Error Propagation Profiling of Operating Systems

Andréas Johansson and Neeraj Suri
Department of Computer Science, Technische Universität Darmstadt
{aja,suri}@informatik.tu-darmstadt.de

Abstract

An Operating System (OS) constitutes a fundamental software (SW) component of a computing system. The robustness of its operations, or lack thereof, strongly influences the robustness of the entire system. Targeting enhancement of robustness at the OS level via use of add-on SW wrappers, this paper presents an error propagation profiling framework that assists in a) systematic identification and location of design and operational vulnerabilities, and b) quantification of their potential impact. Focusing on data (value) errors occurring in OS drivers, a set of measures is presented that aids a designer to locate such vulnerabilities, either on an OS service (system call) basis or a per driver basis. A case study and associated experimental process, using Windows CE .Net, is presented outlining the utility of our proposed approach.

1. Introduction

The robustness of the OS, i.e., the ability to withstand perturbations with continued service provision, directly affects the robustness of the whole system. As applications depend on the OS delivering correct and timely services, a failure to do so will likely impact the system's ability to deliver its services. In spite of the significant ongoing efforts to enhance OS robustness, and advances in the underlying SW development processes, OS's still occasionally fail to deliver stipulated services. One reason is the complexity of OS level interactions with varied SW components in the system that are often unknown at OS design time, e.g., applications, drivers, libraries. This "dynamic" nature of OS's makes it hard to design an OS to be comprehensively robust against errors that can occur in its operational environment.

This paper focuses on delivering a methodology whereby the OS platform can be profiled using experimental fault injection (FI) and data error propagation path analysis. This profiling characterizes the behavior of the OS when exposed to perturbation in its interaction with other system components. The targeted errors are errors occurring in device drivers. Drivers are reported to be a significant source of OS failures [5, 16, 19]. This arises as they are often not tested as rigorously as the OS kernel; are often designed external to the OS development team, lacking complete details of the system; represent a significant part of the system in terms of code size; and may also be directly affected by malfunctioning hardware or external errors. Throughout this paper, errors refer to data level errors, i.e., errors that affect the value of some variable in a program.

The goal of this paper is to demonstrate a *quantifiable* and *repeatable* method for assessing error propagation for an OS, where *no source code access* is viable, as is the case for most commercial OS's. The lack of source code poses two main obstacles: (a) no changes can be made internally to the components, neither the OS nor its drivers, and (b) the error propagation analysis must be based on observations made only on the interface between components. Error propagation information reveals which errors occurring in the OS environment (device drivers) will "flow through" the OS and ultimately affect applications. The non-dependency on source code also makes it possible to target systems without source code readily available. It also avoids the problem of having the code bias the testing results, which sometimes is a problem for white-box testing approaches.

To demonstrate the proposed profiling methodology a case study is presented, using the Windows CE .Net operating system. The target was chosen for its OS representativeness and also as it offers a high degree of control and limited complexity. The results from the case study are interpreted using a set of measures, defined to capture the error propagation properties of the OS. We study the relations between individual OS services to services in drivers as well as composite metrics describing the exposure of errors on OS services and the diffusion of errors from specific drivers. Two drivers are studied and the error diffusion properties of the two drivers are compared. The main results of the study conclude that many errors do not propagate, or propagate in a robust manner, i.e., the effect of an error is visible at the application level, but does not lead to a failure. However, a few significant error propagation paths are identified which represent a serious threat to a highly robust design.

To recap, the contributions found in this paper include:

- Development of OS error propagation measures.
- *Quantifiable* methods for assessing the aforementioned measures *without* source code availability.
- A case study to demonstrate the experimental assessment process.

Paper Organization: Sec. 2 puts our work in context by discussing related work. Sec. 3 presents the system and error model used in this work. Sec. 4 develops and details the measures used for profiling an OS for errors in drivers. Sec. 5 and 6 present the case study and obtained results. Analysis and discussion of the results and future research directions are presented in Sec. 7. Sec. 8 presents conclusions.

2. Related Work

Robustness studies of large SW systems have been conducted for many years. In [8] the robustness of C-libraries was evaluated and enhanced using wrappers. POSIX interfaces are the targets in [14], where OS's are compared using failure mode analysis. The techniques used in [14] correspond, in part, to ours (injection strategy, error model, etc), but with applications instead of drivers as the source of errors. The effects of driver errors on the Linux kernel were studied in [1]. The main difference with our approach is that we focus on error propagation measures, as a facilitator for wrapper placement. The work presented herein complements [1] in this respect, as the approaches and error models are similar. In [2] an extensive failure mode analysis was performed, using code mutations in drivers with the purpose of building a dependability benchmark. Micro-kernels have also been targets for studies [3], where bit-flips in either kernel API's or memory are used to simulate hardware (HW) as well as SW faults. The main contribution of our work, with respect to the other studies, are again that we focus on quantifiable measures of error propagation. Also, we focus specifically on data level errors, whereas other studies have used other error models such as bit-flips [3] and code mutations [2].

