Improving Robustness Testing of COTS OS Extensions*

Constantin Sârbu, Andréas Johansson, Falk Fraikin, and Neeraj Suri

Computer Science Department - Technische Universität Darmstadt,
Hochschulstr. 10, 64289 Darmstadt, Germany
{cs,aja,fraikin,suri}@informatik.tu-darmstadt.de
http://www.deeds.informatik.tu-darmstadt.de

Abstract. Operating systems (OS) are increasingly geared towards support of diverse peripheral components, both hardware (HW) and software (SW), rather than explicitly focused on increased reliability of delivered OS services. The interface between the OS and the HW devices is provided by device drivers. Furthermore, drivers have become add-on COTS components to support the OS's capabilities of widespread device support. Unfortunately, drivers constitute a major cause of system outages, impacting overall service reliability. Consequently, the testing of drivers becomes important. However, despite the efforts to develop appropriate testing methods, the multitude of possible system configurations and lack of detailed OS specifications makes the task difficult. Not requiring access to OS source code, this paper develops novel, non-intrusive support for test methods, based on ascertaining test progress from a driver's operational state model. This approach complements existing schemes, enhancing the level of accuracy of the test process by providing test location guidance.

Key words: software testing, robustness testing, COTS, operating system, device driver

1 Introduction

Device drivers[1] are commonly written to provide OS support for new computer peripherals in an commercial environment where time-to-market and low cost are key aspects. However, producing high quality drivers typically requires deep knowledge and understanding of OS internals and a proper testing strategy. Unfortunately, the details of the intricate internal structure are not available for many COTS OSs. Deadline pressure also reduces the time allocated for testing, creating a need for systematic and effective testing strategies.

Drivers are software components typically running in privileged mode, i.e., within the operating system kernel. Most COTS operating systems available today run kernel-mode device drivers.

* This research has been supported, in part, by Microsoft Research, EU FP6 NoE ReSIST and EU FP6 IP DECOS.

[1] "drivers" in Microsoft Windows, "modules" in Unix / Linux

If something goes wrong with the driver and the OS kernel is not able to properly handle the exception situation, the error can propagate across the OS, causing instability and potentially crash the entire system. Also, if the driver is the recipient of a faulty command issued by the kernel (for instance, as a result of an application request), a driver with no (or improper) error handling capabilities can crash, reducing the system's ability to use the related piece of hardware. All the scenarios described above lead to reducing (or even refusing) the level of service provided by the system.

Assuming the monolithic architecture of current COTS OSs, drivers are difficult to test due to the complex OS kernel structures they deal with. During normal operation of an COTS OS, a large number of running processes can interact in unforeseeable ways, mainly due to the unpredictable set of active processes.

For instance, testing an operating system under different loads may lead to completely different results as the operational context is different each time. To ensure a controllable testing process and repeatability of experiments, we need an accurate definition of how the tested operating system performs under certain, reproducible conditions.

Our work represents an effort to capture the functional execution patterns of a COTS driver under test, despite the non-deterministic behavior that characterizes its observable activity. The presented approach is based on architectural specifications of the communication between OS kernel entities and drivers, as defined by the Windows Driver Model (referred throughout this paper as *WDM*) [1]. We considered the Microsoft Windows operating system drivers as a case study, but our approach is general enough to be applied to other COTS operating systems with minimal changes (i.e., no modifications are required to the OS kernel or drivers), as long as the specification of the communication interface between OS and the tested driver is available.

Overall, the approach entails developing and identifying a driver state model, and locating relevant test states. Moreover, in this paper we are not trying to identify proper test cases. Instead, we provide guidance for existing test methodologies to better localize the current execution of the driver under test and monitor test progress. Therein, the development of a non-source code based and reproducible driver robustness testing support represent the contributions of this paper.

The paper is organized as follows: Sect. 2 presents the robustness problem we intend to solve. Section 3 localizes our work in the related research context. Section 4 introduces our system model and Sect. 5 explains the technical background of the considered drivers. Section 6 describes our approach in detail while Sect. 7 discusses the proposed testing coverage metrics. Section 8 presents the experimental work and Sect. 9 discusses various aspects related to them. We summarize presenting conclusions and future work in Sect. 10.

2 The OS Robustness Problem Statement

Currently, OS kernels have reached a certain design maturity and are now diminishing as the main cause for system failures. With numerous hardware producers rapidly introducing new peripherals onto the market, many new drivers are developed and installed onto the OS every day. These constitute relatively "immature" software compared to the rest of the OS kernel. Therefore, many faults can be present in the driver source code. High defect density combined with the fact that, for instance in Linux, about 70% of the total of lines of code are driver code [2], indicate a high likelihood that a driver-related fault is the cause of the overall system failure.

Additionally, the set of loaded drivers is likely to be different across computer systems. This aspect limits testing's effectiveness, given the difficulty to ensure representativeness of the test setup (in the driver development stage).

Testing of a COTS component, such as a driver, can be performed in different phases, either by the *component-provider* in the design/development phase or by the *component-user* in the system implementation phase [3]. Test campaigns of the component-provider may require access to the component's source code, whereas the component-user is usually limited to a black-box testing approach. This paper focuses on the latter case, considering that the component-user (i.e., system designer) wants to assess (and later improve) the robustness of a given driver in the context of a known system, without having knowledge about its implementation details. The only information assumed available for the driver under test is the interaction with the OS kernel (see Sect. 4).

In this paper the source code is considered not accessible. Therefore, we use a black-box testing approach, meaning that the only available information is the binary, i.e., the executable form of the driver, and the OS $\Leftrightarrow$ driver interface specification. As driver implementations vary considerably, the usage specification provided to the component-user along with the COTS driver is not required or used in the approach presented throughout this paper.

