On Enhancing the Robustness of Commercial Operating Systems[⋆]

Andréas Johansson, Adina Sârbu, Arshad Jhumka and Neeraj Suri

Department of Computer Science
Technische Universität Darmstadt, Germany
{aja,adina,arshad,suri}@informatik.tu-darmstadt.de

Abstract. A ubiquitous computing system derives its operations from the collective interactions of its constituent components. Consequently, a *robust* ubiquitous system entails that the discrete components must be robust to handle errors arising in themselves and over interactions with other system components. This paper conceptually outlines a profiling framework that assists in finding weaknesses in one of the fundamental building blocks of most computer based systems, namely the Operating System (OS). The proposed framework allows a system designer to ascertain possible error propagation paths, from drivers through the OS to the applications. This knowledge significantly helps enhance the OS (or driver/application) with selective robustness hardening capabilities, i.e., robustness wrappers.

1 Introductions: The Ubiquitous Computing Perspective

A ubiquitous computing (UC) environment harnesses the collective capabilities of diverse computational components via dynamic resource management as warranted in mobile, networked and heterogeneous system environments. The utility of such UC systems arises only if adequate robustness in the UC infrastructure exists for it to provide for dependable service delivery. It is evident that achieving an acceptable level of trust in such consolidated systems also necessitates corresponding design methods for evaluating (and forecasting) how perturbations in the system affect the services provided. Such perturbations could arise as device defects, in UC component interactions, bugs in software etc. For UC components themselves, the aspects of heterogeneity and mobility translate to the problem of not knowing at design time the precise characterization of their operational environment(s). The problem is further complicated that many of the devices are too deeply embedded to be modifiable. However, during deployment, (robustness) profiling of the system is viable. This will give rise to suggested locations of robustness gaps (hence, of enhancements) within the system. However, owing to the dynamic nature of UC, profiling the overall system is not viable. Thus, we propose to perform a two-level profiling: (i) profiling the platform, including components that will be long-term present in the system (middleware, OS,

[⋆] We appreciate the inspration & insights from Dr Martin Hiller and the funding support of Microsoft Research through the Innovation Excellence Program.

drivers, HW etc) (ii) profiling the applications, which continually change over the operation of the system. The problems with equipping the system with added capabilities during run-time or deployment include (a) not knowing which errors to protect against, (b) not knowing where to add enhancements in the system, and (c) how to design effective and efficient wrappers.

1.1 The Role of OS: Paper Objectives

Targeting the OS as a key building block in any UC system, OS robustness hardening is a fundamental driver for providing robustness of the systems built upon them. The OS manages the hardware resources available and acts as a supplier of services for applications through programming interfaces. A potential problem for any OS is that it must be able to handle diverse hardware resources and applications. The risk of robustness gaps in the OS is apparent and the need for robustness enhancements exists. Naturally, adding robustness enhancements comes as a cost trade-off with other system properties, e.g., performance, determinism etc. A trade-off analysis needs span the need for enhancements (wrappers) and their consequent properties (coverage, timing etc).

As the overall goal is to prevent application failure, the interactions across application and OS are key. Owing to the dynamic nature of UC, the information needs to be categorized into two parts: (i) an OS profile, and (ii) an application profile. These profiles will aid in identifying vulnerabilities in the system, thereby guiding the effective placement of wrappers in the system. Thus, our research objectives, to facilitate development of a systematic robustness process, i.e., placement and compositions of wrappers in an OS span the following themes:

- A profile of possible OS vulnerabilities based on system error propagation.
- Application profiles, including (a) criticality of the applications, (b) the applications use of the OS, and (c) sensitiveness to robustness gaps in the OS.
- Error detection mechanisms ranging in type, size and complexity depending on their placement. Estimations of the properties of the detectors (e.g., completeness, accuracy, overhead in performance, size, cost).
- Error correction mechanisms following error detection; ranging from halting the system to performing advanced recovery.
- Assessment and tradeoffs - effectiveness & cost - analysis framework.