In our prior work, the EPIC framework (Exposure, Permeability, Impact and Criticality), together with the supporting experimental tool PROPANE [11, 12], profiles static, modular software (fixed set of modules that interact in a predefined manner) for error propagation to ascertain effective placement of wrappers. The framework focuses on profiling the signals used in the interaction between modules using 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. In our current OS-themed work, the emphasis is on *dynamic* software interactions as the set of applications is not generally known in advance. Thus, all possible interaction paths (and consequently error propagation paths) are not known *a priori*.

Tools have also been developed with the specific purpose to protect a system from malfunctioning drivers. In [19] drivers are wrapped to track erroneous memory accesses, which is a major problem in device drivers. One issue raised in [19] was that some data level errors cannot automatically be traced. We believe that our approach complements [19] to this end. In [18] an *Interface Definition Specification* is used to improve driver robustness. In the Microsoft SLAM project [4] a static verification technique is used to verify that device drivers adhere to a given interface. This project effectively has the same goal as ours, with the difference that we use no source code knowledge or formal specification of the interfaces. Both approaches are promising and will likely coexist for different scenarios.

3. System and Error Model

Given the diversity of OS implementations, we target a simple system model that comprises four major layers; hardware, drivers, OS and applications, see Figure 1. This model is generic enough to be representative and applicable to most modern OS's, e.g., Windows, GNU/Linux and UNIX. Each system layer supplies a set of services to its neighboring layers. For us, a service is typically a function call, e.g., function entry points in drivers and OS system calls. We say that a set of services constitutes an interface, for instance the OS-driver interface.

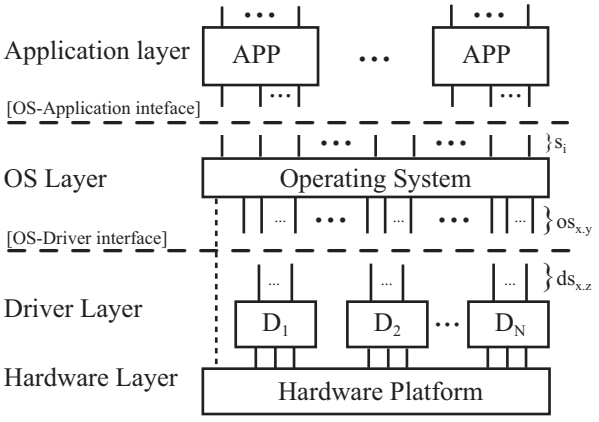

Figure 1. Generalized system model.

The driver layer is responsible for handling interactions between the hardware and the OS. Direct interactions without the involvement of a driver are indeed possible, however, we focus only on the general case of interactions with

driver involvement. We have chosen to treat drivers as components of the *system* and not as a subcomponent of the OS. One may argue that drivers are actually an intrinsic part of the OS, but our view facilitates fuller representation of component interactions.

The system has a number of drivers enumerated as $\{D_1, D_2, \cdots, D_N\}$. Each driver both imports (for instance via system calls) and exports services to the OS (provided by the driver) and we consider each category of driver-OS interactions separately. Each driver exports a set of services. These can be enumerated as $\{ds_{x.1}, ds_{x.2}, \cdots, ds_{x.M}\}$, where $ds_{x.y}$ is the y^{th} service exported by driver D_x.

In the OS layer, we include shared libraries existing in the system, reflecting the viewpoint of a programmer, i.e., libraries support functionalities that programs need. For this layer we define two primary sets of interfaces where interaction takes place. First, through the OS-Application interface, the OS provides a set of services $\{s_1, s_2, \cdots, s_S\}$, through which applications interact with the OS. The second interface contains the interactions between the OS and the drivers (through the OS-Driver interface). For a specific driver D_x, the set of services it imports from the OS is denoted $\{os_{x.1}, os_{x.2} \cdots os_{x.K}\}$. Using the same notation as for driver services, $os_{x.q}$ is the q^{th} imported OS service by D_x. Note that the same services can be used by both drivers and applications.

Applications running on the system utilize services provided by the OS to, in turn, provide services to the users of the system. Thus, as different OS services have different robustness properties, the way applications make use of OS services will impact the robustness of their operations.

Throughout this paper we use a black-box software model, i.e., no source code level knowledge is assumed about the components, only their interface specifications. This system view corresponds to the view of developers using the system for a particular design. The specification can be textual, like a reference manual, but also header files needed for compiling a program are considered to be part of the specification. Components are supplied in binary form. We assume that we have access, and possibilities to inspect and modify the binary image of the driver and the possibility of inspecting and modifying them such that that the interaction between the layers can be monitored and modified. Note that this does not require source code access!

3.1. Error Model

Our error model is transient data level errors, which are the result of implementation defects in the driver as well as value faults related to malfunctioning HW. This error model, when used for error injection, has three basic components: a) *the type, or nature of the error*, b) *the location of the error*, and c) *the timing of the injection*. In the following paragraphs we define each of these properties.

Error Types: For this study we chose data level errors, based on the C-type of each parameter, i.e., the data type used for the parameter. We use C-types as the OS-Driver interface for this target is defined using the C language. We find data errors of high interest as they are hard to detect with general error detection techniques (like tracking of memory accesses) and may still lead to severe failures. They also allow for flexible insertion using a black-box view of the system. Sec. 2 discusses other error models found in literature.

