Execution Path Profiling for OS Device Drivers: Viability and Methodology*

Constantin Sârbu[1], Andréas Johansson[2] and Neeraj Suri[1]

[1] Department of Computer Science – Technische Universität Darmstadt,
Hochschulstr. 10, D–64289 Darmstadt, Germany
{cs, suri}@cs.tu-darmstadt.de
[2] Department of Mechatronics and Software – Volvo Technology Corporation,
Sven Hultins gata 9C, SE–41288 Göteborg, Sweden
andreas.olof.johansson@volvo.com

Abstract. Operating Systems (OSs) mediate across the hardware and software applications, leading to overall system service provision, but often sacrifice service robustness while favoring increasing feature richness and peripheral support. The OS interface to peripherals is implemented by components termed as *Device Drivers* (DDs). Unfortunately, despite extensive testing, DDs continue to constitute the prominent cause of system service failures.
To find DD's weakness areas, this paper proposes a novel technique for profiling kernel mode DDs execution paths. Such profiles highlight the frequently used parts of a driver for a workload, helping identify redundant tests. The communication interfaces between the OS and DDs are simultaneously monitored, revealing the kernel functions invoked at runtime and the followed code paths. To highlight execution hotspots, a cluster analysis scheme using string similarity metrics is proposed to distribute the code paths into equivalence classes, reflecting the occurrence weights of both kernel functions and code paths.

Keywords: Operating System, Device Driver, Code Path Profiling, Cluster Analysis, Black-box Testing.

1 Introduction

COTS OSs are invariably required to balance the tradeoff between service dependability and service performance. Often performance aspects are favored in order to offer extensive support for a wide spectrum of applications and peripherals. The OS interface to peripherals, namely device drivers (DDs), are typically produced by third-party developers, often lacking the necessary skill and knowledge required to develop high quality and robust DDs. Moreover, under the

* This research has been supported, in part, by Microsoft Research, EU FP6 NoE ReSIST and DFG TUD GK-MM.

pressure to fulfill market demands, resources allocated to DD testing are often limited. Thus, while OS kernels have reached a certain maturity, the DDs are prematurely released and therefore are more likely prone to failures, affecting the overall provisioning of OS' service robustness.

With hundreds of devices attached to each ordinary computing system (about 250 in a Windows XP or Vista installation [1]) the drivers' code represents a significant share of the total OS code. In Linux, for instance, about 70% of the total lines of code belongs to DDs [2]. Given the immaturity of their code, this trend suggests that driver code is responsible for many OS service outages. This observation is confirmed by the OS reliability research community's results from several independent [3], academic [2, 4–7] and industry [8] sources.

As DDs coexist in privileged space with critical OS kernel structures, an error in a defective DD can propagate to the kernel, eventually leading to degraded OS service level or even generalized system failure. Recent studies [9, 10] have shown that OS kernels are permeable to error propagation, mostly due to the fact that in kernel space various components communicate under a "gentleman's agreement". This means that, for the sake of performance, kernel components perform only minimal (if any!) parameter validation, assuming that their communication parties are error-free and non-malicious. DDs should also follow this policy, hence passing the responsibility of producing error-free code to driver developers. This means that, beside programming experience, driver developers have to possess a deep understanding of OS kernel intricacies and be fully aware of the DD's runtime context.

It is reasonable that system integrators themselves test the DDs installed in their systems to verify that the specified level of service and reliability is provided. Typically, *black-box* testing is the only viable approach. Therefore, working continually under deadline pressure, system integrators limit DD testing to simpler acceptance and integration tests.

Execution profiling information is an important prerequisite for helping rigorous DD validation. It is an abstract model describing how a DD behaves under the influence of external stimuli. As such it can help DD testers identify which part of the DD code is most exercised for a representative workload. This can be used to guide selection of test cases, by focusing on the *most frequently used parts in an operational setting*, which may substantially differ from statically selected test cases.

Regardless, the ability to identify DD execution profiles increasingly represents a serious technical challenge as: (a) the access to the OS kernel space is limited (debugging is non-trivial); (b) the access to source code is limited (usually, testers cannot access the source code of the tested object); and (c) envisioning the runtime environment for COTS DDs is difficult (virtually each individual computing system has its own unique set of HW and SW components).

Paper Emphasis and Contributions

With the overall aim to enhance OS robustness, in this paper we develop a profiling methodology for kernel-mode DD execution paths by considering an

additional communication interface alongside with the I/O requests considered in our prior work [11, 12]. In this communication paradigm, at runtime, a DD acts as a consumer of the services (i.e., functions) provided by various kernel libraries. Therefore, a DD's runtime activity can be defined by the sequences of calls made to external functions. As DDs act on kernel calls, the call sequences are delimited by the I/O requests generated by the OS, and thus infer the execution path taken in the DD's code, helping to evaluate and to compare the effects of different workloads (i.e., test suites or individual test cases) by revealing execution hotspots. The presented process for caption and evaluation of the call traces does not require source code access to any of the involved components.

The presented results show a key phenomenon, the tendency of call traces to cluster with respect to the code being executed. We consequently present a cluster analysis method to ascertain the relative similarity of the code paths taken. The obtained trace clusters represent (together with their occurrence indexes) effective representations of a DD's execution hotspots. From a testing perspective, this strongly indicates the possibility to significantly reduce the testing effort needed to cover the exercised code paths by thoroughly testing only a single representative code path from each equivalence class.

Additionally, we show how the number of equivalence classes can be decided by varying the similarity threshold (the cutoff factor of the dendrogram - a tree-like structure describing the clustering). This represents a powerful tool for directing the efforts that a subsequent testing campaign needs undergo.

Overall, this paper outlines a methodology to obtain execution profiles for kernel DDs, and ascertain its viability against a set of actual Windows DDs. By using tracing information from two driver communication interfaces, our technique provides insights that help understand a DD's runtime behavior in terms of execution paths. The main contributions of this paper are:

- A novel method to accurately profile a driver's runtime behavior in terms of the called kernel services.
- An occurrence-weighted list of kernel functions accessed by a driver indicating possible error propagation paths among kernel libraries and drivers.
- A novel application of clustering algorithms to identify and tune equivalence classes of test cases.
- A tendency of call traces to cluster is demonstrated in a real-world scenario, outlining execution hotspots for an actual DD (the floppy disk driver).

Paper Organization

The paper is organized as follows. Section 2 introduces the related work, followed in Sects. 3, 4 and 5 by the presentation of the terminology and main work concepts used throughout the paper. After a discussion on clustering aspects in Sect. 6, Sects. 7 and 8 present and then discuss the experimental validation of the method. Finally, Sect. 9 concludes and briefly presents our ongoing research activities.