3 Related Work

Since the 1970s, the software testing research community has pursued determining the relative effectiveness between partition and random testing. Myers [4] considered random testing an inefficient method for finding faults. Later, Hamlet and Taylor [5] claimed that partition testing is successful "only when sub-domains with a high failure probability can be identified". In the early 1990s, Möller and Paulish [6] showed that software faults have a tendency to concentrate in certain parts of the code, so splitting the test inputs into sub-domains matching the most defective partitions of the code seems to be the right thing to do. Weyuker and Jeng proved analytically in [7] that partition testing effectiveness strongly depends on the failure rate associated with each sub-domain.

If these failure rates can be obtained, partition testing can be very efficient in finding faults. Though, when testing is targeting COTS components (i.e., source code is not available) this task is non-trivial.

In this paper we present an empirical method to divide the operational behavior of a driver into partially overlapping sub-domains. The method attempts to solve the COTS component robustness problem and relies on the functional specifications of the driver (rather than the source code). The testing progress can be evaluated based on the operational behavior of the driver under test under an expected workload, as in [8].

A similar approach was proposed in [9]. The paper introduces a load testing technique called Deterministic Markov State Testing for describing the operational model of a telecommunication system. The incoming and completion of five types of telephone calls define the state of the system. However, as the work was driven by the necessity to test the given telecommunication system, detailed knowledge of the system internals was required. This obviously limits the usability of the method for a different black-box system. In contrast, our approach makes no assumptions about the driver under test.

In the area of OS error propagation profiling, Johansson et. al [10, 11] used a different level of the OS $\Leftrightarrow$ driver interface in their experiments (i.e., the specification of functions imported / exported by the targeted driver). In this paper we use a shared memory communication scheme used by the OS $\Leftrightarrow$ driver interface as defined in Sect. 5. As the two methods are complementary, we intend to extend our approach in the future to include both of them.

4 System Model

Our system model is illustrated in Fig. 1. It describes the structure of a computer system equipped with a COTS OS acting both as a hardware resource manager and as an execution platform for applications. By *COTS OS* we mean any monolithic operating system, designed for general purpose and not for specialized needs (i.e., real-time, safety-critical or highly-available systems). We currently use Microsoft Windows XP as a case study, and extending this approach to other operating systems is part of our ongoing work.

The system model is divided into three layers: (a) user space, (b) OS kernel space and (c) hardware space. The two dashed lines represent communication interfaces between two neighboring layers. For instance, an application in user space might want to read the content of a file stored on the local hard-disk. The request will be passed to the OS by means of standardized calling methods defined in the *System Services* layer. Further on, the command is handled by the *I/O Manager*, which will transform it into a command for one of the system's *device drivers*. In this specific example the command is passed to the driver associated with the local hard-disk. The driver will access the hard disk by instructing the reading heads to move to the proper position on the magnetic disk, will store read data in a memory buffer which will be made available

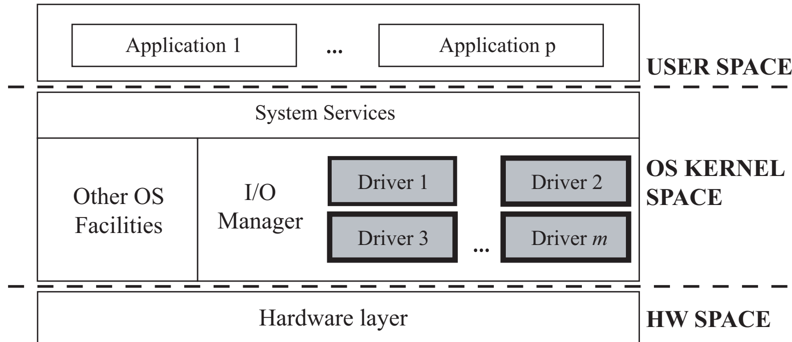

Fig. 1. A system featuring a COTS OS with m drivers.

to the application that initially issued the `read` request. Our focus is on the communication between the I/O Manager and the drivers.

We consider a driver as a collection of functions designed for controlling hardware. For Windows, drivers are implemented as Dynamic Link Libraries (.DLL or .SYS files). The files containing driver code, follow the Microsoft PE (Portable Executable) format and, therefore, publish a list of functions imported and/or exported from/to other libraries. The "import" list contains functions used by the driver, and the "exported" functions are entry points available to other software entities. We will refer to this level of interaction between drivers and OS as *software communication interface*. This functional behavior is captured in [10, 11].

Another level of interaction between these two entities is the *WDM (Windows Driver Model) communication interface* and will be detailed further in Sect. 5.1. In short, this is a shared memory communication scheme. Two participants communicate with each other by accessing specialized shared data structures and by notifying their counterpart after operating changes to these structures.

A third communication scheme exists, in which the driver/OS kernel is responding to events generated by the hardware, however this is not addressed in this paper.

Alternatively, we consider that each request from the I/O Manager will trigger the execution of a piece of driver code associated with the request type. The execution is finished when the driver returns the request back to the I/O Manager, together with a result and a status flag. The driver is *processing* a given request if the request was received and not yet returned to the caller.

