On the Selection of Error Model(s) For OS Robustness Evaluation*

Andréas Johansson, Neeraj Suri
Dept. of CS, TU-Darmstadt, Germany
{aja,suri}@informatik.tu-darmstadt.de

Brendan Murphy
Microsoft Research, Cambridge, UK
bmurphy@microsoft.com

Abstract

The choice of error model used for robustness evaluation of Operating Systems (OSs) influences the evaluation run time, implementation complexity, as well as the evaluation precision. In order to find an "effective" error model for OS evaluation, this paper systematically compares the relative effectiveness of three prominent error models, namely bit-flips, data type errors and fuzzing errors using fault injection at the interface between device drivers OS. Bit-flips come with higher costs (time) than the other models, but allow for more detailed results. Fuzzing is cheaper to implement but is found to be less precise. A composite error model is presented where the low cost of fuzzing is combined with the higher level of details of bit-flips, resulting in high precision with moderate setup and execution costs.

1. Introduction

This paper focuses on ascertaining the robustness of OSs to errors in device drivers. While multiple OS robustness studies using fault-injection have been reported, for instance [2, 9, 13, 15], in most cases, the results are applicable mainly for the specific underlying error model. The choice of *error model* and *location* of injection directly influences the accuracy and usefulness of the obtained results.

The relative effectiveness of three distinct error models at the OS-Driver interface is compared. The OS-Driver interface was chosen as it represents an interface shared by all drivers, it is reasonably well documented and supplies the level of access needed for this type of studies. The error models are compared for efficiency (cost and coverage), implementation complexity and execution time requirements and a new *composite error model* is presented. Consequently, the paper proposes guidelines on selecting the appropriate error model for OS robustness evaluation. The results of such an evaluation are useful both in system design, i.e., where the OS and the drivers are integrated as part of a larger system, for finding hot-spots in the system warranting refinements, and component evaluation for comparing the suitability of certain components in a system. The chosen approach makes possible a comparative study of the influence of drivers on system robustness, without requiring source code access. Experimental quantitative approaches, such as this one, complement analytical approaches (like [4]) and provide easy means for quantifying dynamic behavior of the system under study.

The chosen error models span: **a)** *bit-flips* (BF), where a bit in a data word is flipped, from 0 to 1, or vice versa. BF were used, for instance, in [9] where the robustness of the Linux kernel was evaluated. **b)** *Data type-based corruption* (DT), where the value of a parameter in a call to a function is changed, according to its data type, for instance to boundary values. This technique was used in [1] and [13]. In [3, 11] both BF and DT errors are used. **c)** *Fuzzing* (FZ), which assigns a randomly chosen value to the parameter in a function call. FZ has previously been used, for instance, in [17, 19].

Paper Contributions & Structure: Selecting an "effective" error model to use in a particular setting is not straightforward and guidelines are of value for both OS designers and evaluators. Building on previous experiences on OS robustness evaluation [13], this paper represents a step towards providing such a guideline. Using a case study based on Windows CE .NET and three different drivers (serial, network and storage card drivers) the paper specifically provides two distinct contributions, namely **i)** a comparative study of error model effectiveness in terms of coverage and cost, and **ii)** it establishes the effectiveness of using a composite error model for OS robustness evaluation.

The paper is presented detailing four main blocks:

Prerequisites: Defining the system model [Section 2]; background information on the studied error models [Section 3]; a presentation of the evaluation criteria for the error models, i.e., error propagation, failure mode analysis and execution time [Section 4].

Implementation: Presentation of the target system and the experimental technique used [Section 5].

Results: Presentation of the results from fault-injection

*Research supported in part by EC DECOS, ReSIST and Microsoft

experiments [Section 6] with interpretations [Section 7].

Composite Model: Definition and results for the *composite error model* [Section 8]; discussion and summary of the main findings [Section 9].

2. System Model

Similar to [1, 2] we use a four-layered model of the OS: Application, OS, Driver and Hardware layer. This model applies to most common monolithic OSs, such as Windows, UNIX and Linux. The *OS-Driver* interface (Figure 1) is our target of study.

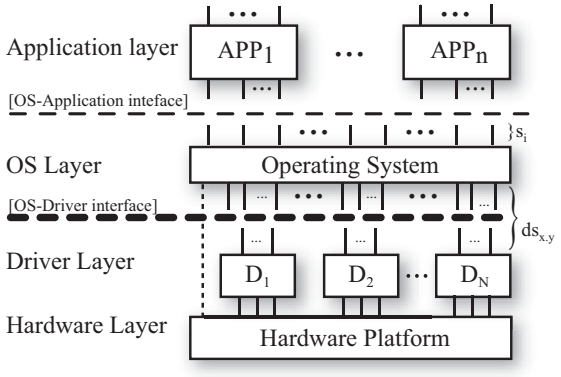

Figure 1. System model

A system containing N drivers ($D_1 \dots D_y \dots D_N$) is considered. Each driver exports and imports a set of services $ds_{x.y}$ where $x.y$ is x^{th} service of driver D_y . The effects of errors are observed at the OS-Application interface by manually instrumenting a number of benchmark applications with assertions. For each system service s_i used, we study its behavior to detect deviations from the golden run.