2 Related Work

Weyuker recommends to focus testing of general SW onto the functionalities with high occurrence rates in the field [13, 14] in order to find faults with high likelihood to perturb service provision early on. Intuitively, such an option is enabled only under the assumption that a runtime profile of the program targeted by testing is available. Our methodology create such profiles for kernel DDs by revealing the taken code paths and the set of driver-external functionalities required at runtime. Defect localization studies for general [15] and OS-specific SW [6] support Weyuker's recommendation by showing that defects tend to cluster into certain areas of code. By profiling a DD's activity, the work presented in this paper guides a rigorous partitioning of the code by indicating runtime execution hotspots. Moreover, Weyuker warns about the necessity to validate COTS components in their new environments, even though they successfully passed their producers's testing campaigns. As our methodology is completely disconnected from the need to access any OS part's source code, it can be used for black-box level DD profiling, thus easing the testing efforts of a DD's user.

Johansson et al. proposed in [16] a selection method for SWIFI injection triggers which is based on call blocks of driver-external functions. The methodology presented in this paper for profile construction is similar in terms of the used monitoring strategy, but in contrast we consider the effects of the kernel's I/O requests on the DD's behavior. Mendonca and Neves [1] used a SWIFI technique to evaluate the robustness of the kernel libraries. The target functions were selected statically, by inspecting the import tables of the DDs (see 5.1) of several Windows installations and choosing the ones that are used by most of the DDs. In accordance with Weyuker's recommendations, our results suggest that the target functions should be selected on a dynamic basis (using profiling) by building occurrence indexes to guide the selection process.

Ball and Larus [17] acknowledged the application of path profiling for test coverage assessment, "*by profiling a program and reporting unexecuted statements or control flow*". They used binary instrumentation to obtain instruction traces that reveal a program's control-flow to identify paths and their execution frequencies. The paths ended at loop and procedure boundaries. An extension is represented by the "*whole program paths*" described in [18], which cross both boundaries to reveal a better picture of a program's execution pattern. Though, these approaches are not directly applicable to DD as the they are implemented as function libraries rather than programs in the classical sense. Moreover, instrumentation induces a high execution overhead and produces large amounts of data, two characteristics which penalize the use of this approach inside the OS kernel space.

Leon and Podgursky [19] used profiles generated by individual test cases and a clustering technique for evaluating test suite minimization by selecting one test case per cluster. The profiles used were generated by third-party tools, so the cluster analysis had to rely on their accuracy. While test cost reduction is out of the scope of this paper, we focus on building viable and accurate DD profiles, as a prerequisite mean to reducing test efforts.

3 System Model: The Entailed OS Kernel Components

In this paper we consider a model of a computing system as depicted in Fig. 1. It represents a computer equipped with Windows XP, the chosen OS for the case studies presented in Sect. 7. Nevertheless, the system in Fig. 1 is general enough to represent the architecture of most of contemporary COTS OSs. Here, the OS defines the layer between the hardware and user-mode applications. It provides to the applications an abstract view of the hardware peripherals and a set of services for accessing and managing them. In Fig. 1, the entities relevant for the approach are located inside the OS kernel space.

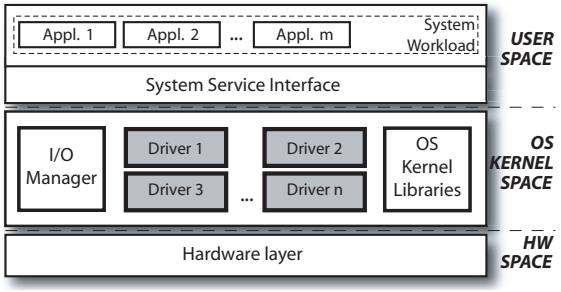

Fig. 1. A HW-SW system featuring a COTS OS with n DDs (*Windows XP*).

The *System Service Interface* provides a uniform service interface to the applications. That is, applications issue access requests to different services offered by the OS and this abstraction layer translates them into specific calls to various OS structures, hiding the diversity of the peripheral access interfaces from the applications.

The *device drivers* (DDs) can be considered device-specialized toolboxes to access each particular hardware peripheral. The DDs are loaded by the OS at initialization time or on demand, when it needs to communicate with a certain hardware device. In Windows XP, the structure of the DDs is specified by the Windows Driver Model (WDM) [20]. WDM defines the format of the kernel structures associated with DDs, the programming interface they need to follow and the communication paradigm with the I/O Manager, described below. To support the concepts described by the WDM, a Driver Developer Kit (DDK) containing tools and documentation is available for DD development.

The *I/O Manager* is a combination of various OS kernel structures with role in naming, registering and managing the DD objects. The I/O Manager is concerned with preparing and sending commands to the DDs, for OS administrative purposes or on behalf of the applications. Also, the I/O Manager prepares the results of the I/O invocations of the peripherals (received from the corresponding DDs) and forwards them to the calling applications. Section 4 contains a more detailed discussion on DD's interaction with the I/O Manager.

The *OS Kernel Libraries* are dynamic-linked libraries implementing general functionality and mechanisms that ensure core OS service provision (i.e., process and thread management, synchronization primitives, scheduling). In the Windows-family OSs the kernel libraries are built as portable executables, following the PE/Coff standard [21]. The DDs and other kernel components use the services offered by the kernel libraries by calling their exported functions. Section 5 discusses in detail how DDs use the functionalities stored in kernel libraries.

4 Developing the Basis for Code Tracing: The I/O Request Packet (IRP) Interface

I/O Request Packets (IRP) are kernel structures built by the I/O Manager when a request needs to be sent to a DD. The IRP structure contains the request type and the parameters needed by the recipient DD to start executing the request-associated activity. When a result of the operation is available, the DD uses the same IRP structure to piggyback it back to the I/O Manager.

Currently, WDM specifies 28 types of IRP requests (for instance `READ` for reading data from the device and `CLEANUP` for preparing the device for unload etc.). A DD must implement dispatch functions for every IRP type it supports and register its list of supported IRPs with the I/O Manager. This request type-based code separation of WDM-compliand DDs is relevant for our approach, as one can infer the functionality executed at any instant, based only on the type of the issued IRP.