5 Technical Background

5.1 Windows Driver Model and WDM-compliant drivers

To standardize and simplify the OS ⇔ driver communication inside kernel space, Microsoft created a unified communication framework, called WDM (Windows

Driver Model) [1]. WDM is a definition of the interface between the Windows OS kernel (NT4, 2k, Me, XP) and the drivers designed for these operating systems. It specifies the way that drivers are supposed to use certain structures inside the kernel, being developed mainly for forward-compatibility among Windows versions. By sharing this communication framework to the driver manufacturers, Microsoft establishes an unified method for building new drivers for its emerging OS versions. At the same time, WDM represents an attempt to standardize the way drivers execute in privileged kernel space, without harming the functionality of the operating system.

A *WDM-compliant driver* has to follow certain rules for design, implementation, initialization, plug-and-play, power management, memory allocation etc. A benefit of this approach for driver developers is the feature-set partitioning that WDM brings to focus: the driver must implement several standard routines, the rest of the code being application dependent. This is not always beneficial from a robustness viewpoint, as programmers tend to reuse large pieces of code when building new drivers (assisted by modern programming languages and IDEs) for reducing development time. This practice typically creates residual functionality that is not present in the specifications, decreasing the coverage of existing testing techniques applied by the component-user (who does not have access to the source code and thus cannot measure code coverage).

After a WDM-compliant driver code is loaded into memory and all the kernel structures related to that driver are initialized, it is ready to accept commands. These commands are issued by the I/O Manager (see Fig. 1), as a result of an application request. Such commands are called *I/O Request Packets* (IRP) and are OS kernel structures which contains a request code and all the necessary parameters needed by the driver to service the particular IRP request.

The driver receives the request and, following its processing, a result of the operation is returned to the caller. This data flow is depicted in Fig. 2.

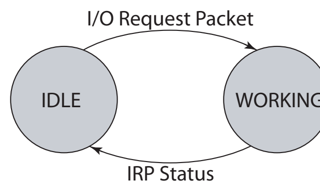

Fig. 2. Basic processing of a WDM-compliant driver.

5.2 WDM Kernel Structures

There are three structures defined in the WDM guidelines which are important for our approach. These structures are `DRIVER_OBJECT`, `IO_REQUEST_PACKET` and `IO_STACK_LOCATION`.

The former one, `DRIVER_OBJECT`, contains pointers to dispatch functions inside the driver code, every function being responsible for dealing with a specific IRP command. The latter two structures contain standard parameters of the request (`IO_REQUEST_PACKET`) and additional fields for parameters, respectively.

`IO_REQUEST_PACKET` and `IO_STACK_LOCATION` are strongly related and, therefore, when using the term *IRP* we refer to both of them, jointly.

a) `DRIVER_OBJECT`: This is a structure used by the operating system for publishing relevant information about a driver. Each driver is associated with an instance of this structure when the driver code is loaded to the kernel's memory. After loading, a standard method of the driver is executed (*DriverInit*) and the structure is populated with information published by the driver. Relevant to our approach are the pointers to functions inside the driver's code for managing each of the OS standard requests (IRPs). If the driver does not provide methods for a specific IRP, the associated pointer is set to `NULL`. Looking into the `DRIVER_OBJECT` structure, one can find out which IRPs are supported by the driver by simply determining the value of the associated pointer.
b) `IO_REQUEST_PACKET` and `IO_STACK_LOCATION`: These structures represent the trigger that makes the driver change its runtime mode from IDLE to WORKING (Fig. 2) and back. The receipt of an IRP by a driver initiates execution of a specific functionality. Whenever the result of the execution is available, the driver returns the two structures to the originator, after filling them with the result and setting a status flag. The current WDM version defines 28 IRP types, each of them associated with a particular operation supported by the driver (e.g., CREATE, READ, WRITE or CLOSE).

6 State Space and Operational Profile

From a robustness testing perspective it is important to accurately pinpoint what the system under test is processing at any chosen time instance. Therefore, the software testing community identified an increased demand of proper methods to precisely define the *"state"* of the system under test. In the case of static (deterministic) systems this task is relatively simple but for operating systems this is non-trivial. OSs are complex and dynamic, and often perform in a non-deterministic manner, mainly as a result of user input. A complete definition of the state must include information related to (a) the values of all used variables and (b) the functionality currently in execution.

In this paper we consider the total *state space* of a driver being infinite, therefore testing all states is infeasible. However, a driver is a special software component and can only be utilized by the user applications in a restricted manner (defined by the WDM interface). This means that a workload applied to the driver produces a particular operational behavior which triggers visiting only a certain state space subset. In our work we refer to this subset of the state

space as the *operational profile* of the driver. As long as the operational profile of a driver (related to a given workload) is known, we can concentrate the testing only to this subset of states. The main reason behind our approach is to avoid spending resources on testing functionalities that will likely not be exercised in the field. Consequently the testing effort is better focussed on the more likely operational states.

Obviously, this approach can primarily be used in component-users' testing scenarios (i.e., testing campaigns performed by COTS driver users) when the set of applications exercising the driver can be precisely determined. For instance, our approach is particularly useful for testing embedded systems using COTS operating systems. In this type of environments, the set of applications and the way they use the driver under test can be precisely determined, leading to an accurate outline of the driver's operational profile.

Nevertheless, our approach can prove itself useful to component-providers, too. For instance, their testing can make use of different user profiles to define sets of applications and characterize independent operational profiles for later guidance of their testing methods.

6.1 Driver State in Our Approach

As we assume a black-box driver under test, we try to provide state identification methods only with regard to the accessible communication interface. To avoid confusion with "driver state", we use the term "driver mode" in this paper. The *driver mode* is defined at a specific instance in time, with respect to the IRP requests the driver is executing at that moment. Thus, we use the information provided by ongoing functionality for guiding focused and reproducible testing campaigns of a driver.

