated Analysis of x86 Binaries with RevGen”, In Proc. of the Int. Conf. on Dependable Systems and Networks, June 2011
17. Libpcap file format, http://wiki.wireshark.org, Feb. 2013
18. WireShark, http://www.wireshark.org/, Feb. 2013
19. Sweex, http://www.sweexdirect.co.uk, Feb. 2013
20. WinDbg, http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx, Feb. 2013
21. SystemTap, http://wiki.eclipse.org/Linux_Tools_Project/Systemtap, Feb. 2013
22. Ftrace, http://lwn.net/Articles/322666/, Feb. 2013
23. J. Passing, A. Schmitdt, M. Lowis, A. Polze, “NTrace: Function Boundary Tracing for Windows on IA-32”, In Proc. of the Working Conf. on Reverse Engineering, October 2009
24. D. Bruening, T. Garnett, S. Amarasinghe, “An Infrastructure for Adaptive Dynamic Optimization”. In Proc. of the Int. Symp. on Code Generation and Optimization, March 2003