4.1 The Processing of I/O Requests

To illustrate how an I/O request is processed, consider a simple example of an application that issues a `read` request to a hardware device. Figure 2 depicts this procedure where the main stages are: (1) the application calls `ReadFile` function of the WinAPI; (2) the WinAPI traps the OS kernel into the I/O Manager which selects the DD managing the target HW device; (3) the I/O Manager encodes the I/O request in an IRP structure and forwards it to the DD; (4) the DD contacts the HW device, instructs it to retrieve the data and completes the IRP; (5) the I/O Manager reads the completion information from the IRP and (6) returns the result to the WinAPI, in terms of a pointer to data; (7) the WinAPI copies the data to a buffer accessible to the calling application (in user space) and (8) informs the application about the result of the operation and the location of the requested data. In this paper, we capture the IRP flow between the I/O Manager and a DD. At this communication interface, two relevant events are recorded: *incoming IRPs* (from I/O Manager to DD, step (3)) and *outgoing IRPs* (from DD to the I/O Manager, step (5)). Onwards, we call this communication level the "*IRP interface*".

According to the WDM specification, each HW device has at least two drivers servicing it, the *function driver* and *the bus driver*. The function driver is handling the receipt and completion of IRPs, whereas the bus driver is responsible

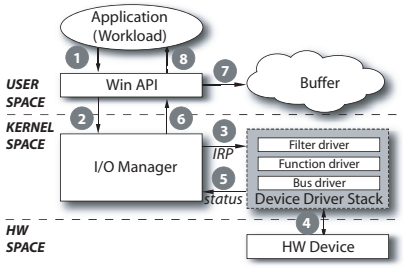

Fig. 2. Processing of I/O requests.

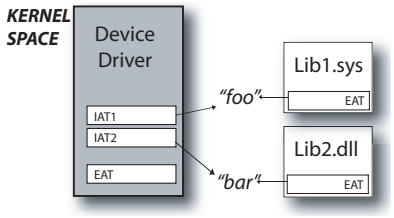

Fig. 3. A DD importing functions from two libraries.

for connecting the device with the host computer, being usually the driver located closest to the HW peripheral. On top of the function driver, a driver stack can contain one or several *filter drivers*, which act as wrappers for the underlying drivers. Filter drivers are usually responsible for the preparation of the the IRP structure or initialization of other kernel structures needed by the drivers located below in the stack.

In order to capture the IRP flow we have built a filter driver and installed it on the top of the driver stack. Its location enables it to be the first to receive the incoming IRPs and the last to see the outgoing ones. Our filter driver logs the incoming and outgoing IRPs and forwards them, unchanged, to their original recipients while keeping the induced computational overhead to minimum. As each incoming IRP triggers the execution of a dispatch function and each outgoing IRP signals the termination of the associated computation, we use the captured IRP flow logs to specify a DD's activity at any certain instant. The filter driver is written with portability in mind, so that it can be installed on top of any WDM-compliant DD, without requiring modifications.

4.2 Mode, Transition and Operational Profile of a Device Driver

As we do not assume access to the DD's source code, we consider that DD state changes are caused by the I/O request flow having the DD as recipient. In our previous work [11] we introduced an abstracted DD state definition called *mode* that allows for expressing DD activity at runtime. The *mode of a driver* is defined as follows:

Definition 1 (Driver Mode). *The mode of a driver D is the tuple of binary predicates, each assigned to one of the n distinct IRP types supported by the driver:*

$$M^D : \ < P_{IRP_1} \ P_{IRP_2} \ P_{IRP_3} \ .. \ P_{IRP_n} >, \quad where \, P_{IRP_i} \ is$$

$$P_{IRP_i} = \begin{cases} \mathbf{1}, & if \ \textbf{performing} \ the \ functionality \ triggered \ by \ receival \ of \ IRP_i \\ \mathbf{0}, & otherwise \end{cases}$$

A *transition* between modes is triggered by the receival of a new IRP or completing an executed IRP. As the I/O Manager serializes the IRP flow, our model assumes that only one bit of the tuple describing the mode of the DD is flipped at a time. Therefore, the DD can switch only to modes whose binary tuples are within Hamming distance of 1 from the current mode. Because of this behavior, the number of possible transitions in the model is $n \cdot 2^n$ (each mode can be left on n exit transitions).

We call the set of the modes (N_{op}) visited under a certain workload relevant for the DD, together with the traversed transitions (T_{op}) the *operational profile* of the DD. In [11] we have demonstrated that irrespective of the chosen workload, the operational profile is a small subset of the total, theoretically-possible state space of the DD ($N_{op} \ll 2^n$ and $T_{op} \ll n \cdot 2^n$).

5 Developing the Basis for Code Tracing: The Functional Interface

The communication between OS kernel and DDs is not limited to the IRP scheme. A DD also communicates with the OS kernel using a second interface, which we onwards call the "*functional interface*". Enabled by the concept of dynamic linking, at this communication level the parties involved are kernel libraries and DDs, as image files. In fact, this scheme forms the basis of OS modularity, and is the most commonly used data communication paradigm between binaries. The OS provides a set of kernel libraries containing functions required by the different kernel components. Each library publishes a list of the available functions. On the other side, the DDs (as consumers of the services provided by the libraries) contain a list of necessary libraries and for each of them a list of the used functions from the respective library. For both kernel libraries and DDs the lists mentioned above are stored in the headers of the binary files.

5.1 The PE/COFF Executable Format and DLL-Proxying

In Windows, the PE/COFF format [21] specifies the file headers that permit a Windows executable file to publish the contained functions and variables (*exports*) and to use functions defined externally by another library (*imports*). The example in Fig. 3 depicts a DD that imports functions implemented in two external libraries, `Lib1.sys` and `Lib2.dll`. Each contains an Export Address Table (EAT) that publishes a list of functions exported by the respective library. At runtime, the DD links to the kernel libraries on demand, when the result of the functions `foo` and, respectively, `bar` are needed. Therefore, the header of the DD file contains an Import Address Table (IAT) for each of the needed libraries. The IAT contains only the function names which are used in the DD's code.

At DD load time, the OS automatically checks if all the required libraries are present in the system by inspecting the DD's IATs. If they cannot be found, an error message is issued and the DD loading is aborted. At load time, no verification is done to check if the libraries found actually contain the necessary

functions for the DD to execute correctly. Only at runtime, when a portion of DD code containing calls to external functions is reached, the DD accesses the associated library to utilize its services.