Only the interface specifications (OS-Driver and OS-Application) are required, but no source code, neither for the drivers, nor for the OS itself. However, access is needed to the source code of the benchmark applications, for instrumenting them with assertions. The availability of interface specifications is a basic requirement for any OS open for extensions by new types of drivers/applications.

3. The Spread of Error Models

The three error models are chosen based on their appropriateness as reported in multiple previous studies and by their relation to real faults, such as those defined by Orthogonal Defect Classification (ODC) [5]. ODC is a method of classifying defects into orthogonal classes, enabling process feedback and control. As it mainly focuses on in-process defects, it uses source code as an intrinsic basis for classification. Our decision not to require access to source code implies that we cannot directly use the ODC classification. However, an attempt is made to classify the error models depending on the defect classes they belong to. The *interface* class of ODC is a potential source for each of the models, as it deals with external interaction, such as drivers. The error models were also chosen as they represent a class of errors that previously have been reported as difficult to detect and recover from [20].

Errors are injected in service parameters. We do single injections, i.e., we do not simultaneously inject in multiple parameters. Injection is performed by intercepting calls and manipulating data at runtime.

For all three error models we use a transient error occurrence and duration model, i.e., the error appears once and then disappears. Errors are injected *on first occurrence*. This implies that the error is injected the first time a service is invoked. Previous studies have suggested that the impact of the time of injection is small [21]; however this remains to be comprehensively established.

All three error models handle pointers and structures as special cases, and inject errors in the target of a pointer and the members of a structure, when possible. Only if a pointer or structure member is already set to `NULL` an injection is not performed.

3.1. Bit-Flip Error Model

Bit-flips is an extensively used error model deriving its origins from transient hardware defects. Its ease of use and implementation has made it a candidate to also be used to simulate software (SW) defects. As it changes the value of a parameter (by manipulating one of the bits) it belongs to the *Assignment* class of ODC defects.

In the BF model, each parameter is treated as an integer (typically a 32-bit word). The model's greatest advantage is also its greatest disadvantage, namely simplicity. While it makes it very easy to implement, it suffers in expressiveness, with respect to abstract data types (strings etc.).

Each injection case flips one bit, thus resulting in 32 injection cases per parameter. However, some parameters are 16 (or 8) bit wide only, and we therefore restrict these parameters to use fewer injections.

3.2. Data Type Error Model

Data type errors are chosen depending on the data type of the targeted parameter. As the targeted interface is defined using the C language, the data types considered are all C-style. This excludes high level abstract data types supported in other high-level languages, such as classes in C++.

We follow established testing practice by choosing for each type a set of predefined (no randomness) test values, offset values and boundary values. The offset values allow us to modify the previous value (present in the intercepted call), such as adding a number to it. The number of injections defined is typically less than ten, allowing this error model to incur fewer injections (on average) than the BF model. However, there is no such inherent property and it depends on the number of cases defined for each type. This model also belongs to the *Assignment* class of ODC, though it could also be a *Checking* defect, resulting from a failing or missing check of a data value.

The number of data types for which injection cases need to be defined depends only on the data types actually used[7, 15]. Many services use the same data types for their parameters, making the approach scalable. The same observation was made in the Ballista project [15] regarding the POSIX API. In total, the three drivers targeted in this paper result in injections for 22 data types being defined. An overview is shown in Table 1. For structures (`struct`) the number of cases depends on the members (marked with * in the table). DT errors also treat pointers as special data type and reserves one injection case for the pointer, namely setting it to `NULL`. Wrong use of NULL-pointers is a common programming mistake. To further illustrate how DT errors are defined, Table 2 shows the cases for the type `int`.

Data type	C-Type	#Cases
Integers	`int`	7
	`unsigned int`	5
	`long`	7
	`unsigned long`	5
	`short`	7
	`unsigned short`	5
	`LARGE_INTEGER`	7
Misc	`* void`	3
	`HKEY`	6
	`struct {...}`	*
	Strings	4
Characters	`char`	7
	`unsigned char`	5
	`wchar_t`	5
Boolean	`bool`	1
Enums	multiple cases	#identifiers

Table 1. Overview of the data types used.

3.3. Fuzzing Error Model

Fuzzing a parameter implies assigning it a pseudo-random value from the set of legal values for the type.

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`.

Therefore, the result of the injection may differ across experiments, resulting in the need for multiple experiments to obtain statistically valid conclusions. A specific discussion on the number of injections needed appears in Section 6.4.

Whereas BF and DT use the parameter values, FZ differs in that it ignors these values. FZ belongs to either the *Assignment* or *Checking* classes in ODC. This model was considered in [17, 19, 10].

The pseudo-random values are chosen using the `rand()` C-runtime function. The last value produced in a round is stored in persistent storage and is used as the seed for the next round, thus ensuring that different random values are used every time.