Error	C-Type	#Cases
Integers	`int`	7
	`unsigned int`	5
	`long`	7
	`unsigned long`	5
	`short`	7
	`unsigned short`	5
	`LARGE_INTEGER`	7
Void	`* void`	3
Characters	`char`	7
	`unsigned char`	5
	`wchar_t`	5
Boolean	`bool`	1
Enums	multiple cases	#identifiers
Structs	multiple cases	1

Table 1. The error types used in this study.

We have currently implemented this error model for 27 different C-types, both basic and user defined (here the "user" refers to the OS). Table 1 shows an overview of the types needed to perform the case study presented later in Sec. 5, and shows the number of test cases defined for each type. For each C-type, different data values are chosen as test cases, including offset values, common values and boundary values. The values are chosen to both cover equivalent classes of values and boundary values. As an example, table 2 shows the errors for type `int`. Note that Cases 1 and 2 allow for under/overflow to happen.

Case #	New value
1	(Previous value) - 1
2	(Previous value) + 1
3	1
4	0
5	-1
6	INT_MIN
7	INT_MAX

Table 2. Error cases for type `int`.

Error Location: As already mentioned, we focus on data errors in the OS-driver interface. By introducing errors in the parameters used in this interface we simulate errors

occurring within the driver in question. An example of an OS service in this interface is shown below:

```
LONG RegQueryValueEx([in]     HKEY    hKey,
                     [in]     LPCWSTR lpValueName,
                     [in]     LPDWORD lpReserved,
                     [out]    LPDWORD lpType,
                     [out]    LPBYTE  lpData,
                     [in/out] LPDWORD lpcbData);
```

For each service, the input/output parameters are identified (using the documentation). Each output parameter of a driver service $ds_{x.y}$ is targeted for injection as well as each input parameter for the OS-driver services $os_{x.q}$, see Sec. 3. An error is injected after a call to/from a driver is made by changing the value of a parameter and then continuing the execution with the corrupted value.

Error Timing: Each error is injected once, thus simulating transient service corruption. We do not consider permanent errors, occurring every time a function is called, as these are easier to detect using normal functional testing techniques. Transient errors better represent the rare cases that are not easily detectable with normal testing techniques.

Each error is injected on first occurrence, meaning that the first call made to the service in question will be the target for the injection. Previous studies on OS's indicate that "first occurrence" injection provides comparably effective results to other injection instances [20].

4. Error Propagation Measures

Based on the initial discussion presented in [13], we have defined a set of quantifiable measures that guide the location of vulnerabilities. An effective placement of wrappers is where errors are likely to occur (high probability of error propagation) and where the impact of errors is highest [12] (the consequences of errors are system failures). Therefore, these are the objectives for our measures:

- **(a)** Measure for degree of error porosity of an OS service: *Service Error Permeability*
- **(b)** Measure for error exposure of an OS service: *OS Service Error Exposure*
- **(c)** Measure of driver error co-relation with service set: *Driver Error Diffusion*

It is important to note that the measures presented below implicitly use a uniform distribution of errors. This is a consequence of the fact that no "profile" exists describing how the system is used in a real scenario. A longer discussion on this aspect can be found in Sec. 7.

4.1. Service Error Permeability

We define two measures for error permeability, one for a driver's export of services (P**D**S) and one for its import of OS services (P**O**S). For a driver D_x, the set of export services ($ds_{x.1} \cdots ds_{x.N}$) and the import services ($os_{x.1} \cdots os_{x.M}$), see Sec. 3. The *Service Error Permeabilities*, for export ($PDS^i_{x.y}$) and import ($POS^i_{x.z}$) of services, relate one driver service to one OS service. The Service Error Permeability is the conditional probability that an error in a specific driver service ($ds_{x.y}$) or in the use of OS-driver service ($os_{x.z}$) will propagate to a specific OS service (s_i). For an OS service s_i and a driver D_x:

$$PDS^i_{x.y} = Pr\,(\text{ error in } s_i | \text{error in } ds_{x.y}) \qquad (1)$$

$$POS^i_{x.z} = Pr\,(\text{ error in } s_i | \text{error in use of } os_{x.z}) \qquad (2)$$

The *Service Error Permeability* (PDS for exported and POS for imported services of a driver) 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) a wrapper needs to be designed to protect the OS from driver errors, or b) applications using the affected services needs to take precautions when using them. Note that Equation 2 allows us to compare the same OS service used by different drivers. The impact of the context induced by different drivers can thus be studied.

4.2. OS Service Error Exposure

To ascertain which OS service is the more exposed to errors propagating through the OS, the full set of drivers needs to be considered. We use the measure Service Error Permeability, to compose the *OS Service Error Exposure* for an OS service s_i, namely E^i:

$$E^i = \sum_{D_x} \sum_{os_{x.j}} POS^i_{x.j} + \sum_{D_x} \sum_{ds_{x.j}} PDS^i_{x.j} \qquad (3)$$

For E^i we consider all drivers. The OS Service Error Exposure gives an ordering across OS services which orders services based on their susceptibility to errors passing through the OS. Note that this expression implies aggregating all imported and exported Service Error Permeabilities (1 & 2 above). The OS Service Error Exposure indicates which services are more exposed to propagating errors; its use is thus mainly to direct placement of wrappers on the OS-application level.