6.2 Driver Mode

The I/O Manager serializes the IRPs that need to be sent to a driver D though some can be concurrently processed by the target driver. Hence, D can start processing the next request(s) received from I/O Manager, before a result of the current operation is available. Therefore, we represent the *mode* of the driver as follows:

Definition 1. *The mode of a driver D is defined by a tuple of predicates, each assigned to one of the n IRPs that the driver supports:*

$$M_D : \quad < P_{IRP_1} \; P_{IRP_2} \; P_{IRP_3} \; .. \; P_{IRP_n} >,$$

where P_{IRP_i} $(1 \leq i \leq n)$ shows if the driver D is performing or not the functionality associated with IRP_i, as below:

$$P_{IRP_i} = \begin{cases} \mathbf{1}, & \text{if } \textbf{\textit{performing}} \text{ the functionality triggered by } IRP_i \\ & \text{(the } IRP_i \text{ was received by the driver and the driver} \\ & \text{did not yet return a result to the caller)} \\ \mathbf{0}, & \text{otherwise} \end{cases}$$

At any instant in time, a driver can be processing multiple different IRPs. Therefore, we can build the complete set of tuples (as binary strings of length n) that represent all the different modes a driver can be in, the complete *state space* of the driver under test. Transitions between modes are triggered either by receipt of an IRP or by finishing the execution of an IRP.

6.3 Driver Modes and Transitions Between Modes

Let us consider a simple example of a driver supporting four different IRPs (Fig. 3), i.e., CREATE, READ, WRITE and CLOSE. After the driver is loaded, it will be in an IDLE mode (no IRP requests are currently processed), i.e., $< 0000 >$. The driver will stay in this mode as long as no IRP requests are issued. Let us assume that the driver receives a READ request. This request is associated with the second bit in the tuple, and therefore the driver switches to $< 0100 >$. We assume that the driver receives and finishes execution of the functionality associated with IRP requests in a sequential manner, i.e., only one bit from the bit string describing current mode can flip at a time. Therefore, from the mode $< 0100 >$ the driver can only switch to :

(a) $< 0000 > \rightarrow$ READ operation finished; driver returned to IDLE mode;
(b) $< 1100 >$ or $< 0110 >$ or $< 0101 >$ (any of the modes containing two set bits and having 1 on the second position) $\rightarrow$ READ operation not finished and another IRP was received.

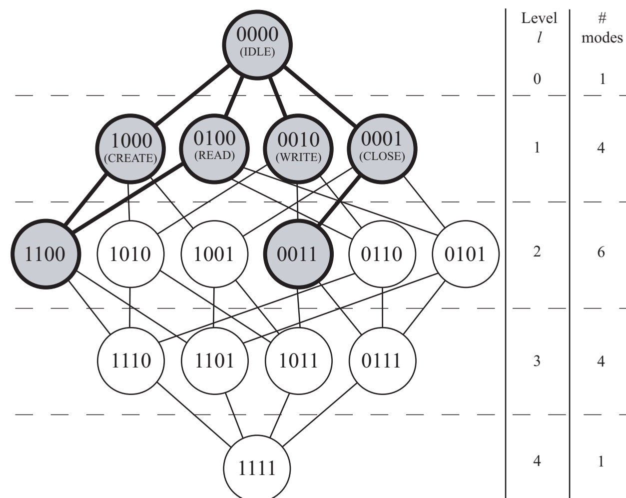

Fig. 3. The *state space* of a WDM-compliant driver in our model. The modes depicted in gray are the ones visited (using the bold edges) under a certain workload, so they represent the *operational profile* of the driver with respect to the given workload. Note: all lines represent bi-directional edges.

In our model, the size of the state space (the total number of modes) is $N = 2^n$, since a mode is defined as a binary string of length n. We represented the set of modes in a layered form (i.e., the modes represented by tuples containing l number of ones are on level l), having N_l modes on the l-th level, where

$$N_l = \binom{n}{l}, \; with \; 0 \leq l \leq n \tag{1}$$

The number of modes accessible from a current mode is restricted to a limited subset of modes. Assuming the IRPs are sent to a driver in a serialized manner, the driver can switch from a mode located on the level l to only few of the modes located on levels *immediately* above or below. More precisely, a driver can switch from the current mode *only* to modes whose tuples are at the Hamming distance of 1.

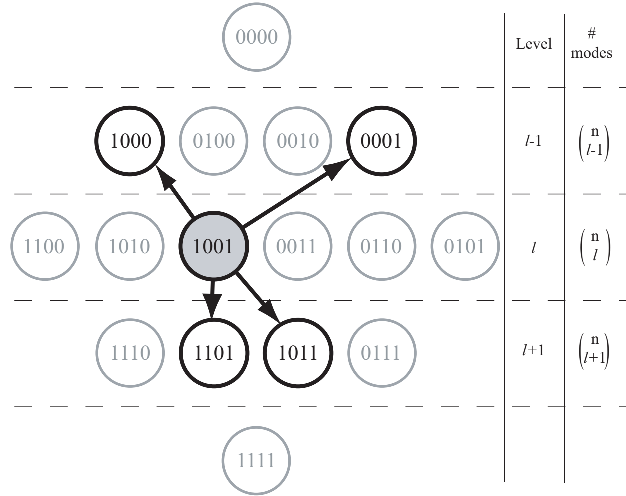

Fig. 4. A mode on level l ($< 1001 >$) can only switch to selected modes on level $l-1$ or $l+1$.

This behavior is represented in Fig. 4. Supposing that the driver under test is currently in mode $< 1001 >$, on level l (in this example $l = 2$) it can switch only to the highlighted modes on the levels $l-1$ and $l+1$. These modes are $< 1000 >$, $< 0001 >$, $< 1101 >$ and $< 1011 >$. Observe the fact that these modes differ from the current mode only by a single digit.