The work presented in this paper relies on the ability to capture the calls to external functions at DD runtime. While various methods for capturing calls to externally located functions exist (eg., Detours [22], Spike [23]), they are specific to user-space software and are therefore not directly applicable to kernel-mode programs. In contrast, we need a kernel space mechanism to monitor the function calls. Therefore, we have chosen to implement a *DLL-proxying* technique. Briefly, DLL-proxying consists of building a DLL library which act as a wrapper of the original library. In order to leave the functionality of the DD unaffected, the wrapper library has to implement all the functions required by the DD, or to forward its calls to the original library. By modifying the IAT tables of the target DD to point to the wrapper library instead of the original one, the wrapper library (also called *DLL-proxy*) is interposed between the two parties. Section 7 details our implementation of DLL-proxies inside the Windows kernel.

Our kernel-mode library wrappers are used exclusively for capturing the sequences of functions called by a DD at runtime, when exercised by a selected workload. Consequently, we only need to log the function names but not modify any parameters or behavior of the wrapped kernel APIs. Therefore, the overhead induced by the DLL-proxy is kept to minimum.

5.2 Call Strings as Code Path Abstractions

As external function calls correspond to DD code being executed as a result of IRP requests (or other OS kernel maintenance requests), grouping them using IRPs as boundaries is intuitive. Therefore, we introduce the notion of *call string* as follows:

Definition 2 (Call String). *A call string (CS) is a sequence of DD-external function calls issued at runtime by a DD, delimited by incoming and outgoing IRP requests.*

In this paper we consider each CS an abstraction representing the code path taken by the DD at execution time. As we use the incoming and outgoing IRP requests as CS delimiters, each CS can be associated with a DD mode and, subsequently, with an IRP dispatch function.

Illustrating the CS capturing method, the left part of Fig. 4 shows an abstract representation of the WDM-compliant DD's code with dispatch functions for handling `READ` and `WRITE` requests. Assuming that the DD can handle only those two IRP requests, the visited modes are defined by bit strings with length two; the first bit is associated with `READ` and the second with `WRITE` operation. Note that both dispatch subroutines call functions implemented externally by other kernel libraries. Assuming that at a certain instant the DD receives the `READ` request followed by an `WRITE` request, the log file that combines the events recorded by monitoring the two communication interfaces is depicted on the

right side of Fig. 4. Hence, the call strings CS_i and CS_{i+1} can be constructed and associated with the modes $< 10 >$ and, respectively, $< 01 >$.

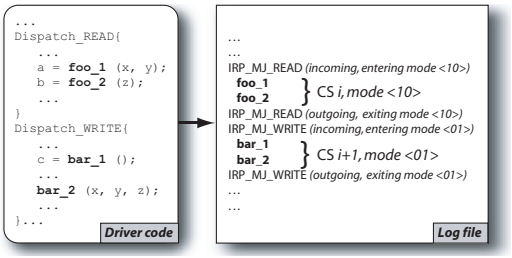

Fig. 4. The code path taken in a DD when `READ` and `WRITE` requests are called.

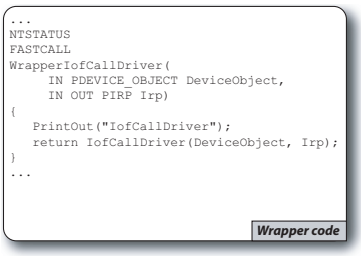

Fig. 5. A wrapper for the `NTOSKRNL::IofCallDriver` API.

Consequently, the call CSs can be studied from two perspectives: (a) *per mode basis*, i.e., CSs belonging to the same DD mode are compared to reveal possible differences in the code paths taken each time the DD performs the activity associated with the respective mode, and (b) *per CS basis*, i.e., all CSs are compared among themselves to identify similarities and to group them accordingly in equivalence classes. Hence, we define the term *execution hotspot* as follows:

Definition 3 (Execution Hotspot). *A group of similar CSs belonging to the same equivalence class represents an* execution hotspot*. The magnitude of each hotspot is given by the sum of occurrences of the CSs contained within the equivalence class.*

The methodology for building kernel DD profiles presented in this paper reveals the execution hotspots together with their magnitudes. The construction of the equivalence classes is achieved by employing a cluster analysis algorithm, as described in the following section.

6 Identifying Execution Hotspots: Call String Clustering Aspects

Given the size of the pool of data collected in the monitoring phase, a data clustering method greatly facilitates organizing and interpreting the data trends. *Cluster analysis* is a multivariate technique that helps partitioning a population of objects into equivalence classes. The partitioning decision is taken on object similarity, i.e., similar objects are grouped together in the same *cluster*. The most common clustering approaches are hierarchical and partitional. Usually slower than hierarchical algorithms, the *partitional clustering* initially divides object population in k clusters (randomly), improving the clusters at each step by

redistributing the objects. *Hierarchical clustering* approaches fall in two classes, agglomerative and divisive.

Agglomerative clustering (also called bottom-up clustering) initially assigns each object into its own cluster, at each step similar clusters are merged. The agglomerative clustering algorithms stop when all objects are placed in a single cluster, or when a number of k clusters (given as a parameter to the algorithm) remain. *Divisive* clustering (top-down clustering) algorithms initially assign all the objects from a given population to a single cluster, divided at every step in two non-empty clusters. A divisive clustering algorithm stops when each object sits in an own cluster or when a number of k clusters is reached.

In this paper we use automated agglomerative analysis to divide the CS population into similar clusters. We use `AgNes`, an agglomerative algorithm provided by the R statistical programming environment [24]. `AgNes` requires as input a matrix containing the distances between every pair of objects, in our case CSs. It outputs a dendrogram, which is a tree-like representation of the clustering. The Figs. 10 and 11 represent examples of such dendrograms. The CSs are represented as leaves, and branches intersect at a height equal to the dissimilarity among the children. Cutting the dendrogram at a given height reveals the clusters and the contained call sequences at the respective distance. That is, a *cutoff* of the dendrogram indicates the equivalence classes that partition the CS population for the respective distance. For a cutoff set at 0, the equivalence classes contain only the CSs which are identical. Therefore, the cutoff value acts as a tunable mask for CS diversity.

6.1 Metrics to Express Call String Similarity

To obtain relevant dendrograms of the CS clusters, an appropriate similarity metric has to be first selected. In the areas of bio-informatics and record linkage (duplicate detection) researchers have developed a series of metrics to quantify the relation between two strings. Depending on their application area, some metrics express the similarity while other measure the difference (dissimilarity) of the compared strings.

