Impact of Error Models on OS Robustness Evaluations*

Stefan Winter‡, Constantin Sârbu‡, Andréas Johansson† and Neeraj Suri‡
Technische Universität Darmstadt‡
Hochschulstr. 10, Darmstadt, Germany
{sw,cs,suri}@cs.tu-darmstadt.de

Volvo Technology Corporation†
Sven Hultins gata 9C, Göteborg, Sweden
andreas.olof.johansson@volvo.com

1 Introduction

Operating Systems (OSs) are mediators between application specific software (SW) and general purpose hardware (HW). For the operational delivery of OS services, device drivers (DDs) have increasingly been identified as a prominent cause of OS failures [1, 2]. Multiple research efforts, including ours, have targeted the robustness evaluation of commercial OSs's DD interfaces, for which robustness evidence is usually difficult to obtain due to the limited amount of available information about the OS's internal structure or its development process. A widely used approach for robustness evaluation is the application of experimental *software implemented fault injection* (SWIFI) techniques to expose the DD interface to operational conditions which explicitly violate its specification. A key factor determining the effectiveness of SWIFI is the selection of the most relevant *error model*.

Error models describe how the OS's DD interface specification is violated by defining three fundamental properties. The *error type* describes how a modeled perturbation manifests itself in the OS's operational environment, i. e. as a fault[1] in a DD or an error at the OS/DD interface. The *error location* specifies where in the OS's operational environment a perturbation manifests itself. The *error timing* refers to injection triggers and latencies.

Goals: The goal of our research is to investigate how the choice of an error model can influence the outcome of robustness evaluations. The results of such an investigation are useful in several ways. If, for instance, one error model can be shown to detect more vulnerabilities with less effort, it is advisable to choose this model over less efficient models for systematic vulnerability detection. If, on the other hand, evaluations of two distinct models yield highly similar results, the less costly model (in terms of implementation, setup, and run time effort) can be justifiably substituted for the more expensive one. Furthermore, a comparative evaluation of error models provides guidance on error model selection for evaluation efficiency optimization to system engineers.

Related work: Moraes et al. provided a comparison of two SWIFI frameworks with differing error models and fundamentally different injection mechanisms in [4]. The goal of their comparison was to investigate whether injections of errors *at the interface* of a component under evaluation (CUE) (e. g. at an OS API) and of faults *into an interacting component* (e. g. an application using the respective API) are equivalent with respect to the resulting failure mode distribution of the CUE. Besides this difference in terms of location, the applied error models also differed in terms of the error type: the faults that were injected into interacting components were specifically chosen to resemble the effects of common programming mistakes, while at the CUE interface errors were injected by corrupting parameter values according to their respective data types. For both error models injections were performed with permanent latency.

In prior work from our group in [3], three interface injection models were compared with respect to the obtained failure mode distributions of the CUE as well as various other effort and performance measures. For these evaluations, transient errors were injected on the first occurrence of a targeted function call. The modeled error types were data type (DT) specific parameter corruptions, parameter value bit flips (BF), and parameter value substitutions with random values (*fuzzing*, FZ). Model evaluations were specifically performed targeting the OS/DD interface of Windows CE 4.2. Interactions with three different DDs were subject to injection: a serial port driver, an Ethernet driver and a flash disk driver.

2 Error Model Impact: First Results

While our SWIFI framework supported complex injection triggers and different error types, it was technically incapable of performing code mutation based injections and thus not applicable for comparisons of code mutation and interface injection models, such as the one presented in [4]. We recently enhanced the framework to support code mutations of DDs while preserving the framework's flexibil-

*Supported in part by Microsoft, IBM, GK MM, and EC Inco-Trust and Inspire

[1]We acknowledge the conceptual distinction, but stick to the widespread terminology of *error* models for *fault* injection.