The number of transitions from a l-level mode is given in Table 1. Since a transition implies changing one digit in the mode, there are n possible transitions from a mode located on every level l in the model. Therefore, the total number of transitions that need to be covered in our model is

$$T = n \cdot N = n \cdot 2^n \tag{2}$$

Table 1. Number of transitions from a mode located on level l.

Current mode on level l	Can *only* switch to:	
	Modes on upper level ($l \rightarrow l$–1)	Modes on lower level ($l \rightarrow l$+1)
0	0	n
1	1	n − 1
2	2	n − 2
$\vdots$	$\vdots$	$\vdots$
l	l	$n - l$

7 Testing Coverage Metrics

Software defect prediction research showed that software faults are not uniformly distributed throughout the code; they tend to cluster in certain areas [6, 12]. Assuming that different inputs to the driver trigger different functionalities inside the driver, a logical implication is that each input can be associated with a certain likelihood of finding a fault.

In our approach each driver mode is associated with the execution of one or more disjoint pieces of code. A tester can use it to test each functionality separately (i.e., the modes on the level 1 are associated with single IRP requests) and in a combined manner. It makes sense to test the modes located on levels below 1 ($l \geq 2$) because there are faults that surface only when a functionality is executed in conjunction with others, for instance faults related to accessing shared resources. Therefore, a complete testing coverage of the operational profile defined by our methodology is highly desirable because it represents a quantifiable measure of the test status. Moreover, no modes and transitions outside the operational profile need to be tested.

As long as a driver is in a certain mode, it can only switch to a limited number of neighbors, located on the immediately upper and lower levels. A testing campaign aimed at transition coverage must exercise the entire set of transitions from a given mode, for all accessible modes. Ensuring traversal of all transitions in the operational profile can reveal errors that occur at entering or exiting a driver function. For instance, mode $< 1000 >$ in Fig. 3 can be visited even if the transition $< 0000 > \rightarrow < 1000 >$ was never traversed (i.e., only the path $< 0000 > \rightarrow < 0100 > \rightarrow < 1100 > \rightarrow < 1000 >$ was followed). Intuitively, this means that some of the transitions between modes might never be traversed, even if all the modes were visited, so *transition coverage* is more complete than *mode coverage* testing but is still not complete enough.

As drivers use memory to communicate and store variables while processing IRP-related functionalities, the sequence of IRP requests that put the driver in a certain mode is relevant for the testing process, too. In the example above, one should be aware of the fact that the two paths between $< 0000 >$ and $< 1000 >$ may lead the system to two different states (i.e., having different memory contents). Our method cannot capture the whole driver state (see Sect.

6), but is useful from a testing viewpoint because it can capture the effect of input sequences, i.e., ordering of requests. Knowing the paths between two modes (from inspecting the mode graph), a tester can develop test cases to traverse all of them to discover faults that occur as a result of certain input sequences. For instance, assuming the initial mode being the same, the sequence READ, CLOSE might work fine in contrast to the sequence CLOSE, READ that might yield an error.

For complete test coverage of our operational model, a testing process should ensure that (a) mode coverage, (b) transition coverage and (c) path coverage are satisfied at the same time [4].

7.1 Mode Coverage (MC)

Our approach improves the granularity of the testing process: the WORKING mode of a driver in Fig. 2 is split into several refined modes (see Fig. 3, all modes located on levels $l \neq 0$). Using our method, one can precisely pinpoint which functionality the driver is executing at a given instance, with regard to the received IRP requests. Moreover, the approach captures the concurrent execution of several requests, so a testing campaign can identify faults in procedures that only surface when the driver is executing them in conjunction with other procedures (modes on level 2 to n in Fig. 3).

If complete mode coverage is intended, a testing process can use a set of test cases which put the driver under test into every mode, thus implying that all the functionalities were executed at least once, separately *and* in conjunction with all other ones.

Definition 2. *A testing technique having 100% mode coverage (MC) ensures that every mode of the operational profile is tested.*

The mode coverage of a testing procedure can be quantified by relating the tested modes and the total number of nodes which form the operational profile, as defined below:

$$MC = \frac{|\text{tested modes} \cap \text{operational profile modes}|}{\#\text{ of operational profile modes}} \quad (3)$$

7.2 Transition Coverage (TC)

Since the state space is not large (WDM defines 28 distinct IRPs, see 5.2), covering all of it is feasible, given that a set of test cases capable of putting the driver into each mode can be devised. However, the mode coverage metric only ensures the execution of IRP-related functionalities, in all possible combinations, without considering the ways a driver can leave the current mode. Therefore, we need a more comprehensive coverage metric and we can use the concept of transition coverage.

Definition 3. *A testing technique with 100% transition coverage (TC) ensures that for each mode which belong to the operational profile all outgoing transitions (traversed under given workload) are tested.*

$$TC = \frac{|\text{tested transitions} \cap \text{operational profile transitions}|}{\#\text{ of operational profile transitions}} \tag{4}$$

Satisfying complete transition coverage requires a larger number of test cases than mode coverage, but it is a better measure of testing completeness and it implies also 100% mode coverage.

7.3 Path Coverage (PC)

The concept of path is used in our approach to express a sequence of IRP requests that lead the driver under test from an initial mode to another. For instance, in Fig. 3 are two 2-hop paths from mode $< 1000 >$ to $< 1110 >$ (an infinite number if we consider cycles and multi-hop paths):

1. $< 1000 > \rightarrow < 1100 > \rightarrow < 1110 >$;
2. $< 1000 > \rightarrow < 1010 > \rightarrow < 1110 >$.