3.4. Other Key Contemporary Models

The work in [1, 2, 3, 8, 12, 14] explored the use of various error models and injection techniques for OS robustness evaluation and benchmarking. For instance, [12] compares errors similar to the ones considered here, but injected at different levels of the Linux kernel. In [2] a mutation error model is used, in which the code segments of drivers are targeted. Further, in [18] the authors observe that effects of errors at the interface, though being useful for robustness evaluation, do not represent defects in code. As we do not inject errors at the code level we can neither verify, nor falsify this observation. We believe that our systematic comparison of error models, is a useful contribution to the community as this comparative aspect at the OS-Driver interface has not been treated in depth before.

The chosen models, even though not complete, still represent a large operational spectrum. [6] studies defects in two large OSs and almost 50% of the found defects belong to the ODC classes represented in this paper.

4. Comparative Evaluation Criteria

The chosen error models are studied based on a diverse set of commonly used evaluation criteria, namely i) error propagation (Diffusion), ii) error impact (failure class), iii)

implementation complexity, and iv) execution time. Each criteria is elaborated in the following subsections.

4.1. Error Propagation Criteria (Diffusion)

In this paper, the focus of error propagation is on *Driver Error Diffusion*[1] [13]. Diffusion is defined as the degree to which a driver can spread errors in the system. It allows drivers to be compared and ranked making it possible for a system evaluator/designer to make a judgement on where to expend more resources in terms of testing/verification and quality improvement.

Diffusion considers the propagation paths from a driver, through the OS to user applications. It is the sum of the conditional probabilities for an error to propagate, given that an error exists. Diffusion is itself not a probability but a metric of sensitivity to input errors, which can be exactly estimated by code analysis, or approximated experimentally.

$$D^x = \sum_{s_i} \sum_{ds_{x.y}} PDS^i_{x.y} \quad (1)$$

In Equation 1, $PDS^i_{x.j}$ represents the conditional probability that errors in the driver's use of OS services ($ds_{x.j}$) lead to failure (see also Figure 1).

Diffusion can be used to compare the suitability of drivers for a particular system based on to their *potential* for spreading errors. A higher diffusion value implies that a driver is more liable to spread errors. However, note that drivers are not tested per se. Consequently, we stress that the intent is *not* to give absolute values for error sensitivity, but to obtain relative rankings. A detailed discussion on diffusion and error propagation metrics is found in [13].

It is important to note that an error can propagate in the system and still remain latent (i.e., not lead to failure) without immediate detection. As the triggers for dormant faults is not known, we take an optimistic approach and consider a failure free run of the test applications to imply that the likelihood of a dormant fault is very low.

4.2. Error Impact Criteria (Failure Class)

To determine and distinguish the impact of a propagated error we use failure mode analysis. A set of four modes of increasing severity is defined including the non-propagating one representing normal or failure-free behavior.

Class NF: No discernible violation observed as outcome of an experiment, i.e., *No Failure* class. Three distinguishable explanations account for an injection resulting in this class, namely a) the error location was not activated in this execution; b) the error was injected, but was masked by the system; or c) the fault is dormant. Section 5.3 describes how case a) can be avoided.

Class 1: The error propagated to the benchmark applications, but still satisfied the OS service specification. Examples of class 1 outcomes include unsuccessful attempts to use services where the error code returned is in the set of valid codes for this call. The propagation of data errors also fall into this category.

Class 2: This failure mode captures behaviors violating the specification of the service. It could be an unforseen hang or crash of the *application* due to the error or an incorrect error code being returned. An application is considered hung after 40 seconds of non-responsiveness, exceeding 100% of normal execution time in a golden run experiment. Note that the rest of the system is still functioning after the failure.

Class 3: The OS becomes irresponsive due to a crash or a hang. No further progress is possible.

The failure modes give rise to a partitioning of the experiment outcomes. Similar to for instance [16], when an experiment could be placed in multiple classes, e.g., when it first gives an application error code (class 1) and then the system crashes (class 3) the more severe class is assigned.

4.3. Implementation Complexity Criteria

The complexity of implementing the FI campaign is a subjective and qualitative estimation of the effort needed to implement the three different error models. A discussion on the implementation complexity appears in Section 7.

4.4. Experiment Execution Time Criteria

Experiment execution time significantly influences the usability of the chosen approach. We therefore track the execution time of all experiments. Failures requiring manual intervention (Class 3) are assigned 200 seconds. This is the standard timeout used by the system to detect if no progress is made and a reboot is required. It is set to be sufficiently large to capture any delays incurred by an error, i.e., to detect that the system is hung and is not just delayed.

5. Target Setup

The conducted experiments use Windows CE .Net 4.2. The hardware is a development board, using the Intel XScale PXA255 platform, with 64 MB RAM and 32 MB ROM (flash). The board is connected using serial and Ethernet connections. The board also provides a Compact Flash (CF) slot. We have used this setup as its structure is very similar to most other OSs and hardware. It is also small in size making it easy to work with and control.