This paper focuses on the first two problems, i.e., to develop systematic methods for OS profiling. Profiles provide information on where errors in the OS are more likely to appear and cause damage. The purpose is to determine robustness gaps in the OS that can be used to guide locating and constructing effective robustness wrappers. We discuss the relevant profiling process and their experimental assessment. The result of this profiling will invariably depend on the errors considered. We specifically intend to address errors occurring in OS drivers and their impact on services the OS provides for applications.

1.2 Robustness Hardening: Related Work

Software profiling represents the process of assessing the data flow properties within the software structures. A common type of profiling is determining where the execution time of a program is spent. Such profiling can locate performance bottlenecks and even design errors in a program. Other profiles might tell the programmer which functions in the OS are used and the relative time spent performing them, e.g., the `strace` utility for UNIX-like systems, which profiles a program's use of the OS by showing the system calls made and the signals used.

In our EPIC framework (Exposure, Permeability, Impact and Criticality)[9] static modular SW (fixed set of modules that interact in a predefined manner), was profiled for error flow to ascertain the most effective placement of wrappers. The framework (and the supporting experimental tool PROPANE [8, 7]) focused on profiling the signals used in the interaction between modules. The profiling consisted of both error propagation and effect profiles. Using the permeability, and exposure metrics, propagation profiles reveal information on where errors propagate through the system and which modules/signals are more exposed to propagating errors. The error effect profiles are used to cover the cases where an error is unlikely to occur, but potentially has a high impact on system output. The software model for EPIC is static software; this makes it possible to find all communication paths in the system and then profile accordingly. In our OS themed work, the emphasis is on dynamic SW interactions. Also the set of applications is not generally known, all possible interaction paths (and consequently error propagation paths) are not known *a priori*.

In [5] errors in C-libraries were studied. A tool called HEALERS was developed which automatically tests (using fault injection) the library functions against their specification and generates wrappers to stop non-robust calls to be made. This approach differs from our profiling strategy since it only focuses on robustness in library functions. We are more interested in how errors can propagate in the system, thus allowing us to choose more than one possible location for wrappers (OS-application or OS-driver interface). However, the type of errors proposed in HEALERS are similar to the ones considered in our work.

A related approach is used in Ballista [4, 3]. Here, OS interfaces are tested with a combination of correct and incorrect parameter values and the behavior of the OS categorized according to a failure scale, ranging from "OS crash" to "no observation". Using this relatively simple approach, this method managed to find a number of robustness flaws (crashed or hung OS's) in both commercial and Open Source OS's.

Both HEALERS and Ballista are different from our proposed approach as they study the effect of malfunctioning applications on the OS whereas our method considers the effect of malfunctioning drivers on the OS. Both of these areas are important and they should be seen as complementary perspectives.

A fault injection based approach was used in [1], where the behavior of microkernel based OS's in presence of faults was studied. Faults were injected in both the memory of the kernels and in parameter values in calls to kernel primitives by applications. The effect on applications was studied along with error propagation between the kernel modules. This study differs from ours as we consider a constrained model of the OS, where internal communication details are not accessible. Also the errors considered are different. We focus on errors occurring in drivers and not in applications or internal kernel errors.

Nooks [15, 14], focuses on errors in the driver subsystem considering that drivers are a primary source of OS failures. A new subsystem layer is defined in which kernel extensions (like device drivers) can be isolated in protection domains. A driver executing in a protection domain has limited possibilities of corrupting kernel address space and objects. Nooks provides good coverage for many software and hardware faults occurring in extensions at the price of high execution overhead (up to 60% slowdown for a web server benchmark). Nooks focuses on isolating drivers from all errors impacting the OS. However, we focus on data level errors and find out which specific errors actually propagate and then protect against these. Nooks also requires a white box approach unlike our black box model of SW. Both methods are useful for malfunctioning drivers but they have different properties such as coverage and performance.

2 System & Error Model

Our intent is to develop a profiling framework applicable to diverse OS's. Thus, we utilize a generic model of a computer system, comprising of four major layers; hardware, drivers, OS and applications, see Figure 1. Each layer supplies a set of services to the next layer. For us, a service is typically a function call, i.e., for drivers it is an entry point in the driver and for the OS it is the system call or library functions. We say that a set of services makes up an interface, for instance a driver interface.

