On Equivalence Partitioning of Code Paths inside OS Kernel Components*

Constantin Sârbu†, Nachiappan Nagappan‡ and Neeraj Suri†

Technische Universität Darmstadt†
Darmstadt, Germany
{cs,suri}@cs.tu-darmstadt.de

Microsoft Research‡
Redmond, WA, USA
nachin@microsoft.com

Abstract

Commercial-off-the-shelf operating systems (COTS OSs) are increasingly chosen as key building blocks in embedded system design due to their rich feature-set available at low costs. Unfortunately, as the complexity of such OSs increases, testing key OS components such as device drivers *(DD) to ensure continuous service provision becomes increasingly challenging. Despite the improving test efforts targeting DDs, they still represent a significant cause of system outages as the test coverage is invariably limited by the inability to exhaustively assess and cover the operational states. Consequently, if representative operational execution profiles of DDs within an OS could be obtained, these could significantly improve the understanding of the actual operational DD state space and help focus the test efforts onto the execution hotspots.*

Focusing on characterizing DD operational activities while assuming no access to source code, our work enables profiling the runtime behavior of DDs solely based on I/O- and functional-call tracking. Such profiles are used to improve test adequacy against real-world workloads by enabling similarity quantification across them. The profiles also reveal execution hotspots in terms of functionalities activated in the field, allowing for dedicated test campaigns.

1 Introduction

An OS acts as a mediator between the hardware and software applications, providing for overall system services. However, the support for an ever increasing set of features and peripherals often overrides the needed emphasis on enhancing OS robustness. The OS interface to peripheral devices is implemented by specific software kernel components, termed as *device drivers* (DDs). Unfortunately, DDs are also the primary cause of OS service failures despite sustained testing efforts [9]. 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 and loaded on demand, at runtime. Consequently, the need for black-box testing techniques to help identify DD weakness areas in the operational mode is rapidly increasing, especially for embedded system design.

Our work focuses 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 DD for a given workload, helping uncover code areas warranting concentrated testing efforts. The communication interfaces between the OS and the DD are monitored, revealing the subroutines invoked at runtime by the selected DD. As background work towards DD state profiling, in [6] 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 [7] we defined a set of occurrence- and time-based quantifiers for the sojourned DD states and also proposed a methodology for test prioritization. In [8] we presented a methodology for finding execution hotspots in DDs. While in our previous work he have used equivalence partitioning of the code paths followed by DDs at runtime, the thrust in this paper is on identifying the key aspects of this clustering procedure to further enhance its value and usefulness for testing.

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 [3]. Though, in the area of defect localization in software, research showed that faults tend to cluster in certain parts of the OS code [1]. Thus, strategies aimed at revealing the functionality-related code parts and testing them based on their operational occurrence is desired, especially when the resources allocated for testing are limited [10], as they could find earlier the bugs likely to occur in the field.

2 OS and DD Profiling: Preliminary Results

As a conceptual basis for the proposed equivalence partitioning, our prior work [8] had explored OS profiling. In Windows (used as a case study in our experiments) a DD im-

*This research has been supported, in part, by Microsoft Research, EU Genesys and DFG TUD GK-MM.

plements a set of highly specialized routines, each being executed upon receipt of a specific I/O request from the OS. They piggyback back and forth request parameters and operation results between the OS and the DD responsible for handling the respective I/O call types (eg., READ, WRITE, CREATE, etc.).

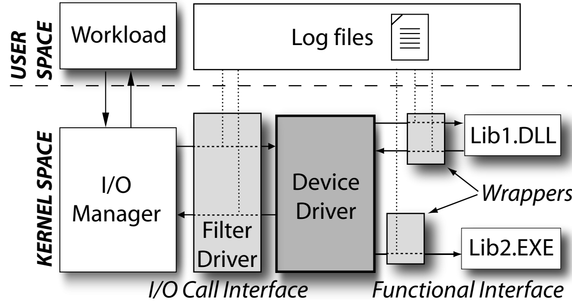

Figure 1. Our DD wrapping strategy.