The *Levenshtein* distance (d_L) is based on the edit distance between the compared strings. Given two strings s_1 and s_2 whose distance needs to be computed, Levenshtein distance express the minimum number of operations needed to transform s_1 in s_2 or viceversa. The considered operations are character insert, delete or substitution and they all have the cost of 1. Used in bio-informatics to decide global or local alignments for protein sequences, *Needleman-Wunsch* and *Smith-Waterman* distances are versions of the Levenshtein metric, additionally considering gap penalties (gap = subsequence that do not match).

Jaro distance is not based on the edit distance, but instead on the number and order of the common characters. The Jaro distance is expressed by the following formula:

$$d_J = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) \tag{1}$$

where m is the number of matching characters and t is the number of necessary transpositions. Two characters are considered matching if they are not farther than $\left\lfloor \frac{max(|s_1|,|s_2|)}{2} \right\rfloor - 1$ from each other. An extension of the Jaro distance was proposed by Winkler, in order to reward with higher scores the strings that match from the beginning (they share a common prefix).

Therefore, the *Jaro-Winkler* distance is defined by the formula

$$d_{JW} = d_J + [0.1 \cdot l(1 - d_J)] \tag{2}$$

where l is the length of the common prefix and d_J is the Jaro distance between the strings.

Many other distance metrics exists and were evaluated for various applications [25]. We have also investigated several of them and subsequently chosen the Levenshtein and Jaro-Winkler metrics, as we believe they express best the distance among the CSs. Levenshtein was selected as it captures neutrally the variability of the CSs. As we expect the CSs to contain short, repetitive subsequences (generated by loops in the code path) and common sequences (generated by shared helper functions), we have also selected the Jaro-Winkler metric as it favors similarities between CSs showing this behavior.

To balance their effects and to minimize the impact of the metric choice on the final cluster structures, we combined them in a compound measure, a simple weighted average:

$$d_C = \frac{norm(d_L) + norm(d_{JW})}{2} \tag{3}$$

Our compound metric uses normalized values for both Levenshtein and Jaro-Winkler functions, therefore $0 \leq d_C \leq 1$. Being a dissimilarity function, small values of d_C indicate high similarity between the compared CSs. The distance matrix required by `AgNes` was computed using d_C for expressing the distance among every CS pairs.

6.2 Cluster Linkage Methods and Agglomeration Coefficient

Besides the distance matrix, `AgNes` requires that a clustering method is specified. *Simple linkage* merges at every step two clusters whose merger has the smallest diameter. This method has as disadvantage a tendency to form long cluster chains (i.e., at every step a single element is added to an existing cluster). *Complete linkage* merges clusters whose closest member objects have the smallest distance. This linkage method creates tighter clusters but is sensitive to outliers. To alleviate the disadvantages of simple and complete clusterings, *average linkage* groups clusters whose average distance between all pairs of objects is minimal.

`AgNes` provides a standard measure to express the strength of the clustering found in the population of CSs. A strong clustering tendency means larger inter-cluster dissimilarities and lower intra-cluster dissimilarities. If $d(i)$ is the

dissimilarity of object i to the first cluster it is merged with divided by the dissimilarity of the last merger, the *agglomeration coefficient (AC)* is expressed by `AgNes` as the average of all $1 - d(i)$. With $0 \leq AC \leq 1$, larger AC values indicate a good cluster structure of the object population.

For our clustering analysis experiments presented in Sect. 7 we have used the average linkage method as we believe this choice factors out best the impact of CS distance variance among the object population.

7 Evaluating the Viability of the Execution Profiling Methodology

For a comprehensive evaluation of the dual-interface DD profiling method presented in this paper, we have used it against the `flpydisk.sys` (v5.1.2600.2180), the floppy disk driver provided by Windows XP SP2.

Figure 6 depicts our experimental setup. To capture the requests occurring on the IRP interface of the target DD we have built a filter driver and installed it between the monitored DD and the I/O Manager. The filter driver receives the incoming and outgoing IRP requests, logs them to a file and forwards them to the original recipient. As the filter driver does not rely on the implementation details of the underlying DD, it can be used to monitor virtually any WDM-compliant DD, as shown in practice by the experimental work in [12].

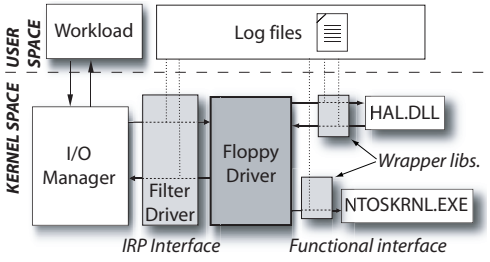

Fig. 6. Our DD monitoring strategy.

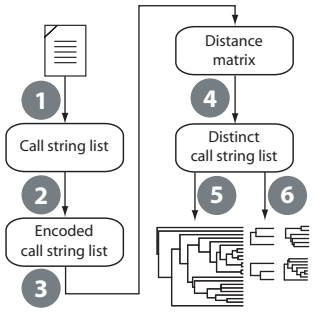

Fig. 7. Our cluster analysis process.

The monitoring of the functional interface is more complex, as it requires building a wrapper library for each of the kernel libraries imported by the floppy driver (Fig. 6). `flpydisk.sys` imports functions from two kernel libraries: `NTOSKRNL.EXE` (61 functions) and `HAL.DLL` (4 functions). After building the library wrappers, the IAT tables of the target DDs were modified in order to look for the wrappers instead of the original libraries. Each API wrapper was built using exclusively the function prototypes provided in the header files available publicly from Windows DDK package. Each time the DD called a function, the

API wrapper is called instead of the original function. The API wrappers are designed as extremely simple C constructs in order to minimize the computational overhead. When a wrapper is called, the call is logged and the call parameters are forwarded, unchanged, to the original function from the original library, as depicted by the code snippet in Fig. 5. In this figure, `IofCallDriver` is the original function implemented by `NTOSKRNL.EXE` and `WrapperIofCallDriver` is our wrapper.

After the floppy driver is exercised by a relevant workload, the resulted log files are analyzed offline by a software application that extracts the CSs and constructs distance matrix files. These files are fed to the `AgNes` algorithm which builds clusterings of the CSs. More precisely, the procedure followed to build the clusterings that evaluate the CS relative similarity is depicted in Fig. 7: (1) collect the CSs by using the monitoring logs; (2) encode each function call to an Unicode character to be able to apply the string metrics; (3) calculate a distance matrix containing the distances between all pairs of CSs; (4) select the distinct CSs and count for each one the occurrence rate; (5) construct a clustering from all distinct CSs to evaluate inter-CS similarities; (6) for each mode, construct a clustering of CSs to reveal intra-mode paths.