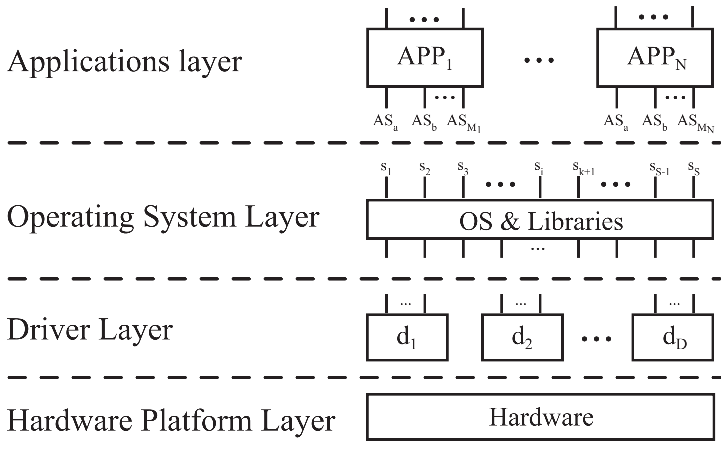

Fig. 1. General System Model

We do not assume any specific hardware architecture (CPU, disks, peripheral devices etc.), as long as the hardware layer contains all devices needed to support the remaining layers of the system.

The driver layer is responsible for handling the interaction between the hardware and the OS. Drivers are SW programs that are generally specific to a certain piece of hardware and OS. The system has a set of drivers $\mathbb{D}$, denoted $d_1, d_2, \cdots, d_D$. Each driver provides a set of services to the OS.

In the OS layer, we also include shared libraries that exist in the system. We include them in the OS layer as it reflects the point of view of a programmer, i.e., the libraries support functionalities that the programs need, very much the same as the OS. The OS provides a set of services (OS services), $\mathbb{S} = \{s_i : i \in [1, S]\}$, to be used by the applications.

The top layer is the application layer, where the programs execute performing some specific task for a user. Applications make use of OS resources either directly or through the use of libraries. The services provided in $\mathbb{S}$ may selectively be used by a certain application, i.e., each application uses a subset of $\mathbb{S}$. Each application, APP_k, uses M_k of the interfaces provided by the OS. Thus we define the set of interfaces that APP_k uses, as: $\mathbb{P}_k = \{s_i \in \mathbb{S} \text{ used by } APP_k\}$.

In this work we consider a black box SW model, i.e., we assume no knowledge about the components in the respective layers but only their interface specifications. They are supplied in binary form (for SW) and can be executed on a given hardware platform. We do not make any assumption constraining the behavior of any component when it is subjected to errors/stresses. Such stress conditions could be heavy load on a component, misuse of its interface or resource depletion.

2.1 Error Model

Focusing on a black-box model of the system, the only place where monitoring/wrapping can take place is at the interfaces between the layers in the system. We define two major interfaces, between the OS and the drivers and the OS and applications. We consider data level errors occurring in the OS-driver interface as our primary error model. Recent studies have demonstrated that drivers are a major source of OS failures [2, 15, 12]. This arises as they are often not tested as rigourously as the OS kernel; they may have been designed external to the OS development team, lacking complete details of the system; and they may also be affected by malfunctioning hardware. Drivers represent a large part of an OS package in sheer size which further warrants their focused consideration (as the number of expected faults generally increases as the size increases). We focus on data level errors as they are possible to use with a black box system. They also represent a set of detectable errors, i.e., one can define detectors in the form of assertions to detect them.

In other robustness studies different fault/error models where used, mostly bit-flips in parameter values and/or data and code memory areas of programs [7, 1, 6]. Instead, we focus on data level faults at interface levels as they best represent actual SW errors.

A data level error implies that the value of the data communicated through the interface is “erroneous”, i.e., not the expected value according to some omniscient viewer. Thus, the “state” of the program changes, and this error propagates to the OS causing subsequent errors. To simulate data errors, we specifically change the values of parameters used in the targeted interface. The type of error injected is decided by the type of the parameter used. It is important to note that the chosen error model and its representativeness of actual OS errors, fundamentally affects the relevance of any obtained results.

