On Building (and Sojourning) the State-space of Windows Device Drivers

Constantin Sârbu and Neeraj Suri
Technische Universität Darmstadt
Hochschulstr. 10, Darmstadt, Germany
{cs,suri}@cs.tu-darmstadt.de

1 Introduction

An Operating System (OS) acts as a mediator between the hardware and software applications, providing for overall system services. However, with a tendency to support an ever increasing set of features and peripherals, the OS robustness emphasis often garners less attention. The OS interface to peripheral devices is implemented by specific SW components, termed as *device drivers* (DDs). Unfortunately, DDs also form the primary cause of OS system service failures despite sustained testing efforts. A basic explanation is the tight interaction required across the DD and the critical kernel structures resulting in inter-related failures. Moreover, as DDs are nowadays implemented as add-on components, they are usually delivered as black-box level binaries which are loaded on demand. Consequently, the need for black-box testing techniques to help identify DD weakness areas is increasingly important.

Our work focusses on techniques for profiling the execution behavior of kernel-mode DDs without assuming access to their source code. Such profiles highlight the operational paths of a driver for a given workload, helping uncover operational areas warranting concentrating testing efforts. The I/O communication interface between the OS and the DDs is monitored, revealing the subroutines invoked at runtime by the selected DD. As background work, in [3] we proposed a DD monitoring and analysis scheme using only the interface specification in order to build a state model for a DD and highlight its execution patterns. In [5] we defined a set of occurrence- and time-based quantifiers for the sojourned driver states and also proposed a methodology for test prioritization. In [4] we presented a mechanism for finding execution hotspots in DDs.

Related Work. Several approaches have used Software Implemented Fault Injection (SWIFI) techniques to test black-box OS extensions. The location of injection probes and the triggering instance of the actual fault injection are either empirically or randomly chosen. Though, in the area of defect localization in software, research showed that faults tend to cluster in certain parts of the OS code [2]. Thus, a strategy aimed at clustering the code into functionality-related parts and then testing them based on their operational occurrence is desired, especially when the resources allocated for testing are limited [6]. In [1] Avritzer and Larson proposed an approach to describe the load of a large telecom system that considered a state-based model to guide stress testing.

2 State Identification Basis for Device Drivers

In contrast, our proposed approach is specific to kernel-mode DDs, being also non-intrusive as the monitoring of the selected DD is using wrappers installed on the I/O communication interface between the OS kernel and a selected DD. Figure 1 depicts the chosen wrapping strategy.

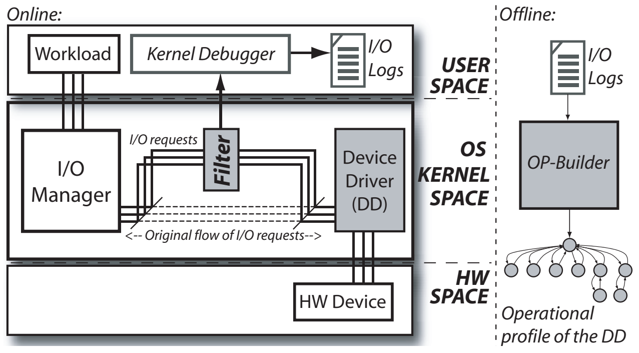

Figure 1. Our DD state identification strategy.

On the communication interface to the I/O Manager the OS kernel sends to the DD I/O requests generated by the activity of a workload. To capture the flow of requests at this interface, we have built a "*filter driver*" that is responsible for hijacking the I/O flow between the I/O Manager and the DD and recording it to log files. Offline, the log files are parsed and a state model of the driver's runtime activity is built in terms of visited states. We call this set of visited states the "*operational profile*" (OP) of the selected driver.

In Windows (used as a case study in our experiments) a DD can be assimilated to a toolbox. It contains a set of highly specialized subroutines, each being executed upon receipt of a specific I/O request. These requests are issued by the OS kernel and are termed "*I/O Request Packets*" (IRPs). They piggyback back and forth request parameters and operation results between the I/O Manager and the DD responsible for handling the respective I/O operation.