[1] We use the shorter term *Diffusion* in the rest of the text.

From a SW perspective, the system comprises two main components, namely the *Interceptor* and the *Experiment Manager* (Figure 2). The Interceptor intercepts all calls to or from a specific driver, and can then inject errors on request. It interacts with the Experiment Manager, receiving commands and sending the results back in form of log messages. The Experiment Manager is responsible for setting up the needed infrastructure, sending injection commands to the Interceptor, transmitting log messages to the host computer and for monitoring the outcome of the experiments. The Experiment Manager starts the test applications and monitors their behavior, receives reports of triggered assertions, and passes them on as log messages. It is also responsible for restarting the machine after each experiment.

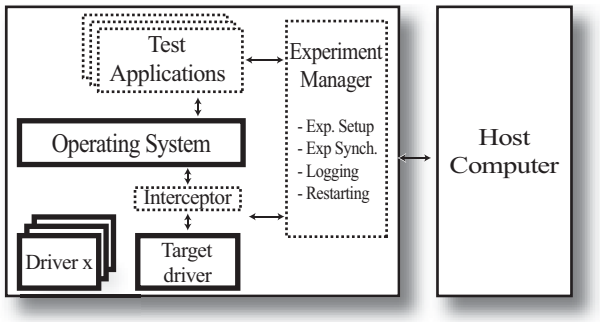

Figure 2. Experiment setup

Information on the selected experiments to perform is stored in a file in persistent storage (flash memory) on the target computer. The file is created the first time the system boots up. The injection is configured using the registry and the Interceptor automatically generates all test cases for the selected set of targeted services and the chosen error model.

Each experiment run (combination of error model and driver) uses a newly built OS image. Each experiment starts with a cold reboot where the OS image is read from ROM into RAM, ensuring that each injection is performed using a fresh, uncorrupted, OS image. Persistence between injections is limited to the error configuration file, and for FZ errors to the seed to the random number generator. Logs are stored on the host computer.

5.1. Targeted Drivers

For comparison, we target three different drivers for our experiments. Table 3 shows the number of services targeted and the total number of injection cases for the three targeted error models. The number of services reported includes both exported services, used by the OS, and imported OS services that the driver uses.

The drivers were chosen to represent three common, yet different, functional classes of drivers. The serial driver (`cerfio_serial`) implements the common RS232 serial interface, a well established and commonly used interface. The Ethernet driver (`91C111`) represents network interface drivers. The CF driver (`atadisk`) represents a different class of interfaces altogether, namely filesystem drivers.

Driver	#Services	# Injection cases		
		BF	DT	FZ
cerfio_serial	60	2362	397	1410
91C111	54	1722	255	1050
atadisk	47	1658	294	1035

Table 3. Overview of the target drivers.

5.2. Benchmark Applications

The benchmark applications consist of five different processes. One application uses the driver that is currently targeted, thus there are test applications testing serial and Ethernet communication (with the host computer) as well as testing multiple reads and writes to the CF card. The general benchmark applications target a variety of general OS services, such as process creation and synchronization, file system operations and memory allocation/manipulation.

The applications are chosen to activate the system in a varied manner and to drive the experiments, i.e., function as workload for the targeted driver. For a system designer, the set of applications to be used in the target product may be known, and if so they should be used to drive the experiments. If not, then benchmark applications form the best choice, as they usually target many common features.

For each injection, all applications are used and their results tracked for deviations using assertions. Each application is written specifically for testing purposes, therefore its expected behavior is known *a priori*. This makes it possible to manually track the used services and add assertions.

5.3. System Pre-Profiling

To expedite the injection process, the system is first profiled to remove injections that will not lead to an error being injected. This is achieved by first generating all injections for a driver and then running the benchmark applications while keeping track of which services are being used. After successful execution of the benchmark applications, the list of injections is reduced to include only services actually called during profiling run. This typically reduces the number of test cases by half or more. The number of injections greatly influences the time required to execute the experiments. The more unnecessary cases identified, the more time is saved. Thus, the most time (in absolute numbers) is saved for the BF error model, since it requires the most injection cases in this study.

6. Experimental Results

A range of experiments were conducted for the three drivers. The next sections present the comparative results for the selected criteria. Due to the nature of the error models studied, BF use a significantly larger set of injection cases. Consequently, the time taken to perform the experiments is also significantly longer. The discussions in the following sections focus on class 3 failures, as these are relevant for robustness evaluation. Appropriate references to the other classes are clearly indicated. For FZ we report values for fifteen injections per parameter. Section 6.4 details a discussion on the number of injections needed.

Driver	BF	DT	FZ
cerfio_serial	1.05	1.50	1.56
91C111	0.98	0.73	0.69
atadisk	1.86	0.63	0.29

Table 4. Driver Diffusion for class 3 failures.