3 Measures of Error Propagation

The purpose of a profiling framework is to outline SW characteristics for a designer. The desired characteristic also drives the method used to gather data. Our focus is on effective location of OS wrappers to handle driver errors. Thus, we target determining the specific drivers that have a higher likelihood of error propagation. Similarly, on error occurrence, to establish which OS services are more exposed to these errors.

The final goal of our profiling is to estimate the impacts errors have on the services provided by applications. The problem is that profiling with respect to a certain set of applications running on the system will not give the same result as a different set of applications. The way applications use the OS and their importance will influence the results of the profiling. To capture this distinction we propose to use a separate profile for an application’s use of the OS. This profile must include information revealing which OS services the application depends upon and notions of each service importance to the application (some are most likely more important for the continued functioning of the application than others). To analyze a system with more than one application, with varying importance, priorities must be established across them. This can then be used when the assignment of wrappers is made.

3.1 Operating System Profiles

The OS profiles consider how errors in drivers spread through the OS to its services for applications. These profiles are naturally specific to an OS and its drivers. The first measure (*Service Error Permeability*) defines the probability of errors propagating through the OS. This measure is used to ascertain which OS services are more susceptible to propagating errors (*OS Service Error Exposure*) and which drivers are more likely to spread them (*Driver Error Diffusion*).

Service Error Permeability The goal of OS profiling is to characterize how errors in drivers influence the OS services provided. We start by defining the service error permeability, $P^{j,k}$ for an OS service $s_j \in \mathbb{S}$ and a driver $d_k \in \mathbb{D}$.

$$P^{j,k} = Pr\left(\text{error in } s_j | \text{error in } d_k\right) \tag{1}$$

Eq. 1 describes the relation of one driver to one OS service. This measure gives an indication of the permeability of the particular OS service, i.e., how 'easily' does the service let errors in the driver propagate to applications using it. A higher probability naturally means that either (a) the driver needs to be enhanced to make sure that it does not produce errors, or (b) some detection is needed in the application or elsewhere to handle errors propagating.

OS Service Error Exposure The OS error permeability considers discrete drivers on a stand-alone basis. To ascertain which OS service is the most sensitive to errors propagating through the OS, then more than one driver needs to be considered. We use the measure OS error permeability, to compose the *OS Service Error Exposure* for an OS service s_j, namely E^j:

$$E^j = \sum^{\forall d_k \in \mathbb{D}} P^{j,k} \tag{2}$$

For E^j we consider the set of all drivers, $\mathbb{G}$. If one driver does not affect a service, for instance it is not at all used by that service, then the OS error permeability will be zero and thus should not affect the OS service error exposure. The Service Error Exposure gives an ordering across OS services (given that more than one service has been tested and have service permeability values) which orders services based on their susceptibility to errors passing through the OS kernel. With this measure the tester can focus attention to particular OS services that may require more work or as a means to place wrappers.

Driver Error Diffusion One might not only be interested in finding out which services are more exposed to errors but also which driver is spreading them the most, i.e., which driver, if acting faulty, has the potential of spreading errors the most in the system. To find these drivers we define a measure that considers one particular driver's relation to many services, *Driver Error Diffusion*, D^k, for a driver $d_k \in \mathbb{D}$ set of services:

$$D^k = \sum^{\forall s_j \in \mathbb{S}} P^{j,k} \tag{3}$$

The Driver error diffusion also creates an ordering across the drivers. It ranks the drivers according to their potential for spreading errors in the system. Note that we do not try to test the drivers per se, so this measure only tells us which drivers **may** corrupt the system by spreading errors. It is actually a property of the OS and not the drivers.