Definition 4. *Path Coverage (PC) denotes traversing all the paths between two different modes of the operational profile, over any number of hops.*

We consider that the number of paths in our model is infinite, but one can use path coverage to compare the influence of following two different paths between two modes, assuming that the parameters which are different can be captured. For instance, if a value at a memory location associated with the tested driver is different for the two paths, this might be an indication of a fault.

Our approach can be used for pinpointing the execution behavior of a driver, combined with other tools for monitoring other parameters of the runtime environment. Together, they can provide insightful information, helpful for debugging or robustness evaluation of different drivers.

8 The `Serial` Driver - A Case Study

To validate the presented approach, we have conducted experiments to monitor the flow of IRP requests sent to a targeted driver. We determined the relative size of the operational profile in contrast with the complete state space of the driver.

We considered the serial driver provided together with Windows XP Professional SP2 operating system. The file containing the executable code of the driver was `serial.sys`, version 5.1.2600.2180. As an indication that it successfully passed Microsoft's quality tests, the driver was digitally signed by Microsoft.

The tests used to test the driver are available in the HCT (Hardware Compatibility Tests) and in the DDK (Driver Development Kit) and include reliability and stress tests.

For our experiments we utilized two Pentium4(HT)@2.80Ghz machines with 1Gb of RAM and an 56k external serial modem (Devolo Microlink 56k Fun II). To monitor the IRP request flow we used `IrpTracker.exe v2.1`, a free tool from Open Systems Resources Inc. This tool is capable of logging all the communication that takes place at the targeted driver interface. Both the receipt of an IRP and the return of the completed IRP to the originator are logged.

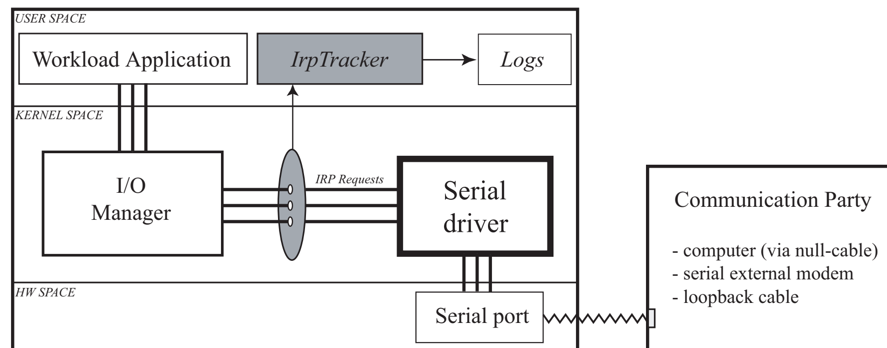

Fig. 5. The experimental setup. The workload application exercises the serial driver (via I/O Manager) by communicating data through the serial port to a second party. IRP requests are captured and logged by `IrpTracker`.

The experimental setup is depicted in Fig. 5. For each of the two experiments we present in this paper, a different application was used to generate relevant workload for the targeted driver. The workload generates IRP requests (via the I/O Manager) that triggers mode switches of the driver. We assume the driver is already installed in the OS kernel and initialized, so each experiment started with the driver in *idle mode*. After the IRP requests were captured and saved, the log files were sequentially parsed and processed and mode graphs similar to the ones presented in Sect. 6.3 were built.

8.1 Experiment 1 - Modem Diagnostic Driver-Usage Pattern

In this experiment we used an external modem, connected to the serial port. As workload we used `ModemTest v1.3`, a diagnostic test from PassMark Software. `ModemTest` sends data packets which are echoed back by the modem. Before sending any data, `ModemTest` first checks the serial port settings and then the modem itself. Any data received is verified to ensure its completeness and correctness. We have chosen this diagnostic tool to generate workload which is representative for modems and thus for serial port usage. At the same time, we

used the diagnostic tool as it generates a repeatable workload, which was one of the requirements in order to ensure the repeatability among different runs of the same experiment.

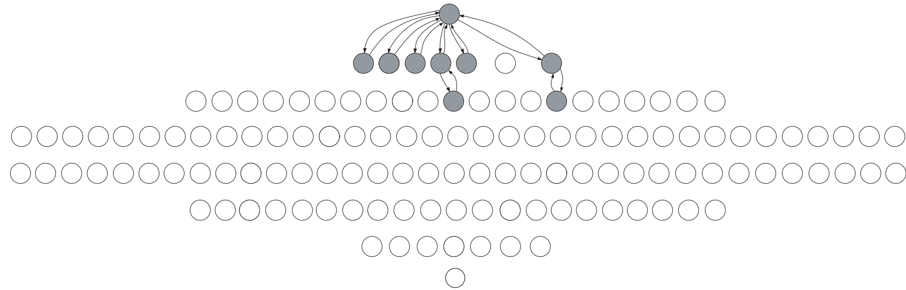

Fig. 6. An experiment using `ModemTest`. The visited modes and the traversed edges (defining the *operational profile*) are represented in gray. Note that only a small fraction of the graph was visited.

The observed behavior of the serial driver under this workload is represented by the mode graph in Fig. 6. For readability reasons, only the transitions between visited modes were depicted. Details can be observed in Fig. 7, which shows only the operational profile (visited nodes and traversed transitions).