6.1. Comparing Drivers

Driver Diffusion (as defined in Section 4.1) is used to compare the drivers. The probability $PDS^{i}_{x.j}$ is approximated as the ratio of failures to the number of injections. Table 4 summarizes the results showing that DT and FZ identify the serial driver to be the most vulnerable driver (higher Diffusion value), whereas BF pin-points `atadisk` to be the most vulnerable.

Table 6 details the results for each driver and error type. Overall the class 3 ratio is below 5%, indicating that the OS is indeed able to handle most introduced perturbations. Furthermore, we conclude that the error model does not significantly impact the ratios for the `91C111` and `cerfio_serial` drivers. For class 3 failures the percentage of injections (last column) varies between 3.22% and 3.97% for the serial driver and 2.35% and 4.24% for `91C111` driver. For `atadisk` the differences are larger, but still within 1.26% and 3.98% with BF identifying it as more vulnerable. The results show slight differences between the drivers as well as between the error models.

While Diffusion values in Table 4 for BF indicate `atadisk` to have highest diffusion, the experimental results from Table 6 show that `91C111` has a higher ratio of class 3 failures. This is due to Diffusion being a "sum of probabilities". Diffusion shows that `atadisk` has more services with higher propagation probability than `91C111`.

For class 2 failures there are some distinct differences between the drivers. `91C111` and `atadisk` drivers have considerably fewer class 2 failures. This is due to differences in how the drivers function, i.e., blocking vs. non-blocking. Failed blocking services are more likely to cause hangs of the system, i.e., class 2 failures. This suggests that there is, as expected, a difference between the tested drivers, which is exactly what the Diffusion metric captures. For class 1 failures, we notice the same difference with the serial driver having fewer cases due to its blocking nature.

Overall, many injections, for all drivers and all error models, end up in the NF category, i.e., no observable deviation from the expected behavior could be seen. This is in line with several previous studies, e.g., [2], [9] and [12]. It is important to note that all cases reporting the errors were in fact activated, since the pre-profiling eliminated the not used services *a priori*. Outcomes in the NF category are either masked by the system, for instance by not being used or overwritten; or handled by built-in error detection/correction mechanisms checking incoming parameter values for correctness. Another explanation could be that the fault is dormant in the system and has not yet propagated to the OS-Application interface.

Driver	Error Model	Execution Time	
		hours	minutes
cerio_serial	BF	38	14
	DT	5	15
	FZ	20	44
91C111	BF	17	20
	DT	1	56
	FZ	7	48
atadisk	BF	20	51
	DT	2	56
	FZ	11	55

Table 5. Experiment execution times.

6.2. Execution Time

Table 5 details the execution time for each experiment run. The BF model with the most injections, has the longest execution time. However, the execution time also depends on the outcome of the experiments (class 2 and 3 take longer time as they typically require timeouts to be triggered) and the nature of the test applications. There are also slight variations in the boot-up time of the target system. The serial driver and the `atadisk` driver both take longer time when failing, which also influences the execution time.

6.3. Comparing Error Models

Table 7 depicts services incurring class 3 failures. It shows the number of failures for each service/error model. BF clearly outperforms the other error models in terms of

Driver	Error Model	No Failure	%	Class 1	%	Class 2	%	Class 3	%
cerfio_serial	BF	1771	74.98%	209	8.85%	306	12.96%	76	**3.22%**
	DT	264	66.50%	65	16.37%	53	13.35%	15	**3.78%**
	FZ	931	66.03%	218	15.46%	205	14.54%	56	**3.97%**
91111C	BF	1166	67.71%	482	27.99%	1	0.06%	73	**4.24%**
	DT	181	70.98%	67	26.27%	1	0.39%	6	**2.35%**
	FZ	670	63.81%	350	33.33%	1	0.10%	29	**2.76%**
atadisk	BF	1246	75.15%	343	20.69%	3	0.18%	66	**3.98%**
	DT	191	64.97%	98	33.33%	1	0.34%	4	**1.36%**
	FZ	531	51.30 %	483	46.67%	7	0.67%	13	**1.26%**

Table 6. The number of experiments is shown for each driver, error model and failure class.

Service	BF	DT	FZ
SERIAL_OPEN	1	x	x
CreateThread	4	1	x
DisableThreadLibraryCalls	6		x
FreeLibrary	4		1
InitializeCriticalSection			1
LoadLibraryW	2	2	
LocalAlloc	2	4	x
MapPtrToProcess	2	1	
memcpy	77	3	32
memset	74	3	29
MmMapIoSpace	11	9	26
NDISInitializeWrapper	1	x	
NDISMSetAttributesEx	4		
NDISMSynchronizeWithInterrupt	7	1	2
QueryPerformanceCounter	2		
SetProcPermissions	1	1	7
wcscpy	6		
wcslen	11		

Table 7. Services identified by class 3 failures. "x" indicates class 2 service failures.

identifying vulnerable services, a key aspect for robustness enhancing efforts, such as using wrappers. In order to increase the identification coverage for the DT and FZ models we have indicated in the table which services exhibit class 2 failures, which increases the coverage slightly. Still there are four services identified only by BF. The DT model performs slightly better than FZ, but in two cases FZ identifies a service which DT does not. It is also important to note that one service is only found by FZ.