3.2 Application Profile

To estimate the effect of errors in OS services, we need to establish the applications usage of these services, specifically (a) which services are invoked and (b) their respective "importance". The *importance* of the service is defined with respect to how critical it is to the continued functioning of the application. If an error in an OS service always causes the application to crash, we say that this OS service is important to the application. On the other hand, if it does not have any effect on the application, because the application has built-in error correction, it is of no importance. The importance $i_j, 0 \leq i_j \leq 1$ of an OS service $s_j \in \mathbb{P}_k$ is defined as the probability that given an error in an OS service, it causes the application APP_k to fail. $\mathbb{A}_k$ is a set of tuples, (s_i, i_i). For each OS interface $s_i \in \mathbb{P}_k$ used by the application there is one and only one such tuple. Each tuple includes the service itself and the value i_i which describes the importance of the service to the application. The application profile can and should be constructed before the system is deployed, i.e., not during run-time.

$$\mathbb{A}_k = \{(s_i, i_i) : s_i \in \mathbb{P}_k\} \tag{4}$$

A third parameter of interest is the *criticality* of the application. If there is more than one application running on the system, some of them might be more important than others. We then say that it has a higher degree of *criticality.* For comparing profiles across applications we need to know if adding wrappers to "help" a certain application is more important than another application. In EPIC [9] a scale from 0 to 1 was used for weighing the output signals according to their criticality. 1 indicated the highest possible criticality and a 0 indicated that the output signal is non-critical. A similar scale is used here too, indicating a range from non-critical to highly critical. A criticality value, C is thus assigned to every application APP_k, $0 \leq C_k \leq 1$.

3.3 Application Service Exposure

The Service Error Exposure and the Application profile together determine how one application is affected by errors occurring in drivers. This can be used to establish wrapper locations, given a set of drivers and applications. Potentially this trade-off can be made online (if applications are shipped profiles) as applications comes (and leaves) the system.

$$SE_k = C_k \cdot \sum^{\forall s_i \in \mathbb{A}_k} \left(E^i \cdot i_i\right) \tag{5}$$

Eq. 2 & 4 aid in predicting how an application will react to errors propagating in the system. The *service exposure* composes the error permeability profiles with the application profiles, to predict the behavior of the application. For an application APP_k, given the profile $\mathbb{A}_k$ and the corresponding service propagation values. The values of the application error exposures are used to create a relative ranking of applications and also the wrapping priority.

4 Experimental Evaluations

To make use of the analytical framework presented in the previous section, values for the profiles must be obtained. Code inspection analyses, expert analyses, bug reports/error logs and fault injection (FI) experiments are all possible approaches in this respect. We have chosen FI as its utility has been established (and also used in the EPIC framework using the PROPANE tool [8, 7]); it is also usable at design time (in contrast to bug reports and error logs) and it can be used without the full source code availability.

As indicated in Sec. 3 the only measure that needs to be experimentally estimated is OS Error Permeability, $P^{j,k}$, as both Service Error Exposure and Driver Error Diffusion can be derived from this measure. To get an experimental estimation of $P^{j,k}$ we will inject faults in the interface between driver d_k and the OS and monitor the result of executions of service s_j by writing an application that uses this service. We estimate the error permeability as the ratio of detected errors, $n_{detected}$ at the service level to the number of injected errors $n_{injected}$.

$$P^{i,j}_{est} = \frac{n_{detected}}{n_{injected}} \tag{6}$$

The values of E^{j}_{est} and D^{j}_{est} can be calculated using $P^{i,j}_{est}$ using Eq. 2 & 3. Naturally not all application cause usuage of all drivers, or all of their services, so the FI experiments will be limited to the driver services actually used. Eq. 6 requires detecting that an error has actually propagated. There are several types of outcomes from a FI experiments and they can be classified in different categories. We use a failure mode scale similar to the one used in [3] as:

- No visible error at the OS service
- Error propagated, but still satisfied the specification of the service
- Error propagated but violated the specification of the service
- The OS hung due to the error
- The OS crashed due to the error

The first category is straightforward as the error injected caused no visible error at the OS service level. The error might be dormant within the system, but in order to limit the time to perform the experiments a timeout will be defined after which the error is considered not to have propagated.