4.3. Driver Error Diffusion

The measure *Driver Error Diffusion* identifies drivers, that when being erroneous are more likely to spread errors, by considering one particular driver's relation to many services. Driver Error Diffusion is a measure of how a driver

can impact OS services (at the OS-Application interface). The more services, and the higher the impact (permeability) a driver has, the higher the value. For a driver D_x and a set of services, the Driver Error Diffusion, D^x is:

$$D^x = \sum_{s_i} \sum_{os_{x.j}} POS^i_{x.j} + \sum_{s_i} \sum_{ds_{x.j}} PDS^i_{x.j} \qquad (4)$$

The Driver Error Diffusion ranks the drivers according to their potential for spreading errors in the system. Analogous to the OS service exposure, the driver diffusion aggregates Service Error Permeabilities. This gives the system designer hints on where wrappers should be placed, i.e., where adding driver wrappers makes sense. 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. Also, we emphasize that the intent of these measures is *not* for absolute values, but to obtain relative rankings.

Once a ranking across drivers is achieved, the driver(s) with the highest Driver Error Diffusion value should be the first targets. Details on specific error paths can now be used (i.e., Service Error Permeability values) to guide the composition and exact placement of wrappers.

4.4. Error Exposure vs Error Impact

The purpose of error propagation profiling is to reveal prominent error propagation paths as well as to identify those that can have a severe impact on the whole system. The measures described in the previous subsections aid in identifying the common error paths. However, the impact can range from no effect at all to the whole system being rendered unusable (e.g., crashed or hung). Thus, it is important to not only measure *if a failure occurred*, but also what *type of failure it was*. Failure mode analysis is an approach that accomplishes just this. A set of failure modes are defined and the outcome of each experiment is classified as belonging to one of them. The classes used in this study follow the classes established in [3, 6, 10]:

Class NF: When no visible effect can be seen as an outcome of an experiment, the *No Failure* class is used. This indicates that the error was either not activated or was masked by the OS.

Class 1: Error propagated, but still satisfied the OS service specification as defined in the documentation. Examples of **Class 1** outcomes are when an error code or exception is returned that is a member of the set of allowed codes for this call or if a data value was corrupted and propagated to the service, but did not violate the specification.

Class 2: Error propagated and violated the service specification. For example, returning an unspecified error code or if the call directly causes the application to hang or crash but other applications in the system remain unharmed, result in this category. Note that not properly handling a return value within the specification does not end up in this class!

Class 3: The OS hung or crashed due to the error. If the OS hangs or crashes, no progress is possible. For a crashed OS, this state must be detected by an outside monitor unless this state is automatically detected internally and the machine is rebooted. Again note that not handling **Class 1** failures resulting in an eventual crash is not treated as a **Class 3** failure.

Using this severity scale, further analysis of the results can be done. Starting with the most severe class (**Class 3**: crash/hung) one can study the exposure to these errors separately, then progressively go downwards in the scale.

5. Case Study: Windows CE .Net

We present a case study to demonstrate the utility of the measures defined in Sec. 4. A commercial OS (Windows CE .Net), together with two drivers, is the target for this study. By use of error injection, the measures defined in Sec. 4 are estimated. Experiments are conducted in rounds, with one error being injected in each round. To assure a consistent system state for each experiment the system is rebooted between rounds.

The rest of this Section contains a description of the target system followed by a description of the experimental setup used, its modules and their roles. Subsequently, details on the experimental process and the estimation of measures are presented.

5.1. Target System

The target of the case study is Windows CE .Net 4.2. The OS runs on a HW reference platform using the Intel PXA255 XScale board (similar to what is found on modern PDAs). The configuration has a 400 MHz processor, with 32 MB flash and 64 MB SDRAM memory, and is equipped with serial ports as well as Ethernet connections. Four separate boards were used to achieve reproducibility of results as well as for expediting the injection processes.

For all experiments, the OS image is configured to use a minimum set of components to facilitate repeatability. Thus components relating to unused HW (Keyboards, mice, etc) and graphical components as well as possible services (web servers, etc) are excluded.

We target two drivers, an Ethernet driver, `91C111.Dll`, and a serial port driver, `cerfio_serial.Dll`. Both are shipped with the hardware platform as binaries. These drivers are chosen as

they a) represent functionality present in many products, and b) are supplied by third party vendors, which prohibit source code access, thus demonstrating the utility of our black-box approach.

5.2. Experimental Setup

In order to estimate the Service Error Permeability, we inject errors in the OS-driver interface and study their effects on the application-OS interface. We use a special driver "wrapper", termed *Interceptor*, which bypasses the normal driver-OS interactions and provides the means for injecting errors at this interface.

The entire setup consists of four main modules, (apart from the target OS); a) the aforementioned interceptor; b) test applications, c) an experiment manager application; and d) a host computer, see Figure 2.