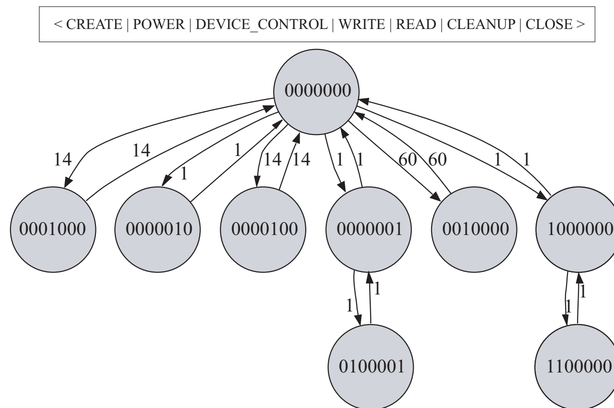

Fig. 7. An experiment using `ModemTest`. Edge label indicate the number of traversals. The upper box shows the corresponding IRP names.

The experiment issued a total of 186 IRP requests (issued and returned) using 7 distinct IRP requests (in order of appearance: CREATE, POWER, DEVICE_CONTROL, WRITE, READ, CLEANUP and CLOSE,). With a state

space of 128 modes and 896 transitions, only 9 modes and 16 transitions were actually visited, forming the operational profile for the considered workload. This means that 7% of the modes and 1.8% of transitions were traversed. Therefore, for properly testing the driver's robustness under the given workload, one has to focus only onto very few modes and transitions.

We executed the same experiment five times, each time using different baud rates (4800, 14400, 19200, 57600 and 115200). The other COM port settings were the default ones (8/N/1/no flow-control). Every time the observed flow of IRPs was identical and produced the same graph of visited modes (Figures 6 and 7). Moreover, the number of IRPs issued was also identical each time.

This fact indicates that the workload application is following a pattern, with each execution generating the same sequence of IRP requests. Additionally, the serial driver under test performed in a deterministic manner (responded to requests in the same manner every time). If a single IRP request would have been dropped or the driver would have finished executing the associated code in a different period of time (so that the driver would have started processing another IRP request), the graph would have had a completely different conformation.

Given the identical behavior of the serial driver in our experiments (under similar workloads), we assume that a system designer planning to use COTS drivers in a system can utilize our framework as a complement to an existing testing strategy. If the operating system, drivers and the set of applications to be used on the designed system are known (or at least a subset of them), the system designer can first build behavioral patterns that describe the manner the OS and the installed applications exercise a driver (the *operational profile*), as sets of visited modes and traversed transitions. If needed, one can associate counters to each node and transition, to observe which are frequently visited or, respectively, traversed. In the following experiment we show that this kind of information is useful for robustness testing.

8.2 Experiment 2 - Aggregated Workload

Under the assumption that the system designer has identified the set of applications to be used on the future system, we collected a set of applications (see Table 2) which we used to generate a workload for the targeted driver, the serial driver.

Similar to the previous experiment, we built the operational profile of the serial driver, for the selected application set. Table 3 contains the results of the aggregated workload experiment, as well the results for each application.

107456 requests were issued in total, out of which 10 were distinct, thus modes are represented by 10 bits. The full graph (not shown here for space reasons) has 1024 modes and 10240 transitions. Only 17 nodes and 35 transitions were visited, which corresponds to 1.66% of modes and 0.34% of transitions.

This result shows that even for a set of varied applications (some of them serial port benchmarks!) only a very small percent of modes and transitions is visited, even if a large number of IRP requests is issued. A second observation

Table 2. The set of applications used to generate relevant workload for the serial driver.

Application	HW at COM port	Description
`A1: BurnIn Test Pro v4.0`	Loopback serial cable	Reliability and stability test; RTS, CTS, DTR, DSR test, cycling all the baud rates
`A2: DirectX Diag. Tool v9.0c`	Computer (via serial cable)	DirectPlay test; text messages exchanged between machines
`A3: HyperTerminal v5.1.2600.0`	Computer (via serial cable)	Exchanged messages and 50k files
`A4: ModemTest v1.3`	External modem	See 8.1
`A5: Win XP modem diagnostics`	External modem	Windows XP queries the modem to check its capabilities
`A6: Win XP Device Manager`	External modem and COM port	Device Manager scans for hardware changes; the serial port and the modem are queried
`A7: Dial modem off`	External modem	Tried to dial a number when modem is off

Table 3. The results of considered applications, one by one and together (aggregated).

Application	**IRPs**		**Total (from which (%) visited)**	
	Issued	**Distinct**	**Modes**	**Transitions**
`A1`	98398	7	128 (7.03)	896 (1.78)
`A2`	220	8	256 (4.68)	2048 (1.07)
`A3`	6704	7	128 (7.81)	896 (2.12)
`A4`	187	7	128 (7.03)	896 (1.78)
`A5`	1326	8	256 (3.90)	2048 (0.87)
`A6`	146	8	256 (4.68)	2048 (0.92)
`A7`	476	8	256 (4.29)	2048 (0.97)
Aggregated:	**107456**	**10**	**1024 (1.66)**	**10240 (0.34)**

is the huge difference between the number of times transitions and modes were visited (see Fig. 8).

We observe that the modes located on lower levels in our model are not visited; we have only two visited modes on level $l = 3$. This can be an indication of the low degree of IRP interlacing, i.e., not many IRPs are permitted by the driver's design to execute concurrently.

9 Discussion

Operational Profile Size: Our experiments showed that the operational profile is only a very small part of the driver's mode graph. This has significant implications on testing coverage. First, it indicates the modes and transitions that have

Fig. 8. An experiment using the aggregation of applications described in Table 2.

a high likelihood to be reached in the field (the set of applications considered in the second experiment exercise most of the serial port capabilities). Second, we observed that few IRP requests are executed concurrently (the deepest visited level is $l = 3$). This may disclose internal constraints on how requests can be executed concurrently. The total number of test cases used by a test campaign can be reduced dramatically, since not all modes are reachable. Third, assuming that a testing technique starts in idle mode, the length of the sequence needed to bring the driver into the mode of interest for testing is very short.