6.4. The Number of Injections for Fuzzing

A crucial question regarding the FZ model is how many injection cases are needed. Figure 3 shows how Diffusion changes with increasing number of injections. The X-axis shows the number of injection and the Y-axis shows the diffusion values using x injections. Diffusion stabilizes after roughly ten injections. We have injected fifteen cases for all three drivers and all of these are included in Tables 3-7.

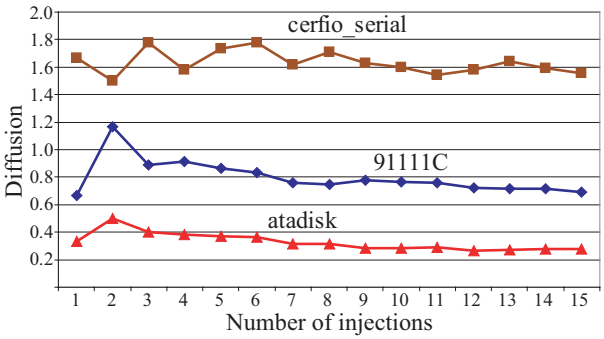

Figure 3. Stability of Diffusion for the FZ model wrt. the number of injections.

7. Interpretation & Discussion

Error Models & Error Severity: The first major finding is that the BF model causes more severe failures than the other models. Table 6 shows that BF finds by far the most class 3 failures of all error models. However, the number of injections used is also high, which comes with a cost in terms of execution time (Table 5). Therefore, when time is crucial, other error models may be considered. In terms of number of injections and execution time the DT error model performs the best, with FZ in the middle.

Comparing BF & FZ: The second major finding found in Table 7 is that BF identifies more services having class 3 failures than any of the other two models. FZ, on the other hand, identifies one service (`InitializeCriticalSection`) that none of the other two identified. So the question is: why are some of the services not identified by FZ? FZ chooses a new random value to be used in an injection, whereas the BF and DT error models modify the existing value. For services that have some basic level of checking of incoming parameters values which are well off the expected, as a random value is likely to be, are easy to find. However, values that are close to the expected value (as when only one bit is changed) are more difficult to find, and may therefore slip through and cause a failure of the system. This happens for instance when targeting different types of handles (to modules, libraries, memory areas etc.) and when targeting typical control values, such as bit-mask flags. Interestingly, FZ is traditionally regarded as not being very effective. However, our results are displaying its surprising effectiveness based on the number of injections used, especially when the intent is to identify drivers using the Diffusion metric.

Model Choice: Table 4 shows a difference in the result between the models. BF identifies `atadisk` as the most vulnerable driver and the other models identify the serial driver as most vulnerable. As described in Section 3 there are significant differences between the models. Ultimately, the choice of error model is influenced by many factors. Section 8 discusses some of the trade-offs that must be made, with respect to time, implementation complexity and more importantly the goals of the evaluation.

Implementation Complexity/Cost: Table 5 shows that BF and FZ are clearly more expensive in terms of execution time compared to DT. However, a major drawback with the DT error model is the cost for implementation. Since for every service in the interface the type of each parameter needs to be kept, it requires implementing support for this. BF and FZ on the other hand do not have this requirement, making their implementation considerably cheaper. The higher cost for the DT model could potentially be reduced by use of automatic parsing tools and/or reflection-capable programming languages. As this cost is a one-time cost for each driver, the cost might be acceptable if the experiments are to be repeated in a regression testing fashion.

Experiment Time: A factor influencing the experiment time is the degree of operator involvement. The operator is required to specify which experiment to run and for which driver. The time to do this is the same for all models. Some experiments force the system into a state where it cannot itself reboot, requiring the operator to manually reboot the system. 21.3% of the class 3 failures result in the system being left in a state where it cannot itself reboot. A consequence is that without external reboot mechanisms the experiment is delayed until the operator takes action, which can prolong the execution time substantially. We have not included this time in the total execution time for the experiments, since we cannot make any assumption on the presence of the operator. Each manual reboot is given a generic penalty of 200 seconds which is the timeout used to detect a hung system which automatically restarts.

Class 2/3 & Bugs: A question one might ask after seeing these results is whether the fact that class 2 and 3 failures are observed indicate that the system contains bugs? The answer is: not necessarily. It has until now been common practice to use a "gentlemen's agreement" between the OS and the drivers. This is mostly due to the fact that the costs of checking each and every call to the kernel would be too high for most systems. So the fact that the system crashes might not be due to a bug in the traditional sense. It is however from a robustness point of view a "vulnerability". All targeted drivers are deployed drivers, i.e., their producer has tested them to some extent and they do work well in our system when no errors are injected.