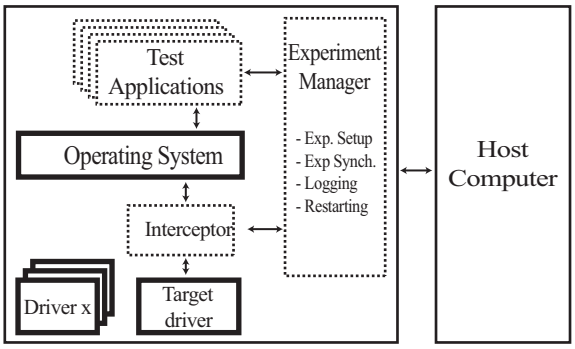

Figure 2. The experimental setup - dotted boxes indicate add-on modules to the setup.

Interceptor: Two methods are used to intercept OS-driver interactions, a) the changing of function table entries in the binary of the driver, and b) re-configuration of the Windows registry. Changes made to the binaries are made before the driver is loaded in the system. By reconfiguring the loading of drivers (in the registry) the interceptor module can be loaded instead of the driver, and thus act as a driver for the OS.

Test Applications: The purpose of the test applications is to exercise the OS services. Ideally real applications should be used. However, the use of custom test applications simplifies the task of exercising specific parts of the OS and detection of propagating errors. Sec. 7 contains a longer discussion on the use of real applications.

Four applications are used, that use the OS in different ways. Three applications are dedicated to test a) the memory management subsystem; b) thread management and synchronization primitives; and c) file system operations. The fourth application is dedicated to the specific driver being tested. Driver-specific applications are simple echo applications that exchange data with a host server. Each application is manually equipped with assertions, checking each call to the OS for any irregularities. Assertions are based on the imported OS services by an application. These are matched to calls being made in the application using the specification of the service.

Experiment Manager: The experiment manager reads a configuration file from persistent memory and notifies the interceptor of the error to be injected. During an experiment the data entries in the configuration file are updated such that after a reboot the manager can tell if the previous experiment ended in a failure or not. The experiment manager is also responsible for starting the test applications and rebooting the device after each experiment. Log messages are sent to the host machine, which stores them as text files.

Host Computer: Its role is to receive log messages from the device (over serial communication or Ethernet) and log them to files on the host machine. The host also runs the test servers, used in communicating with the test applications.

5.3. Experiments: Procedures

The first application to start during boot-up is the experiment manager. It reads the configuration files and sets up communication with other modules and the host machine. The manager also starts a timer, that will reboot the system after a set time (currently three minutes), long enough for all test applications to terminate during failure free scenarios ($>$ execution time + boot up time). The purpose of the timer is to reboot the system if the experiment manager has crashed as a result of an error. The interceptor is loaded together with the target driver. The error is injected at the first instance of the call to the function in question. After the system has booted up, the experiment manager starts the test applications and monitors their progress.

If no failure occurs, the test applications terminate successfully; this is logged by the experiment manager. If the error is activated (i.e., actually executed by the driver) this is logged as well. After the test applications have terminated, the result is logged.

If an error occurs, this is either registered by a test application or by the experiment manager noticing that either the driver/test application has crashed. Assertions in the test applications allow detection of propagated errors. Each time an error code is returned from a call this is logged as a propagation. Distinction between expected and unexpected error codes is made off-line. Assertions are also used to detect deviations from "normal" behavior, e.g., to detect if the correct string was read from the network. The information on "normal" behavior is hard-coded into the test applications. To detect OS crashes/hangs, and to check if the previous experiment exited normally or not, the experiment manager reads the entries in the configuration file during boot up.

5.4. Estimating Measures

The only measures that need to be experimentally estimated are the Service Error Permeabilities, $PDS^i_{x.y}$ and $POS^i_{x.z}$. The test applications together with the experiment manager are responsible for detecting failures/propagations in the system. Software assertions generally require well defined specifications. In the case for Windows CE .Net, we consider the help sections shipped with the tools, as well as any other official documentation provided (e.g., msdn library) as the specification of a service. The specification of a service generally includes syntax information, in/out relationships, and information on possible return values for the parameters and functions. For some services, no entry was found in the msdn database. For these, only a search in the system header files found the specification of those services. Unfortunately the header file does not include information on return values etc. except for the C-type used. The header files are included as part of the SDK for the target and accessing them does not violate our black-box approach.

The estimated Service Error Permeabilities $\widehat{PDS}^i_{x.y}$ and $\widehat{POS}^i_{x.z}$ are the ratios of detected errors, at the OS-service level to the number of injected errors for each location. The estimated values of OS Service Error Exposure ($\widehat{E}^j$) and Driver Error Diffusion ($\widehat{D}^j$) can be calculated using the estimated Service Error Permeabilities using Eqs. 3 and 4.

6. Error Propagation Results

Driver	# Services		Test cases	Activated cases
	Imported	Exported		
Serial	50	10	411	43%
Ethernet	42	12	414	55%

Table 3. Overview of the target drivers.

Table 3 overviews the target drivers used in this study. The two drivers have similar number of services in their interfaces to the OS (60 and 54) which translates into a similar number of test cases. The number of test cases depends on the number of services, the number of parameters targeted and the test cases defined for each type. The selection of errors for each type is discussed in Sec. 3.1 and Table 1 & 2. The drivers differ in the number of activated test cases, i.e., when an injected error is in fact executed, where the network driver has a higher activation rate (55% against 43%). The activation rate is simply a measure of how many experiments actually execute the error. The time to execute the experiment depends on the number of experiments performed. Currently, using four separate boards in parallel, the full experiment time is five to six hours including set-up time. Not included is the time to develop driver specific interceptors and the specific error models.