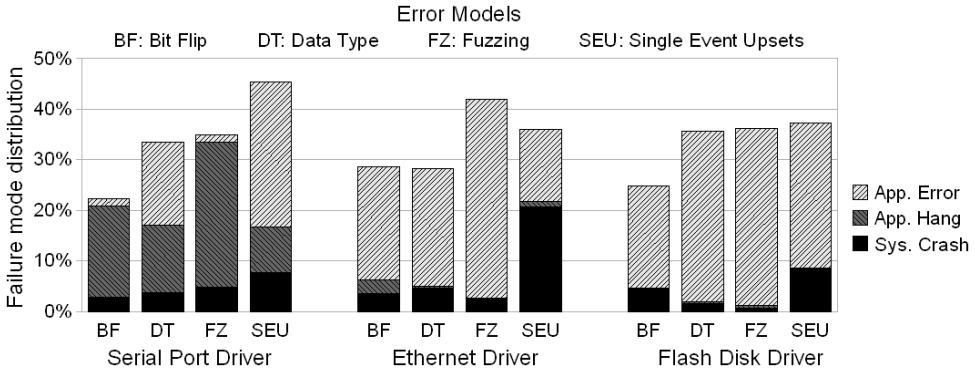

Figure 1. Failure mode distributions for four error models and three targeted drivers.

ity in terms of injection triggers and latencies. Furthermore, we defined a set of efficiency metrics for error model comparison which extends and refines the metric set presented in [3] and performed a re-evaluation of the previously implemented interface injection models. We also implemented a code mutation based error model, which emulates (transient) HW-induced SW errors by flipping single bits of DDs's binary code (*single event upsets*, SEU) and evaluated this model according to the metric set.

The comparison of the error models supports the conclusion from [4] in so far as none of the interface injection models results in a failure mode distribution anywhere near to the one resulting from the code mutation model (cf. Figure 1). However, the obtained failure mode distributions for the code mutation model also differ greatly from that in [4]. We observed a remarkably higher fraction of system crashes for this model than for any interface injection model. A thorough study of the experiment logs revealed that, for most calls to targeted DD functions, control was not returned to the calling kernel component. The injected fault resulted in a crash system failure without any detectable error propagation at the OS/DD interface.

Although we cannot yet provide evidence for this hypothesis, we postulate that for those cases the error propagation from DDs to the OS's kernel takes place by direct kernel space memory corruption. In the evaluated OS DDs are running in kernel mode and hence have the necessary access rights to do so. This invalidates the assumption of exclusively explicit interface interactions, which is stated as a necessary condition in the definition of software components provided by Szyperski in [5].

3 Future Directions

The recent extension of our SWIFI framework to support flexible injection triggers and latencies for code mutation based error models enables the comparative evaluation of a very large set of models. The preliminary results for transient errors yielded a number of unexpected results which require further investigation.

Investigation of OS specific effects: The obtained failure mode distributions appear to be highly target system dependent. In order to determine by which degree failure mode distributions (and other criteria applied for error model comparison) actually depend on error model characteristics, we are planning to port our framework to a number of other OS platforms.

Investigation of DD specific effects: From Figure 1 we can clearly derive the presence of a DD specific effect. For the serial port driver there is a substantially larger fraction of application hang failures than for the flash disk driver or the Ethernet driver, independent of the applied error model. It is, however, yet unclear whether the observed effect is specific for the *targeted DD* or the *targeted DD class*, i. e. whether we would obtain the same effect for any other serial port driver or only for this specific targeted instance.

Identification and evaluation of robustness hardening techniques: Fault containment wrappers have been successfully applied to increase the robustness of SW components in many scenarios. However, they are conceptually based on the assumption of explicit interface interactions among SW components and would thus not suit the observed case of direct mutual memory corruption. Hence, we are planning to identify, evaluate, and refine techniques to enforce SW component isolation, e. g. memory partitioning, which are efficiently applicable to off-the-shelf SW and HW components.

References

[1] A. Chou et al. An empirical study of operating systems errors. In *Proc. SOSP*, pp. 73–88, 2001.

[2] A. Ganapathi. Why Does Windows Crash? Technical Report CSD-05-1393, UC Berkeley, 2005.

[3] A. Johansson. Robustness Evaluation of Operating Systems. PhD Thesis, TU Darmstadt, 2008.

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

[5] C. Szyperski *Component Software – Beyond Object-Oriented Programming.* Addison-Wesley, 1998.