8. Developing the Composite Error Model

The results from Section 6 provides two major findings: a) BF pinpoints the most services for class 3 failures and b) FZ gives similar Diffusion results to BF at markedly a lower cost, but does not find as many services. This section explores these differences and use them to combine the two error models into a *composite error model* (CM) that identifies as many vulnerable services as BF but with fewer injections. We focus on the class 3 failures, as these are of highest interest when conducting robustness evaluation. The composite model combines BF & FZ by not utilizing the full bit space of the BF model. Thus a key step is to identify the subset of the BF model bits to combine with the FZ model. The following two subsections establish this basis to result in the selected composite model (CM).

8.1. CM Setup: Bit Failure Distribution

The relative inefficiency of BF in terms of execution time is a result of the number of injections. As noted in Section 3.1 the 32 bits available for flipping are not used uniformly. Figure 4 indicates that there are more services only sensitive to flips in the least significant bits than in the most significant. The bits below 10 (to the right in the figure) clearly cause failures in more services. Figure 5 shows the cumulative number of services identified, starting at bit 0 (from right to left). The figure shows that after bit 9 only bit 31 identifies a service not previously identified. Thus, the services having (class 3) failures for bits 10-31 also have failures for bits 0-9 (with `InitializeCriticalSection`

being the only exception for bit 31). Thus, focus of the injections should be put on these bits.

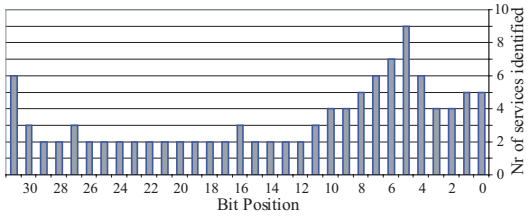

Figure 4. The number of services identified by Class 3 failures by the BF model.

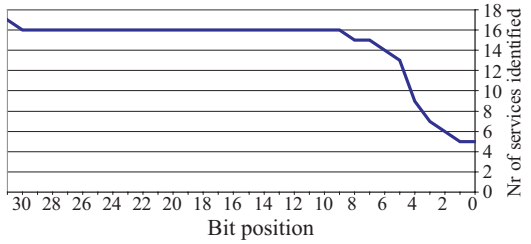

Figure 5. Moving from bit 0 and upwards the number of services increases until bit 10.

The observations made in Figures 4 and 5 allow us to modify the BF model to use fewer injections (only bits 0-9 and 31). Using only these bits reduces the number of test cases for BF by 62% (to 2164 in total), while still identifying all services. It is important to note that these results are system specific and the result of our experimental setup. Further research on methods for extracting such profiles at minimum cost is needed.

8.2. Distinguishing Control vs Data

A study of the parameters targeted for the services identified by BF, but not by FZ, reveals a prevailing trend: the parameters used are all control values, like pointers to data or handles to files, modules, functions etc. It is reasonable that these parameters are more sensitive to changes in the least significant bits (LSB) than to changes in the most significant bits (MSB). E.g., for a pointer that points to data within the process' memory region changes in the LSB will yield a new pointer within the region (but to possibly non-valid data) whereas changes of the MSB will yield a non-valid pointer which is easily detectable on modern hardware. Flipping a bit in the MSB is more likely to yield a non-valid pointer than changes to the LSB, and consequently we see a difference in the failure distribution. FZ, using random

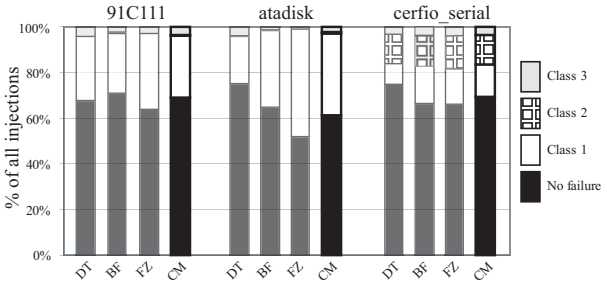

Figure 6. Failure mode distribution for CM compared to BF, DT and FZ.

Driver	Diffusion
cerfio_serial	2.9
91C111	2.0
atadisk	1.9

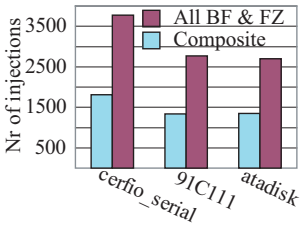

Figure 7. Diffusion and comparison of the number of injections with combined BF & FZ.

32-bit values, is also less likely to produce a valid pointer. However, since the value is random, it can also trigger failures not found by more structured injections (which is limited to changes to existing values or a small subset of special values), as shown by the fact that only FZ identifies a class 3 failure for service `InitializeCriticalSection`.

8.3. Composite Model & Effectiveness

The results in the previous sections have established the need for using multiple error models. Therefore, we recommend using **both** BF and FZ when resources are plentiful. When not, we propose to use *a composite model where the least significant bits (together with the most significant one) are targeted with BF, alongside a series of FZ experiments*. Section 6.4 established that ten FZ injections are sufficient for stabilizing the Diffusion metric. It is reasonable that more injections will increase the probability of finding "rare" cases (such as `InitializeCriticalSection`) but at the cost of increased number of injections.