OS Service	Tests	Fail. Class			
		NF	1	2	3
CreateThread	13	6	4	3	0
CreateEventW	6	4	0	2	0
InterruptInitialize	14	3	10	1	0
memcpy	11	7	3	1	0
Sleep	5	4	0	1	0
LeaveCriticalSection	1	0	0	1	0
LocalAlloc	9	4	5	0	0
EnterCriticalSection	1	0	1	0	0
InitializeCriticalSection	1	0	1	0	0
memset	15	14	1	0	0
Cumulative	76	42	25	9	0

Table 4. Serial driver service errors for `cerfio_serial.Dll`.

OS Service	Tests	Fail. Class			
		NF	1	2	3
FreeLibrary	3	1	0	0	2
LoadLibraryW	3	2	0	0	1
NdisAllocateMemory	20	19	0	1	0
VirtualCopy	16	2	14	0	0
KernelIoControl	18	5	13	0	0
VirtualAlloc	18	7	11	0	0
memset	15	6	9	0	0
NdisMSetAttributesEx	16	10	6	0	0
NdisMSetAttributesEx	16	10	6	0	0
NdisMRegisterInterrupt	17	11	6	0	0
RegOpenKeyExW	17	12	5	0	0
NdisOpenConfiguration	3	0	3	0	0
memcpy	11	8	3	0	0
CreateMutexW	5	3	2	0	0
NKDbgPrintfW	3	2	1	0	0
GetProcAddressW	6	5	1	0	0
Cumulative	187	103	80	1	3

Table 5. Ethernet driver service errors for `91C111.Dll`.

Following the error model of Sec. 3.1, Table 4 shows the results of the experiments for the serial port driver `cerfio_serial.Dll` with respect to the OS services used by the driver. We have selected only those services which resulted in failures and indicated the number of error propagation observations, not their location. Table 5 shows the corresponding results for the Ethernet driver `91C111.Dll` which has more services leading to errors compared to `cerfio_serial.Dll`. The rows in the tables are ordered according to the severity of the failures. For `cerfio_serial.Dll` we see that no service leads to a crash of the system. For `91C111.Dll` on the other hand,

FreeLibrary and LoadLibrary are both vulnerable services (Table 5). `91C111.Dll` does not have as many cumulative **Class 2** failures as `cerfio_serial.Dll` (only one compared to nine) which indicates that using a few wrappers would remove all severe error propagation paths (**Class 2** and **Class 3** failures).

Looking from the OS service point of view, Tables 6 and 7 show the OS services tested, together with the results of the experiments, i.e., the OS Service Error Exposure calculated using Equation 3. Alongside, the number of failures for **Class 1**, we also include the number of *No Failure* observations (**NF**). The **Class 2** (*failure and specification violation*) and **Class 3** (*crash/hung*) affect each OS service listed the same way. This effect is specific to the experiments conducted and does not translate into a general statement of the OS behavior. For **Class 2** failures only the driver specific test application was affected. Thus the OS Service Error Exposures are calculated using only **Class 1** failures. From these tables, one can find the services that are more (on a relative scale) exposed to errors propagating in the system. For some services the number of propagated errors is zero, indicating that the function was not affected by any of the errors injected (from **Class 1**). On top of the OS services tested (Table 6) we have also included the correctness assertions, which detect whether the correct information was received from the host computer. In this case *Correctness 1* failed 27 times, which indicates that the first round of testing done in the application failed, where as the second round (*Correctness 2*) did not. This is not surprising given that each error is injected once.

As can be seen in both tables, the results are grouped in two categories. This shows how the results of the experiments "cluster", i.e., an error in one service implies an error in another. This indicates dependencies across services, as well as non-dependencies (or at least indication of weaker dependency). Some of these dependencies are expected, for instance that CreateFile affects ReadFile and WriteFile (Table 6). Some (non-)dependencies are more unexpected, for instance that SetCommState is not affected by CreateFile. For both drivers, only one cluster appears, with 27 cases for 7 services for the serial driver `cerfio_serial.Dll` and 85 cases for three services for the Ethernet driver `91C111.Dll`.

For the experiments in this case study, no OS service (s_i) experienced failure as a result from more than one driver, i.e., the OS Service Error Exposure is equal to the sum of the Service Error Permeabilities for each driver for this experiment. This suggests that there is little correlation between failures in the OS services tested for both drivers.

Table 8 shows the *Driver Error Diffusion* values for the target drivers. The values for the different failure classes are presented separately as they have different failure impacts. A **Class 3** failure naturally has higher impact than a **Class 2** and so on. Table 8 shows that taking error impact into consideration, the network driver has the more severe errors, whereas the serial driver has more **Class 2** failures. Thus these two classes of failures should be the focus of the first round of wrapping. The network driver has overall more failures but mainly of lesser impact, however these are still candidates for wrapping.