Table 1. The workloads utilized to exercise the floppy driver and the overall experimental outcomes.

Benchmarks for flpydisk.sys	**#Called Imports**			**#Modes**	**#CSs**		**AC**	**Benchmark Description**
	Total	**NT**[1]	**HAL**[2]		**Total**	**Distinct**		
`Sandra`	27	25	2	3	9545	51	.859	Performance benchmark
`DiskTestPro`	28	26	2	5	588	13	.735	Surface scan, format
`BurnInTest`	21	19	2	5	1438	24	.823	Reliability benchmark
`Enable_Disable`	42	38	4	3	136	10	.388	DD load and unload
`DC2`	21	19	2	4	5102	9	.644	Robustness benchmark

To exercise the DD properly, we have used commercial performance and stability benchmark applications which are designed for testing the floppy disk drive. We have also used a robustness testing tool, `DC2` (Device Path Exerciser). `DC2` is part of the DDK package and evaluates if a DD submitted for certification with Windows is reliable enough for mass distribution. It sends the targeted DD a variety of valid and invalid (not supported, malformed etc.) I/O requests to reveal implementation vulnerabilities. The Table 1 lists the outcomes and provides a comparative evaluation of the clustering strength (see Sect. 6.2).

`Sandra` was the workload that issued the highest number of distinct CSs (51 out of 9545), showing the highest cluster strength in the distinct CS population, with $AC = 0.859$. Also, the DD visited only three modes, intuitively indicating that this workload might have the strongest tendency to reveal execution hotspots. At the other extreme, `Enable_Disable` only revealed 10 distinct CSs (out of 136), but instead the calls to the external functions were the most diverse,

[1] The number of functions called from `NTOSKRNL.EXE`.
[2] The number of functions called from `HAL.DLL`.

38 from NTOSKRNL.EXE and 4 from HAL.DLL. As the agglomerative coefficient of this workload is relatively small, we expect that Enable_Disable has the weakest clustering tendency.

7.1 Revealing the Execution Hotspots: MDS Plots of the CSs

To visualize the clustering tendency of the CSs generated by the used workloads and, implicitly, the execution hotspots in floppy driver's code, we used a multidimensional scaling (MDS) plot. MDS is a statistical technique designed to graphically express the degree of similarity or dissimilarity between objects. The points representing similar objects are clustered together in different regions of the 2D-space depicted by the MDS plot, while the points representing dissimilar objects are placed to be far apart from each other. The MDS plot in Fig. 8 is computed using the already available distance matrices.

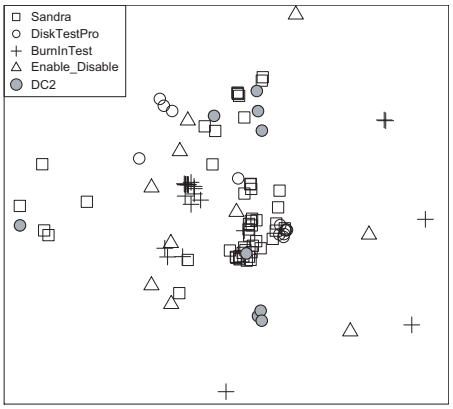

Fig. 8. MDS plot of the CSs for each workload.

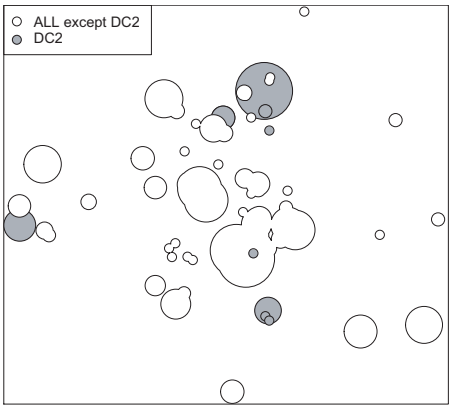

Fig. 9. MDS plot of the execution hotspots with their magnitudes.

With a high AC, Sandra forms the biggest clusters mostly in the center of the figure, while the areas exercised by the Enable_Disable are located farther apart from each other. This visual representation of the CSs also helped reveal another tight cluster close to the center of the Fig. 8, generated by the BurnInTest workload. Also, DiskTestPro's executions form a hotspot, located in the second quadrant of Fig. 8. Overall, the grouping of the CSs in the middle of the MDS plot indicates that most of them share a certain degree of similarity.

Interestingly, the CSs generated by DC2 are located quite differently from the rest of other CSs. This is explained by the fact that DC2 is a robustness testing tool, therefore accessing areas of code seldomly visited under common executions. To better substantiate this tendency, Fig. 9 represents the same MDS plot, where each CS was enhanced with the magnitude of the associated CS. That

is, a bigger circle represents a high rate of occurrence of the respective CS. The circles are scaled using a logarithmic function ($size = \log(magnitude_{CS})$) in order to create a visual balance between CSs having very different occurrence rates. Additionally, the execution hotspots generated by the first four workloads from Table 1 were merged, while the hotspots generated by the robustness testing tool `DC2` were represented in gray. As `DC2`'s hotspots are off-centered, it becomes apparent that the `DC2` covers very few of the execution hotspots generated by all other studied workloads.

Nevertheless, the Figs. 8 and 9 validate our methodology and graphically motivate the usage of execution profiles as a prerequisite step for testing. We believe that a significant amount of testing can be saved by redistributing the effort to covering the execution hotspots. Doing so significantly reduces the test effort, while the test adequacy remains unaffected. While test case filtering is not the scope of this paper, we hypothesize that an iterative method based on comparisons of test suites against an existing execution hotspot map can be devised in order to guide this process.

7.2 Similarity Cutoffs: Testing Overhead versus Diversity Masking

The dendrograms obtained at steps 5 and 6 in Fig. 7 represent useful support for deciding which code paths to test. To ensure high accuracy for the subsequent testing campaigns with respect to the execution hotspots, one should develop test cases that exercise the DD in the same manner as the workload does, or, alternatively, use the test cases themselves as workload for exercising the DD in the profiling phase.