Our DD profiling approach is non-intrusive as the monitoring of the selected DD is using wrappers installed on the communication interfaces between the OS kernel and the DD. On the *I/O call interface*, a filter driver intercepts and logs the I/O traffic, consisting in I/O calls sent by the OS kernel to the DD (*incoming*) and by the DD back to the OS (*outgoing*). On the *functional interface* customized wrappers capture the calls to functions implemented in driver-external kernel libraries which are called at runtime, as the DD is executing (Figure 1). Hence, the functional calls act as probes already present in the source code, revealing the code path taken by the DD at runtime. For performance and reliability reasons, Microsoft encourages the extensive use of functions and macros implemented in the existing kernel libraries (i.e., synchronization primitives, thread management etc.) [4, 5]. Hence, the DD code abounds with calls to external functions, validating the use of this information source for operational mode profiling.

After the monitoring phase, the log files contain blocks of external function calls, delimited by incoming and outgoing I/O calls. Knowing that each I/O call type triggers the execution of a specific I/O dispatch routine of the DD, *the sequences of functional calls represent abstractions of the code paths followed at runtime by the monitored DD*.

3 The Equivalence Partitioning Approach

From a testing perspective it is important to identify the code areas activated in the operational mode, thus permitting to accurately focus and prioritize subsequent testing activities. In [8] we distributed the code paths in equivalence classes based on their relative similarity (i.e., similar paths were grouped together) in order to emphasize the code areas exercised by a certain workload. To reduce the testing effort without losing adequacy, we conjecture that only one code path per cluster needs be tested. Our prior experimental results show that the code paths taken at runtime tend to differ significantly even if the DD is executing in the same mode[1]. This is indicative of the high DD code complexity, warranting our current effort toward providing testing assistance via methodical construction of the clusters. Early results indicate the following as being the key issues:

a. The similarity cutoff - the distance value among the intra-cluster objects (i.e., the degree of similarity among the objects of the same cluster). A small value results in many clusters containing fewer code paths (i.e., high testing overhead), while a too large value groups together code paths sharing little similarity, masking path diversity and thus reducing the adequacy of testing.

b. The similarity metric - the distance quantifier expressing the difference between any two code paths. It should properly capture and reward accordingly the features of the code paths being compared. In [8] we have empirically used as similarity quantifier an equally-weighted average of Levenshtein and Jaro-Winkler similarity metrics [2], as the emphasis was on presenting a valid methodology for DD profiling and not necessarily on its effectiveness for reducing testing effort.

To support an appropriate choice for the similarity metric, one of the research questions we currently investigate is which code constructs generate the ascertained similarity classes. Currently, we distinguish three primary similarity patterns (SP)[2] between any two captured DD code paths:

SP1: [xy**abc**z and m**abc**no] - share a common substring (very often the same prefix); we believe that the same helper (or initialization) routine is performed by both runs;

SP2: [xy**abc**z and mn**abcabcabcabc**o] - a common substring is repeated multiple times; we believe this is generated by a loop in the DD code;

SP3: [**abc** and xyzmno] - independent code paths; they should not be grouped in the same cluster as they need to be covered by different test cases.

Another research question that needs answering is how to decide which is the best-candidate code path for testing from each cluster, as the random choice might not always yield optimal results. For instance, if a cluster contains four code paths out of which only three are very similar to each other, a random choice might elect the fourth one for testing, thus reducing the effectiveness of the equivalence partitioning as a test reduction abstraction.

4 Identification of Frequently Used Kernel Services

Beside enabling execution hotspot discovery inside DDs, an interesting side-effect of our experimental work is revealing the set of kernel functions called by the targeted DD at runtime and the sequence thereof. These functions are OS services implemented in kernel libraries dynamically linked to the DD. As multiple kernel components intensively (and con-

[1] For the definition of *driver mode* see [6, 7]

[2] In the following, each character represents the encoding of a driver-external function called at runtime; for actual examples see Table 1.

currently) use the same set of kernel services, failures of such OS functions might lead to overall system failure. Hence, the information regarding the call frequency of the kernel services is useful for OS developers as it highlights the functionalities whose reliability and performance is critical.

Our preliminary DD profiling efforts show that the set of functions frequently used at runtime by a DD is relatively small. For instance, for the floppy disk driver studied in [8], only 20 functions make 99.97% of all the called functions, see Table 1.