OS Service	Failure Class				$\widehat{E^j}$
	NF	1	2	3	
Correctness 1	384	27	9	0	0.066
CreateFile	384	27	9	0	0.066
GetCommState	384	27	9	0	0.066
GetCommTimeouts	384	27	9	0	0.066
SetCommTimeouts	384	27	9	0	0.066
ReadFile	384	27	9	0	0.066
WriteFile	384	27	9	0	0.066
CloseHandle	411	0	9	0	0
Correctness 2	411	0	9	0	0
SetCommState	411	0	9	0	0
strlen	411	0	9	0	0

Table 6. OS Service Error Exposure for `cerfio_serial.Dll`.

OS Service	Failure Class				$\widehat{E^j}$
	NF	1	2	3	
connect	274	85	1	3	0.205
closesocket	274	85	1	3	0.205
shutdown	274	85	1	3	0.205
getaddrinfo	414	0	1	3	0
getnameinfo	414	0	1	3	0
getpeername	414	0	1	3	0
memset	414	0	1	3	0
select	414	0	1	3	0
sendto	414	0	1	3	0
socket	414	0	1	3	0
strcpy	414	0	1	3	0
WSACleanup	414	0	1	3	0
WSAStartup	414	0	1	3	0

Table 7. OS Service Error Exposure for `91C111.Dll`.

Driver	Failure Class Diffusion			Total
	D^k_{C1}	D^k_{C2}	D^k_{C3}	
cerfio_serial.Dll	0.460	0.022	0	0.482
91C111.Dll	0.616	0.002	0.007	0.625

Table 8. Results of injection experiments.

7. Discussion

Identification of Vulnerabilities: The purpose of our proposed profiling methodology is to identify potential vulnerabilities in the system. We target two distinct interfaces; the *OS-application* interface and the *OS-driver* interface. The main results of this study can be summarized as:

- Studying services at the OS-driver interface reveals specific services for one driver (Ethernet driver) to be susceptible to errors leading to severe failures, i.e., FreeLibrary and LoadLibrary (Table 5).
- A set of services at the OS-application interface is susceptible to errors (Tables 6 and 7). A clustering effect is observed which reveals dependencies across services, as well as unexpected lack thereof.
- Comparing Driver Error Diffusion values, we conclude that the network driver (`91C111.Dll`) is more likely to spread errors ($0.625 > 0.482$ in Table 8) than the serial port driver.
- No correlation across drivers was observed, i.e., no "shared" propagation paths between the drivers.

It is important to note that the vulnerabilities identified may not correspond to "bugs" in the OS. We have chosen to use the word "vulnerability" to signify that this is only a potential weakness in the system. *An exposed propagation path only indicates that there are potential vulnerabilities in the system.* Whether this path will actually be used in the operational mode of the system is not assured. To better consider this relation we intend to extend the model to include applications as well.

The Driver Error Diffusion values are calculated using the whole set of experiments. Thus the values depend on the number of experiments, i.e., the higher the number of experiments, the lower the values. If one disregards the number of experiments, the number of errors is to be compared. Assuming that the potential sources for vulnerabilities increase with the number of functions, the first method becomes useful to compare drivers. The Driver Error Diffusion then indicates the likelihood of actual vulnerabilities being present, given the size of the interface. For impact analysis, the number of failures might be of more importance, and here focusing on each error case, starting with **Class 3** is a relevant approach.

Application Profile: The operational profile of an application, i.e., the way an application in the system uses and depends on the OS, has a high impact on the robustness of the system. The subset of services used and their exposure to errors affect the system wide exposure to errors. For this case study four applications were used, that both load the system for the duration of the experiments and measure the propagation of errors. For propagation profiling to be truly useful, real application profiles must be used. The operational profile of real applications, must influence the interpretations of the exposure and diffusion measures. An application profile must consider a) the services used by an application, b) the impact errors in these services have on the application, c) the frequency at which they are used, and c) the relative criticality of the application. When the profiles exist for the applications in the system, they can be composed with the exposure profile to get a system-wide profile, which ranks the applications (and the services they depend upon) according to their susceptibility to propagating errors. The definition and implementation of such application profiling is part of our future work.

Experiments: An important aspect of our methodology is for it to be *repeatable*. Our experiments are repeatable in the sense that no randomness is involved in choosing the error type or location. However, there is a risk that the state of the system is not the same for every injection. We limit this problem by rebooting the system before each injection and we have found no deviation so far for multiple runs of the same experiment set.

Naturally we acknowledge that our current method is not complete. A concern when using any FI technique is the choice of error model. Without a valid error model the conclusions derived might be misleading or even false. We have chosen a simple, but yet realistic, error model for our experiments. This means that the errors we inject are possible in real systems; they are for instance not caught by the compiler. We believe that the errors we use are representative of real errors. However, it is part of our future work to extend and include more error models, to study their respective properties such as exposing coverage (the number of vulnerabilities exposed and their type), cost in terms of implementation effort and the execution time.

Another concern for any FI experiment is to use a minimum of intrusiveness on the target system, in order for the experiments not to be influenced by the setup. For our setup, the Manager and Interceptor run on the target system, see Figure 2. To minimize their impact on the results, we have designed them to be as small as possible. The number of messages being sent is kept to the bare minimum. Most of the communication takes place before the system has booted up, i.e, before an error is injected.