We believe that the testing effort can be significantly reduced by testing only the distinct CSs. A prioritization scheme for this procedure should consider (and therefore be indexed by) the number of occurrences associated with each CS ($magnitude_{CS}$). Intuitively, a subsequent test campaign can reduce its overhead by testing only one CS per cluster. Figure 10 illustrates this concept: by setting a similarity cutoff $T = 0.2$, the dendrogram is split into four clusters and five alones (CS0, 1, 15, 22 and 23). This indicates nine code paths that must be tested: the alones and any one CS from each of the four clusters, since all the CSs that are contained in the cluster are considered similar. With $T = 0$, 24 CSs should be tested in order to achieve complete hotspot coverage. Therefore, setting the $T = 0.2$ gives an overall reduction of 62.5% of the testing cost (assuming that the cost of testing is equally distributed among the 24 distinct CSs). In practice, the similarity cutoff T has to be chosen as close to zero as possible, because large values of T have a tendency to mask CS diversity. Actually, dendrograms support the similarity threshold decision by their structure. If the CSs cluster at very low heights, a small cutoff value will group many CSs together, significantly reducing the test efforts without having to pay a high cost to diversity masking.

In contrast, Fig. 11 depicts the dendrograms of the CSs for each mode. In this representation it is apparent that in the visited modes the DD was taking at least three different paths into the code. The heights at which they cluster indicate that the CSs are quite dissimilar, even though they are basically associated with

the same DD functionality. This reveals that the IRP dispatch subroutines are quite complex, possibly containing multiple decision branches in the code. In the case of the per-mode dendrograms (Fig. 11), a similarity cutoff T smaller than the shortest cluster will reveal all the code paths taken inside the mode. Though, to balance the testing efforts, T should be chosen anywhere between the height of the smallest cluster and 1. With $T = 1$ the granularity of testing is the same as in our previous approach [11].

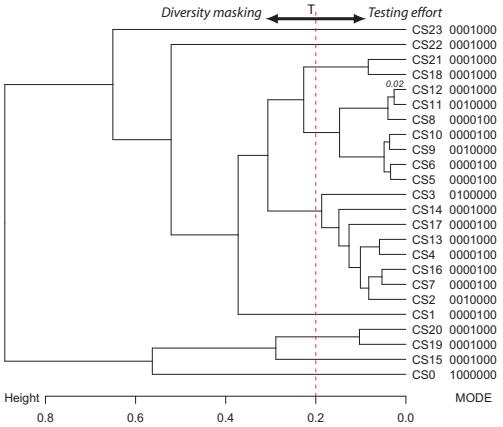

Fig. 10. `BurnInTest`: A threshold set to 0.2 reveals a clustering with 4 clusters and 5 alones (62.5% test cost reduction).

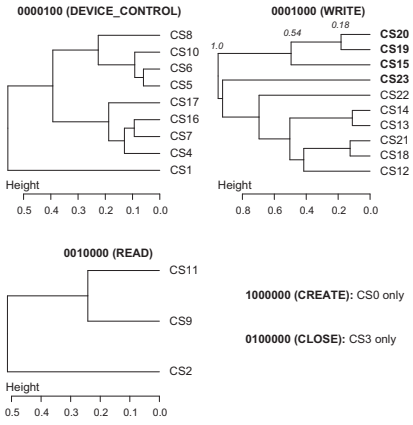

Fig. 11. `BurnInTest`: The distinct CSs called by every mode.

This represents one of the key contributions of this paper in contrast to our previous work, where the smallest DD behavior granularity unit was the notion of *mode*. Using the dual-interface approach presented in this paper, a subsequent testing technique can take advantage of the smaller granularity offered by the new concept of *CS*.

8 Discussion and Results Interpretation

Identification of Repeating Functions: Table 3 displays five distinct CSs, as generated by the `BurnInTest` workload. The respective CSs are highlighted also in Fig. 11. CS15 is formed by a call `IofCompleteRequest` function, followed by `ExInterlockedRemoveHeadList` and `KeWaitForSingleObject`, repeating twice. The distance from CS15 to CS19 is 0.54 and to CS23 is 1.0; the distance from CS19 to CS20 is 0.18 (also depicted in Fig. 11). The low similarity values shared by the CS15, CS19 and CS20 are mainly given by the fact that the sequences share a common prefix and the group of two functions that repeat themselves. These repetitions indicate the presence of short loops in the DD's code. In particular, according to the DDK documentation, `ExInterlockedRemoveHeadList`

routine *"removes an entry from the head of a doubly linked list"* and `KeWaitForSingleObject` *"puts the current thread into a wait state"*. CS23 is heavily penalized when related to CS15 because the position of the only common character is not the same in the two CSs. In contrast, the distance from CS23 to CS0 is (only) 0.83 because both CSs are very short.

Table 2. Five functions and their encodings (used in Table 3).

Function Name	Encoding Char
`IofCompleteRequest`	a
`ExInterlockedRemoveHeadList`	b
`KeWaitForSingleObject`	c
`ExAcquireFastmutex`	d
`ExReleaseFastMutex`	e

Table 3. Four distinct CSs issued by the `BurnInTest`.

CS Name	Encoding	#Occurences
CS0	a	144
CS15	abcbc	2
CS19	abcbcbcbcbc	2
CS20	abcbcbcbc	1
CS23	dea	13

Figure 10 show cases when two CSs are very similar, even though they belong to different modes (i.e., CS11 and CS12, at a distance of 0.02). We believe that they share the same or large portions of a dispatch function. It is also possible that they share a large amount of helper functions, inside the DD's code. We are currently investigating in more depth the reasons behind this observed behavior on publicly available driver source code (the serial port driver).

Frequently Used Kernel Services: Our profiling approach reveals that the set of functions frequently used by a DD at runtime is very small. Table 4 lists the 20 function calls that make 99.97% of all the imports called by the `flpydisk.sys` at runtime in our experiments. In [1] Mendonca and Neves have chosen a set of 20 DDK functions for fault injection experiments by inspecting the IAT tables of all the DDs belonging to several Windows installations. Our results show that their static approach to select kernel APIs is irrelevant in such dynamic environments, as the set of functions called at runtime is radically different. Therefore, we recommend that subsequent fault injection campaigns should primarily target functions having higher runtime occurrence index.

Table 4. The function calls accounting for 99.97% of all recorded calls, for all workloads, sorted descending on occurrence. `ExAcquireFastMutex` and `ExReleaseFastMutex` belong to `HAL.DLL`, the rest to `NTOSKRNL.EXE` library.