For the second category it must be clear what is meant by a specification. In this work we consider the specification given to a programmer for the OS. This can be for instance `man` files for Linux/UNIX or the help sections for a Windows computer. An example of an outcome that will end up in this category is (a) when an error code or exception is returned that is a member of the set of allowed codes for this call, or (b) if a data value was corrupted and propagated to the service, but did not violate the specification.

The third category contains those outcomes where the result is violating the specification of the service. For instance returning a string of certain length when another parameter specifying the length holds a different number. Raising an unspecified exception or returning an unspecified error code also ends up in this category. If the application hangs or crashes but other applications in the system remain unharmed, they end up in this category as well.

If the OS hangs, then no service progresses. This state must be detected by an outside monitor. The difference to the crash category is that the latter produces some form of dump, indicating that the OS has crashed and some additional information. We separate these categories because a crashed computer might be easier to detect and correct (by restarting) than a hung one which inherently must be detected by some watchdog timer.

4.1 Inserting Probes

Sec. 3 outlines the need to facilitate monitoring between the system layers, see Fig 1. Thus, we need to insert probes between the application and OS layers and between the OS and drivers layers. These probes are needed to (a) monitor the communication between the layers in the system and (b) for actually inserting perturbations in this communication, i.e., simulating the occurrence of errors. The first case, can be achieved simply by writing special purpose applications which sole purpose is to actively use the services provided by the OS. To achieve the latter monitoring we design a new driver that is loaded instead of the driver which we want to monitor/wrap. The needed "wrapping driver" is to first load the existing driver and then set up all data structures. Then it passes on any calls from the OS to the real driver and the result back to the OS. None of the methods interfere with our intent to use black-box methods since it does not involve modifications of the OS and/or the existing applications or drivers. Some reconfiguration may be needed but access to source code is not required.

4.2 Estimating Application Profile

To obtain the application service exposure presented in Sec. 3.3 (using Eq. 2 and 5) we need to experimentally derive the application profile $\mathbb{A}_k$ and the error permeability values for each driver $P^{j,k}$. The application profile is derived using a profiling tool that executes the application and produces a list of calls made to the OS. The calls are matched against $\mathbb{S}$ to produce $\mathbb{A}_k$. For each interface that one application uses, an importance value must be assigned. The importance value i_i can be derived using a tool that injects faults in the output of system calls made by the application. Calls are intercepted by additional wrapper layer, the actual call is made to the OS and the return value can then be altered, i.e., an error can be inserted and the result is passed on to the OS. A similar approach was used for the Fuzz tool where UNIX utilities were fed with random input strings and their robustness depended on their response to these streams [11]. Another similar approach was used for Windows NT applications in [13]. The results from a similar tool can be used to assign the importance values.

Finally the criticality value must be assigned. This value represents the criticality of an application relative to other applications. It is used to bias the application service profile towards the applications that are of higher importance. This value is assigned by the profiler and it depends on the set of applications present. C_k is a value between 0 and 1, where 1 indicates high criticality.

5 Discussion and Future Work

As noted in Sec. 3, the profiles presented will only provide a relative ordering across OS services, drivers and applications. To get real values for the probabilities of real errors occurring we need to know precise nature and the arrival rate of real errors. However, our method is targeted at placement of wrappers. It illustrates to the profiler which services are more exposed to errors and thus require wrapping. Note that this is always an estimation based on the probabilistic nature of testing. Additional knowledge about the system can be used to complement the profiling. For instance, if empirical knowledge exists about the "quality" of different drivers, this can be used in the profiling to exclude certain drivers from the profile or weigh their scores so that their impact is reduced.

To be able to decide, with the help of the profiles, if more resources should be spent within the system on adding wrappers, some notion of robustness level is needed. We need methods for determining when enough wrappers have been added to the system. Possible heuristics can entail removing all potential crashes of the system and then one by one removing the less severe errors until a requisite number have been removed incrementally.

Another issue that may impact the location of wrappers is the wrappers themselves. Each wrapper comes with an associated cost, in size and overhead. The cost-efficiency trade-off is part of our intended future work, as well as the actual design and evaluation of wrappers. For evaluating wrappers, important parameters include the completeness and the accuracy, i.e., how many of the real errors are detected and how many mistakes are made. Also, the more complex a wrapper is, the higher its execution cost is. A coverage versus performance trade-off is an essential consideration.