Wrapper Placement & Design: For effective use of wrappers, the actual wrapper composition is of importance. The measures presented above also help for this purpose. Careful inspection of the failure of LoadLibrary (Table 5) reveals that it occurs during boot-up of the system. *This means that no application, however well designed, can cope with this error.* This suggests that a wrapper for this error must be placed at the OS-driver interface level, filtering the parameters, to not allow this type of error to pass. With code access this information can be used to verify that this vulnerability is not used/activated from a driver. For

some other failures of lesser impact, like CreateFile in Table 6, wrappers can be defined at the OS-application level, implementing for instance a restart functionality handling transient errors. The development of wrappers and study of their usefulness for enhancing the robustness of the system is an important part of our future work.

Future Work: Our ongoing and future work includes extending the current prototype in several directions. We intend to use a larger set of targets, both drivers and OS's (e.g., Windows CE .Net 5.0, Windows XP and Linux). We also intend to investigate error models of varied level of detail, to analyze the effectiveness of different error models.

8. Conclusions

Overall, the contributions of this paper lie in developing measures to aid quantification of OS error flow. The relevance of the measures and associated methodology is demonstrated by the OS case study where the experiments exposed a number of potential vulnerabilities. This information can be used either to place wrappers in the system or as feedback to OS or driver designers. We believe that this shows the utility of the proposed measures and methodology. Studying exposure and impact of errors tells the designer not only where many errors may pass but also more importantly where occurring errors may have severe consequences. Without this profiling, this information would not have been available. As demonstrated, the potential vulnerabilities can lead to system failure, no matter how the applications on the system are designed (and verified). We also find it significant that the observed vulnerabilities were identified using limited black-box information.

Acknowledgments: We express our appreciation for help and insights from Martin Hiller, Falk Fraikin, the DEEDS group and the funding support from Microsoft Research.

References

[1] A. Albinet *et al*. Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel. *Proc. of DSN*, pp. 807–816, 2004.

[2] J. Durães, H. Madeira. Multidimensional Characterization of the Impact of Faulty Drivers on the OS Behavior. *IEICE Trans.*, E86-D(12):2563–2570, Dec. 2003.

[3] J. Arlat *et al*. Dependability of COTS Microkernel-Based Systems. *IEEE Trans. on Computers*, 51(2):138–163, Feb. 2002.

[4] T. Ball, S. Rajamani. The SLAM project: Debugging System Software via Static Analysis. *Proc. of POPL*, pp. 1–3, 2002.

[5] A. Chou *et al*. An Empirical Study of Operating System Errors. *Proc. of SOSP*, pp. 73–88, 2001.

[6] J. DeVale, P. Koopman. Performance Evaluation of Exception Handling in I/O Libraries. *Proc. of DSN*, pp. 519–524, 2001.

[7] J.-C. Fabre *et al*. Building Dependable COTS Microkernel-Based Systems Using Mafalda. *Proc. of PRDC*, pp. 85–92, 2000.

[8] C. Fetzer, Z. Xiao. An Automated Approach to Increasing the Robustness of C Libraries. *Proc. of DSN*, pp. 155–164, 2002.

[9] T. Fraser *et al*. Hardening COTS SW With Generic SW Wrappers. *Proc. of OASIS*, pp. 399–413, 2003.

[10] W. Gu *et al*. Characterization of Linux Kernel Behavior Under Errors. *Proc. of DSN*, pp. 459 – 468, 2003.

[11] M. Hiller, A. Jhumka, N. Suri. PROPANE: An Environment for Examining the Propagation of Errors in Software. *Proc. of ISSTA*, pp. 81–85, 2002.

[12] M. Hiller, A. Jhumka, N. Suri. EPIC: Profiling the Propagation and Effect of Data Errors in Software. *IEEE Trans. on Computers*, 53(5):512–530, May 2004.

[13] A. Johansson *et al*. On Enhancing the Robustness of Commercial OS's. *Proc. of ISAS*, pp. 174–185, 2004.

[14] P. Koopman, J. DeVale. Comparing the Robustness of POSIX OS's. *Proc. of FTCS*, pp. 30–37, 1999.

[15] T. Mitchem *et al*. Linux Kernel Loadable Wrappers. *Proc. of DARPA Information Survivability Conf.*, vol. 2, pp. 296–307, 2000.

[16] B. Murphy, B. Levidow. Windows 2000 Dependability. *Proc. of the Workshop on Dependable Networks and OS*, pp. D20–28, 2000.

[17] J. Pan *et al*. Robustness Testing and Hardening of CORBA Orb Implementations. *Proc. of DSN*, pp. 141–150, 2001.

[18] L. Réveillère, G. Muller. Improving Driver Robustness: an Evaluation of the Devil Approach. *Proc. of DSN*, pp. 131–140, 2001.

[19] M. M. Swift *et al*. Improving the Reliability of Commodity OS's. *Proc. of SOSP*, pp. 207–222, 2003.

[20] T. Tsai, N. Singh. Reliability Testing of Applications on Windows NT. *Proc. of DSN*, pp. 427–436, 2000.