Function Name	#Occ.	[%]	Function Name	#Occ.	[%]
`ExAcquireFastMutex`	60414	18.40	`MmMapLockedPagesSpecifyCache`	8178	2.49
`ExReleaseFastMutex`	60414	18.40	`MmPageEntireDriver`	24	0.01
`IofCallDriver`	45976	14.00	`MmResetDriverPaging`	23	0.01
`KeInitializeEvent`	40777	12.42	`KeGetCurrentThread`	10	0.00
`IoBuildDeviceIoControlRequest`	40771	12.41	`KeSetPriortyThread`	10	0.00
`ExInterlockedRemoveHeadList`	22007	6.70	`ObfDereferenceObject`	10	0.00
`ExInterlockedInsertTailList`	16123	4.91	`ObReferenceObjectByHandle`	10	0.00
`IofCompleteRequest`	11562	3.52	`PsCreateSystemThread`	10	0.00
`KeWaitForSingleObject`	11032	3.36	`PsTerminateSystemThread`	10	0.00
`KeReleaseSemaphore`	11003	3.35	`ZwClose`	10	0.00

9 Conclusions and Future Research Directions

In this paper we have presented a driver profiling technique that monitors the activity of a kernel driver at runtime onto two communication interfaces. Our technique disconnects execution profiling from the source code access requirement, for every of the involved OS kernel components. We consider that the driver is receiving requests on the *IRP interface* and start executing the IRP-associated activity. We revealed the effect of this computation as a sequence of calls to external functions, by monitoring the driver's *functional interface*. The CSs obtained were encoded as character strings and cross-compared for similarity. The distinct CSs were found to represent a very small number of the total number of CSs recorded during our experiments, indicating that the number of code paths taken by a driver at runtime is very small. Moreover, we employed an agglomerative cluster analysis technique in order to group together similar CSs and therefore suggest areas of code where the test effort of subsequent testing campaigns should concentrate. Using the same technique, the CSs belonging to the same mode were investigated and showed that the code paths taken by the driver differs even when executing the same IRP dispatch subroutine, a tendency that reveal code branches. Moreover, the MDS plots visually disclose the tendency of the CSs to cluster by revealing the execution hotspots. At the same time, the Figs. 8 and 9 show that `DC2`, a robustness testing tool for drivers from Microsoft, does not cover the execution hotspots generated by the other realistic workloads. This result intuitively supports the idea to re-balance the testing effort to the revealed execution hotspots, thus enhancing the odds to find early the faults having a high occurrence likelihood in the field.

Current research directions include the design and implementation of a fault injection method for testing the robustness of OS kernel drivers, based on the concepts introduced in this paper. The selection of test cases will consider the execution hotspots generated by a prior driver execution profiling phase, in order to reduce overall testing overhead. Test prioritization schemes will also be employed by applying the techniques described in our previous work [11, 12]. We also intend to investigate the possibility to implement state-aware robustness wrappers for kernel drivers, once we will establish a method for detecting deviations from "correct behavior".

References

1. Mendonca, M., Neves, N.: Robustness testing of the Windows DDK. In: Dependable Systems and Networks (DSN). (June 2007) 554–564
2. Swift, M.M., Bershad, B.N., Levy, H.M.: Improving the reliability of commodity operating systems. ACM Transactions on Computer Systems **23**(1) (2005) 77–110
3. Ganapathi, A., Ganapathi, V., Patterson, D.: Windows XP kernel crash analysis. In: Large Installation System Administration Conference (LISA). (2006) 12–22
4. Albinet, A., Arlat, J., Fabre, J.C.: Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In: Dependable Systems and Networks (DSN). (2004) 867–876

5. Arlat, J., Fabre, J.C., Rodriguez, M.: Dependability of COTS microkernel-based systems. IEEE Transactions on Computers **volume 51, issue 2** (2002) 138–163
6. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.R.: An empirical study of operating system errors. In: Symposium on Operating Systems Principles (SOSP). (2001) 73–88
7. Duraes, J., Madeira, H.: Multidimensional characterization of the impact of faulty drivers on the operating systems behavior. IEICE Transactions on Information and Systems **86**(12) (2003) 2563–2570
8. Murphy, B., Garzia, M., Suri, N.: Closing the gap in failure analysis. In: Dependable Systems and Networks (DSN). (2006) 59–61
9. 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
10. Johansson, A., Suri, N.: Error propagation profiling of operating systems. In: International Conference on Dependable Systems and Networks (DSN). (2005) 86–95
11. Sârbu, C., Johansson, A., Fraikin, F., Suri, N.: Improving robustness testing of COTS OS extensions. International Service Availability Symposium (ISAS) **Springer Lecture Notes on Computer Science 4328** (2006) 120–139
12. Sârbu, C., Suri, N.: Runtime behavior-based profiling of OS drivers. Technical report, TR-TUD-DEEDS-05-02-2007 (2007), `http://www.deeds.informatik.tu-darmstadt.de/research/TR/TR-TUD-DEEDS-05-02-2007-Sarbu.pdf`
13. Weyuker, E.J., Jeng, B.: Analyzing partition testing strategies. IEEE Transactions on Software Engineering **17, Issue: 7** (July 1991) 703–711
14. Weyuker, E.J.: Using operational distributions to judge testing progress. In: ACM Symposium on Applied Computing, New York, ACM Press (2003) 1118–1122
15. Möller, K.H., Paulish, D.: An empirical investigation of software fault distribution. In: First International Software Metrics Symposium (METRIC). (May 1993) 82–90
16. Johansson, A., Suri, N., Murphy, B.: On the impact of injection triggers for os robustness evaluation. In: International Symposium on Software Reliability Engineering (ISSTA). (2007) 127–136
17. Ball, T., Larus, J.R.: Efficient path profiling. In: MICRO-29. (1996) 46–57
18. Larus, J.R.: Whole program paths. In: ACM SIGPLAN, 34. (1999) 259–269
19. Leon, D., Podgurski, A.: A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In: 14th International Symposium on Software Reliability Engineering (ISSRE). (2003) 442–453
20. Oney, W.: Programming the MS Windows Driver Model. Microsoft Press, Redmond (2003)
21. Microsoft Corporation, Visual Studio, Microsoft portable executable and common object file format specification. Technical report, (May 2006), `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`
22. Hunt, G., Brubacher, D.: Detours: Binary interception of Win32 functions. In: Proceedings of the 3rd USENIX Windows NT Symposium. (July 1999) 135–144
23. Vasudevan, A., Yerraballi, R.: Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation. In: Australasian Computer Science Conference (ACSC). (2006) 311–320
24. Ihaka, R., Gentleman, R.: R: A language for data analysis and graphics. Journal of Computational and Graphical Statistics **5**(3) (1996) 299–314
25. Cohen, W.W., Pradeep, R., E., F.S.: A comparison of string distance metrics for name-matching tasks. In: International Joint Conference on Artificial Intelligence (IJCAI). (2003) 73–78