IRP Sequences: Generating the IRP sequence necessary to put the driver in a mode of interest for the tester can be difficult, as the receipt and return of a request are events occurring in asynchronous manner. For visited nodes and transitions, the sequence of incoming requests can be saved (as they are structures in the kernel memory, see Sect. 5.2) and reproduced for the testing campaign. If testing is aimed at completeness (all modes / transitions to be tested), many request sequences may be unachievable. Therefore, we suggest using the following technique for testing.

Wave Testing: By monitoring how the driver under test performs under a relevant workload, two parameters that quantify the testing progress can be obtained:

1. the number of visited modes together with the values of the counters associated with each mode;

2. the number of traversed edges in the graph and their access counters.

"Wave testing" is now possible. In the first wave, the visited modes should be tested, accessing the same traversed edges. Obtaining a relevant workload for this step was discussed above. One can even split this wave in smaller units, i.e., first test the modes and transitions with the higher counter value.

After finishing the first wave, a set of modes accessible via one hop from the visited modes are identified. The problem of building an IRP sequence to reach next-hop modes is solved by appending a new IRP request to the sequences already available. The testing process continues by testing these modes and related transitions until no transitions can be traversed.

Limitations: While developing the presented approach, we aimed at generality of its applicability. Unfortunately, there are exceptions and drivers are as different in nature as the hardware they deal with. At the current stage, our method cannot deal with drivers that concurrently process several instances of the same IRP request type. On the same line, the transitions in our model are limited between modes located on neighboring levels; this means that no two or more IRP requests can start or finish their execution at exactly the same moment. Ongoing research includes experimental monitoring of different types of drivers on multiprocessor machines. Though, if such behavior is observed, the method presented in this paper can be extended accordingly.

10 Conclusions and Future Directions

Overall, our contribution provides means to quantify the test progress of already existing techniques for black-box testing of COTS drivers. The presented experiments show that only a small subset of the driver modes are actually exercised by several commercial applications (benchmarks) used to generate workload for the driver monitoring sessions. This result is important, indicating a relevant location to focus a testing method (i.e., put high priority on modes and transitions visited in the field). Therefore, this approach represents a significant improvement over random testing, given the tendency of faults to concentrate in specific areas of the code [6].

Moreover, we can now determine an operational profile of the driver under test, associating with each mode and transition a probability of occurrence in the field, as proposed in [8]. Additionally, the method is non-intrusive, requiring no access to the source code of the operating system and the COTS driver under test.

Future Work: We intend to further develop our method to embody more information which is observable from outside of COTS drivers. For instance, we will incorporate into our model the OS $\Leftrightarrow$ driver interface discussed in [10, 11]. As modes represent execution of driver code, the functionality associated with some of them may call methods defined in other software libraries of the OS kernel. We consider that the driver under test might fail as a result of a fault in the

external functions, not only due to faults in the own code. Therefore, the tester of the COTS driver can take advantage of the new, aggregated insight.

Another direction of future work will include monitoring several drivers in conjunction with relevant applications. We intend to investigate if there are patterns of driver usage that are independent of the workload. This fact might facilitate associating traversal probabilities with each mode and transition. Using this patterns, testing campaigns could concentrate on the modes and transitions accessed more frequently.

Acknowledgments

We would like to thank all DEEDS group's members for the valuable discussions, comments on early versions of the paper and feedback.

References

1. Oney, W.: Programming the MS Windows Driver Model. Microsoft Press, Redmond, Washington (2003)
2. Swift, M.M., Bershad, B.N., Levy, H.M.: Improving the reliability of commodity operating systems. ACM Transactions on Computer Systems. **Volume: 23** (2005) 77–110
3. Harrold, M.J.: Testing: A roadmap. In: International Conference on Software Engineering. (2000) 61–72
4. Myers, G.J.: The Art of Software Testing. 2nd edn. Wiley & Sons, inc. (2004)
5. Hamlet, D., Taylor, R.: Partition testing does not inspire confidence. IEEE Transactions on Software Engineering. **Volume: 16 , Issue: 12** (1990) 1402–1411
6. Möller, K.H., Paulish, D.: An empirical investigation of software fault distribution. In: First International Software Metrics Symposium. (1993) 82–90
7. Weyuker, E.J., Jeng, B.: Analyzing partition testing strategies. IEEE Transactions on Software Engineering. **Volume: 17, Issue: 7** (1991) 703–711
8. Weyuker, E.J.: Using operational distributions to judge testing progress. In: ACM Symposium on Applied Computing, New York, NY, USA, ACM Press. (2003) 1118–1122
9. Avritzer, A., Larson, B.: Load testing software using deterministic state testing. In: International Symposium on Software Testing and Analysis. (1993) 82–88
10. Johansson, A., Sârbu, A., Jhumka, A., Suri, N.: On enhancing the robustness of commercial operating systems. International Service Availability Symposium (ISAS). **Springer Lecture Notes on Computer Science 3335** (2004) 148–159
11. Johansson, A., Suri, N.: Error propagation profiling of operating systems. In: International Conference on Dependable Systems and Networks (DSN). (2005) 86–95
12. Fenton, N., Neil, M.: A critique of software defect prediction models. IEEE Transactions on Software Engineering. **Volume: 25, Issue: 5** (1999) 675–689