Our work [3,5] provides state identification methods with regard to the functionalities currently in execution. Using IRP receival and termination as state transition triggers, the "*driver mode*" is defined [3] as a tuple of predicates, each assigned to one of the n distinct IRP types supported by the DD (Windows XP SP2 defines 28 IRP types) as:

$$M^D : \ < P_{IRP_1} \ P_{IRP_2} \ .. \ P_{IRP_i} \ .. \ P_{IRP_n} >, \ where$$

$$P_{IRP_i} = \begin{cases} \textbf{1}, \text{ if the DD is currently } \textbf{\textit{performing}} \text{ the functionality triggered by the receival of } IRP_i; \\ \textbf{0}, \text{ otherwise.} \end{cases}$$

In [5] we introduced a series of quantifiers measuring the *occurrence rate* and the *temporal weight* of each DD mode, as a base for (a) driver state profiling and (b) test case prioritization. Under the assumption that each DD mode requires the same amount of test coverage, our quantifiers permit rankings based on mode occurrence weights as in Figure 2, where darker shades indicate higher weights.

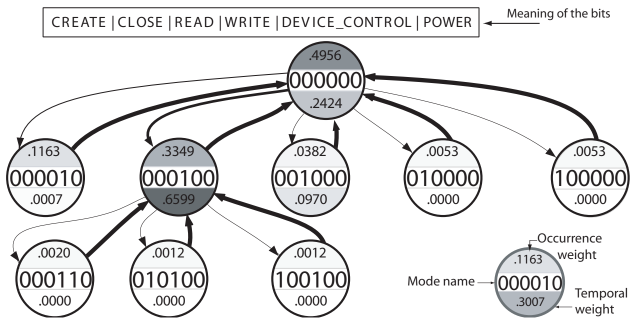

Figure 2. The OP for the floppy disk driver.

3 Preliminary Experimental Results

To validate the presented approach, we have conducted a series of experiments to monitor the operational behavior of diverse DDs provided with Windows XP SP2. The following observations result:

• **DD profile aids test space identification:** the set of visited DD modes and traversed transitions is very small, irrespective of the DD and the workload used. This observation provides for significant reduction of the test effort required to cover them, by selecting only the test cases that reach the modes of the obtained OP (i.e., *test case filtering*).

• **State quantifiers help state discrimination:** very few of the visited modes have high weights, warranting the usage of *test case prioritization* schemes based on the ranking among different modes belonging to the driver's OP. For instance, the modes with higher occurrence and temporal weights can be tested first (see Figure 2, the darker modes).

4 Discussion and Future Work

Using the lessons learnt from our monitoring approach, we are developing techniques for tuning an existing DD test campaign to primarily cover the identified execution hotspots. The selection of test cases should consider the information obtained from a prior driver profiling phase, in order to reduce the overall testing overhead. To expand our approach as a proper tool for driver profiling and testing in the absence of the source code, a more detailed analysis of the results presented above has to be pursued.

One of the research questions we currently investigate is how to forcibly bring the system into the modes of interest. As DDs perform in an arbitrary thread context and under the permanent influence of interrupts from the hardware devices, their runtime behavior is hard to predict. Especially for the modes where more than one IRP is active this is not a trivial endeavor, as the DD might finish the processing for one IRP before the other ones start. For instance, in Figure 2, the mode `010100` cannot be reached if the `WRITE` operation finishes before `CLOSE` is received. Hence, a simple test case that first calls `WRITE` followed immediately after by `CLOSE` might not necessarily bring the driver into the desired mode.

A possible solution might be to use the same workload that generated the driver's OP profile also for testing. As soon as a predecessor of the the desired mode is reached, the parameters of the intercepted IRPs have to be changed on-the-fly and then fed to the DD iff the malformed IRP leads the DD into the mode of interest. This approach requires the development of mechanisms that detects the current state and keeps track of it at runtime. Such idea might work if the actual mechanisms for changing the IRP requests' parameters and state awareness are kept to a very low overhead in order not to disrupt the IRP sequencing.

References

[1] A. Avritzer and B. Larson. Load testing software using deterministic state testing. In *Proc. ISSTA*, pp. 82–88, 1993.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Proc. SOSP*, pp. 73–88, 2001.

[3] C. Sârbu, A. Johansson, F. Fraikin, and N. Suri. Improving robustness testing of COTS OS extensions. In *Proc. ISAS*, Springer Verlag LNCS 4328, pp. 120–139, 2006.

[4] C. Sârbu, A. Johansson, and N. Suri. Execution path profiling for OS device drivers: Viability and methodology. In *Proc. ISAS*, Springer Verlag LNCS 5017, pp. 90–107, 2008.

[5] C. Sârbu and N. Suri. Runtime behavior-based profiling of OS drivers. TR-TUD-DEEDS-05-02-2007, 2007.

[6] E. J. Weyuker. Using operational distributions to judge testing progress. In *ACM Symposium on Applied Computing*, pp. 1118–1122, 2003.