A desired property of wrappers is to be able to generate them during run-time. When an application is loaded for execution, its associated profile is composed with the OS profile to find out if the current placement of wrappers in the system is optimal. We therefore need both the means to do the trade-off during run-time as well as the means to instantiate and deactivate wrappers without interfering with the functioning of the rest of the system.

A limitation of our approach is that dependencies between the application and the OS might exist that we currently do no cover. For instance we do not consider the order in which calls are made by an application to the OS. The ordering of calls, with respect to errors, might have an impact on the resulting behavior of the application. Investigating such dependencies and their implications for wrapper placement is part of future extensions for our work.

When determining if an error propagated, one needs to know what constitutes an error for a service. This information can (ideally) be derived from the specification of the service. This is not always possible due to the lack of or incompleteness of the specifications given. In the past, many fault injection experiments have utilized a so called golden run, i.e., a test program is executed without faults and the outcome is then compared with one run with faults presents to find deviations, i.e., errors. For OS's creating a golden run is non-trivial given the non-determinism in scheduling etc. One option would be to restart the system before every experiment and then conduct the tests (golden runs as well as FI experiments). Another option would be to run several tests without faults and use a mean behavior as the golden run.

6 Summary

Overall, the proposed profiling framework described in this paper aids a designer to effectively enhance the OS with error protection wrappers. By studying how errors propagate from drivers through the OS, potentially exposed OS services can be identified. Combining this with knowledge on the dependencies between the OS and an application, a suggested placement of wrappers can be obtained.

References

1. J. Arlat et al. Dependability of COTS Microkernel-based Systems. *IEEE Trans. on Computers*, 51(2):138–163, 2002.
2. A. Chou et al. An Empirical Study of Operating System Errors. In *Symposium on Operating Systems Principles*, pp. 73–88, 2001.
3. J. DeVale and P. Koopman. Performance Evaluation of Exception Handling in I/O Libraries. In *Proc. DSN*, pp. 519–524, 2001
4. J. DeVale and P. Koopman. Robust Software - No More Excuses. In *Proc. DSN*, pp. 145–154, 2002
5. C. Fetzer and Z. Xiao. An Automated Approach to Increasing the Robustness of C Libraries. In *Proc. DSN*, pp. 155–164, 2002
6. W. Gu et al. Characterization of Linux Kernel Behavior Under Errors. In *Proc. DSN*, pp. 459–468 , 2003
7. M. Hiller, A. Jhumka, and N. Suri. PROPANE: An Environment for Examining the Propagation of Errors in Software. In *Proc. of ISSTA*, pp. 81–85, 2002
8. M. Hiller, A. Jhumka, and N. Suri. An Approach for Analysing the Propagation of Data Errors in Software. In *Proc. DSN*, pp. 161–170, 2001
9. M. Hiller, A. Jhumka, and N. Suri. EPIC: Profiling the Propagation and Effect of Data Errors in Software. in *IEEE Trans. on Computers*, pp. 512-530, May 2004
10. C. Michael and R. Jones. On the Uniformity of Error Propagation in Software. In *Proc. of COMPASS*, pp. 68–76, 1997
11. B. Miller et al. An Empirical Study of the Reliability of Unix Utilities. *CACM* 33(12):32–44, 1990
12. B. Murphy and B. Levidow. Windows 2000 Dependability. In *Proc. of the Workshop on Dependable Networks and OS, DSN*, 2000
13. M. Schmid et al. Techniques for Evaluating the Robustness of Windows NT Software. In *Proc. of DARPA Information Survivability Conference & Exposition*, volume 2, pp. 1347–1360, 2000
14. M. Swift et al. Nooks: An Architecture for Reliable Device Drivers. In *Proc of the Tenth ACM SIGOPS European Workshop*, pp. 101–107, 2002
15. M. Swift et al. Improving the Reliability of Commodity Operating Systems. In *Proc of SOSP*, pp. 207–222, 2003