Table 1. Twenty function calls accounting for 99.97% of the calls recorded for the floppy disk driver (Windows XP SP2), sorted in descending order on occurrence.

#	Function Name	#Occ.	[%]
1	`ExAcquireFastMutex`	60414	18.40
2	`ExReleaseFastMutex`	60414	18.40
3	`IofCallDriver`	45976	14.00
4	`KeInitializeEvent`	40777	12.42
5	`IoBuildDeviceIoControlRequest`	40771	12.41
6	`ExInterlockedRemoveHeadList`	22007	6.70
7	`ExInterlockedInsertTailList`	16123	4.91
8	`IofCompleteRequest`	11562	3.52
9	`KeWaitForSingleObject`	11032	3.36
10	`KeReleaseSemaphore`	11003	3.35
11	`MmMapLockedPagesSpecifyCache`	8178	2.49
12	`MmPageEntireDriver`	24	0.01
13	`MmResetDriverPaging`	23	0.01
14	`KeGetCurrentThread`	10	0.00
15	`KeSetPriortyThread`	10	0.00
16	`ObfDereferenceObject`	10	0.00
17	`ObReferenceObjectByHandle`	10	0.00
18	`PsCreateSystemThread`	10	0.00
19	`PsTerminateSystemThread`	10	0.00
20	`ZwClose`	10	0.00

This partial result indicates also that an approach aiming at testing the robustness of the OS kernel services should not select the targeted functions randomly (for instance, such a random strategy is used in [3]). Instead, the targeted functions should be selected as a result of a profiling step similar to the profiling presented in our experimental studies in order to focus testing onto the services which are *actually* called by the DD in the operational phase. We believe that such a procedure saves testing resources while increases the test adequacy and also the likelihood to find earlier the most relevant defects for the operational mode.

5 Discussion and Current Work Directions

By simultaneously monitoring the communication interfaces of a selected DD (the "*I/O Call*" and "*Functional*" interfaces in Figure 1), we identified the distinct code paths taken by the DD without requiring access to its source code. As the captured code paths share a significant amount of similarity, they can be grouped in equivalence classes. The equivalence classes represent useful abstractions (execution hotspots) as they considerably simplify DD testing: if one code path from a cluster is tested, then all other code paths belonging to the same cluster are also considered tested.

The monitoring of the functional call interface of a DD revealed also another class of execution hotspots. This is represented by the set of intensely used kernel services implemented in OS libraries, therefore external to the monitored DD. The information (frequency, sequence, patterns, etc.) about the execution hotspots of the kernel libraries involved in the communication with the DDs might reveal fundamental defects in OS structures, as well as lead to performance or reliability enhancements.

To answer the research questions mentioned in this paper and to expand our approach as a tool for DD profiling and testing in the absence of the source code, a detailed analysis of the presented equivalence partitioning aspects has to be pursued. Hence, we are currently examining the source code of several DDs to identify the coding patterns that generate the observed behavior.

Using the lessons learnt from our DD profiling approach, we are also developing techniques for tuning existing DD test campaigns to primarily cover the identified execution hotspots. The selection of test cases should consider the information about the followed code paths obtained in the profiling phase, in order to reduce the overall testing overhead.

Also, we intend to map the obtained code paths to the control flow graphs of the DDs. This serves as validation of our black-box profiling methodology by quantifying its capacity to disclose the followed code paths. At the same time, we conjecture that this evaluative approach provides for a proper comparison of the available black- and white-box test methods for DDs from the code coverage perspective.

References

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

[2] W. W. Cohen, R. Pradeep, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IJCAI*, pp. 73–78, 2003.

[3] M. Mendonca and N. Neves. Robustness testing of the windows ddk. In *Dependable Systems and Networks (DSN)*, pp 554–564, 2007.

[4] W. Oney. *Programming the MS Windows Driver Model*. Microsoft Press, 2003.

[5] P. Orwick and G. Smith. *Developing Drivers with the Windows Driver Foundation*. Microsoft Press, 2007.

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

[7] C. Sârbu, A. Johansson, N. Suri, and N. Nagappan. Profiling the operational behavior of OS device drivers. In *ISSRE*, 2008 (to appear).

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

[9] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, 2005.

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