A composite model, using only bits 0-9 and 31, together with ten FZ injections, identifies the same set of class 3 vulnerable services as the full set of BF and FZ injections. An overview of the results is shown in Figure 7. The table presents the Diffusion results for the composite model and it also shows how the composite model saves injection cases compared to performing all BF and fifteen FZ cases.

The Diffusion results are very similar to those presented for the individual error models in Table 4, identifying the serial driver as being the most vulnerable one, followed by the `91C111` and `atadisk`.

Figure 6 shows the results of CM alongside the other models, and it clearly show a similar trend as for the original error models, with a significant portion of the experiments ending up in the NF class. The number of injection cases is in the same range as those for BF, but higher than that for FZ with fifteen cases. Compared to performing both (BF & FZ) it *corresponds to performing only 48.7 % of the injections*, a significant reduction. Many other factors influences the actual execution time of the experiments. Assuming the execution time being proportional to the number of injections the CM gives a saving of up to 60 hours experimentation time for the combined BF & FZ.

9. Conclusions

This paper reports on extensive fault injection experiments carried out for three commonly used error models: bit-flip, data type and fuzzing. The results show bit-flips as the most acute one, but with the highest implementation cost. Based on these findings a new composite error model has been defined that compared to extensive bit-flip and fuzzing experiments achieves **a)** comparable error propagation results, and **b)** identifies the same set of vulnerable services. This is achieved using less than half the number of injections.

As this paper reports on experimental techniques the results must be viewed in this specific context, but we believe that there are some general guidelines that can be applied in the selection of the error model, namely:

- When comparing drivers on their potential to spread of errors, or evaluating the robustness of the OS to driver errors all three error models (and the composite) suffice to give guidance using the Diffusion metric. The experiments also validate the effectiveness of the Diffusion metric as an initial guideline.
- Data type errors come with a higher implementation cost, whereas bit-flips have a higher execution cost. If implementation cost (time) is a critical factor then bit-flips or fuzzing are recommended. Fuzzing gives similar Diffusion results as bit-flips with fewer injections. Thus making it the appropriate model to use when comparing drivers using Diffusion.
- When identifying services that may have serious failures, bit-flips is the most efficient error model followed by data type. However, fuzzing, being random in nature, may find cases where other models do not.
- A new composite error model, consisting of selective bit-flips with a series of fuzzing injections gives accurate results at a moderate execution/setup cost, compared to performing extensive bit-flip campaigns together with fuzzing injections.

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 and H. Madeira. Multidimensional Characterization of the Impact of Faulty Drivers on the Operating System Behavior. *IEICE Trans.*, E86-D(12):2563–2570, Dec. 2003.

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

[4] T. Ball et. al. Thorough Static Analysis of Device Drivers. *Proc. of EuroSys*, pp. 73-85, 2006.

[5] R. Chillarege, et. al. Orthogonal Defect classification-a Concept for In-Process Measurements. *IEEE TSE,* , 18(11):943–956, 1992.

[6] J. Christmansson and R. Chillarege. Generation of an Error set that Emulates Software Faults Based on Field Data. *Proc. of FTCS*, pp. 304 – 313, 1996.

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

[8] W. Gu, et. al. Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors. *Proc. of DSN*, pp. 887–896, 2004.

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

[10] M. Howard and S. Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

[11] T. Jarboui, et. al. Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study. *Proc. of PRDC*, pp. 51 – 58, 2002.

[12] T. Jarboui, et. al. Experimental Analysis of the Errors Induced into Linux by Three Fault Injection Techniques. *Proc. of DSN*, pp. 331– 336, 2002.

[13] A. Johansson and N. Suri. Error Propagation in Operating Systems. *Proc. of DSN*, pp. 86–95, 2005.

[14] K. Kanoun, et. al. Benchmarking the Dependability of Windows and Linux using PostMark Workloads. *Proc. of ISSRE*, pp. 11–20, 2005.

[15] P. Koopman and J. DeVale. Comparing the Robustness of POSIX Operating Systems. *Proc. of FTCS*, pp. 30–37, 1999.

[16] E. Marsden and J.-C. Fabre. Failure Mode Analysis of CORBA Service Implementations. *Proc. of Middleware*, pp. 216–231, 2001

[17] B. P. Miller, et. al. An Empirical Study of the Reliability of Unix Utilities. *CACM*, 33(12):32–44, Dec. 1990.

[18] R. Moraes, et. al. Injection of Faults at component interfaces and inside the component code: are they equivalent? *Proc. of EDCC*, pp. 53–64, 2006.

[19] P. Oehlert. Violating Assumptions with Fuzzing. *IEEE Security & Privacy Magazine*, 3(2):58–62, 2005.

[20] M. M. Swift, et. al. Improving the Reliability of Commodity Operating Systems. *Proc. of SOSP*, pp. 207–222, 2003.

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