Robustness Evaluation of Operating Systems

Vom Fachbereich Informatik der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades eines Doktor-Ingenieur (Dr.-Ing.)
vorgelegt von

Andréas Johansson

aus Falkenberg, Schweden

Referenten:
Prof. Neeraj Suri, Ph.D.
Prof. Christof Fetzer, Ph.D.

Datum der Einreichung: 19.11.2007
Datum der mündlichen Prüfung: 14.01.2008

Darmstadt 2008
D17

Summary

The premise behind this thesis is the observation that Operating Systems (OS), being the foundation behind operations of computing systems, are complex entities and also subject to failures. Consequently, when they do fail, the impact is the loss of system service and the applications running thereon. While a multitude of sources for OS failures exist, device drivers are often identified as a prominent cause behind failures.

In order to characterize the impact of driver failures, at both the OS and application levels, this thesis develops a framework for error propagation-based robustness profiling for an OS. The framework is first developed conceptually and then experimentally validated on a real OS, namely Windows CE .Net. The choice of Windows CE is driven by its representativeness for a multitude of OS's, as well as the ability to customize the OS components for particular needs.

For experimental validation, fault injection is a prominent technique that can be used to simulate faults (or errors) in the system by inserting synthetic ones and study their effect. Three key questions with such a technique are *where*, *what* and *when* to inject faults. This thesis shows how injecting errors at the interface between drivers and the OS can be very effective in evaluating the effects driver faults can have.

To quantify the OS's robustness, this thesis defines a series of error propagation measures, specifically tailored for device drivers. These measures allow for quantifying and comparing both individual services and device drivers on their susceptibility and diffusing abilities.

This thesis compares three contemporary error models on their suitability for robustness evaluation. The classical bit-flip model is found to identify a higher number of severe failures in the system. It also identifies failures for more services than both other models, data type and fuzzing. However, its main drawback is that it requires substantially more injections than the other two. Fuzzing, even though not giving rise to as many failures is able to find new additional services with severe failures.

A careful study of the injections performed with the bit-flip model shows that only a few bits are generally useful for identifying new services with robustness weaknesses. Consequently, a new composite model is proposed, combining the most effective bits of the bit-flip model with the fuzzing model's ability to identify new services, giving rise to new model without loss of important information and at the same time incurring a moderate number of injections.

To answer the question of when to inject an error this thesis proposes a novel model of a driver's usage profile, focusing on high-level operations being carried out. It guides the injection of errors to instances when the driver is carrying out specific operations. Results from extensive fault injection experiments show that more service vulnerabilities can be discovered. Furthermore, a priori profiling of the drivers can show how effective the proposed approach will be.

Kurzfassung

Der Hintergrund dieser Dissertation beruht auf der Beobachtung, dass das Betriebssystem, welches die Grundlage für den Betrieb von Rechnersystemen darstellt, eine sehr komplexe Struktur aufweist, was häufig zu Fehlern im Betriebssystem führen kann. Wenn diese betriebssysteminternen Fehler Ausfälle von Diensten zur Folge haben, sind auch die im Rahmen des Betriebssystems laufenden Applikationen gefährdet. Auch wenn es im allgemeinen viele Fehlerquellen gibt, werden oft fehlerhafte Treiber als die häufigste Ursache angegeben.

Um die Auswirkungen von Treiberdefekten auf der Betriebssystem- und Applikationsebene zu charakterisieren, wird in dieser Dissertation ein auf der Ausbreitung von Fehlern basierendes Framework für Robustheitsauswertung entwickelt. Das Framework wird sowohl konzeptionell entwickelt als auch auf einem echten Betriebssystem experimentell validiert. Das gewählte Betriebssystem, Windows CE .Net, ist repräsentativ für viele andere Betriebssysteme. Es ist modular aufgebaut, was die Anpassung der Betriebssystemkomponenten an verschiedene Bedürfnisse erheblich vereinfacht.

Fehlerinjektion ist eine bedeutende Technik für die experimentelle Validierung, wobei Fehler simuliert werden indem man sie in das System injiziert und ihre Folgen beobachtet. Drei wichtige Aspekte, die hierbei berücksichtigt werden müssen, sind: Welche Fehler sollen wo und wann injiziert werden. In dieser Dissertation wird gezeigt, dass Fehlerinjektion in die Schnittstelle zwischen dem Betriebssystem und den Treibern eine effektive Vorgehensweise darstellt, die Folgen von Treiberfehlern abzuschätzen.

Um die Robustheit eines Betriebssystems zu quantifizieren, werden eine Reihe von Fehlerausbreitungsmetriken definiert, die speziell auf Treiberfehler zugeschnitten sind. Anhand dieser Metriken können Dienste und Treiber hinsichtlich Empfindlichkeit und Ausbreitungsvermögen verglichen werden.

Diese Dissertation vergleicht drei zeitgemäe Fehlermodelle in Bezug auf ihre Tauglichkeit zur Robustheitsbewertung. Das klassische Bit-Flip-Modell ermittelt am häufigsten schwere Ausfälle im System. Mehr als die beiden anderen Modelle, Data Type und Fuzzing, ermittelt dieses Modell auch die meisten Dienste, die zu Ausfällen führen könnten. Der gröte Nachteil dieses Modells ist allerdings, dass es sehr viele Injektionen erfordert. Fuzzing ermittelt weniger Dienste, dafür aber neue fehlerhafte, von Bit-Flip nicht erkannte Dienste.

Eine sorgfältige Untersuchung der Ergebnisse des Bit-Flip-Modells zeigt, dass schon eine Teilmenge der Bits ausreichend ist, um neue Dienste, die zu Robustheitsausfällen führen, zu ermitteln. Daraufhin wird ein neues, zusammengesetztes Modell vorgeschlagen, das die guten Eigenschaften des Bit-Flip-Modells und das Vermögen des Fuzzing-Modells neue Dienste zu identifizieren miteinander kombiniert. Das neue Modell verliert keine wichtige Information, und erfordert insgesamt deutlich weniger Injektionen.

Um die Frage zu beantworten wann es sinnvoll ist Fehler zu injizieren, wird ein neues, an das Benutzerprofil des Treibers angelehntes Timingmodell vorgeschla-

gen. Das neue Modell basiert auf der Ausführung von Befehlen in einer höheren Schicht. Bestimmte Fehlerinjektionen werden zum Zeitpunkt der Ausführung bestimmter Befehle getätigt. Die Ergebnisse der Fehlerinjektionen zeigen, dass ein Vielfaches an störungsanfälligen Diensten gefunden werden kann. Auerdem gibt das Benutzerprofil des Treibers im Voraus Aufschluss über die Effektivität der neuen Methode.

Acknowledgements

The path to a Ph.D. is a long, winding one. At the beginning it is wide and spacious, only to become narrower and narrower. Sometimes it goes steeply upwards, sometimes downwards. Sometimes you think you see an opening and light after the next turn, only to find a dead end. There are crossings, where one has to choose which path to pursue. Some paths look more promising than others but you quickly learn that the easy path is not always the shortest.

I am now at the end of the path, only to realize that it is the beginning of a new. I would not have gotten here without the assistance and encouragement of several people. First of all I would like to thank my guide and mentor, Prof. Neeraj Suri. Thanks for your guidance and support. I would also like to thank all present and former members of the DEEDS group: Martin, Vilgot, Örjan, Arshad, Robert, Adina, Dinu, Marco, Dan, Ripon, Brahim, Faisal, Majid and Matthias. Many thanks also to Birgit, Ute, Sabine and Boyan. A great many thanks also to Prof. Chrstof Fetzer for accepting to be my opponent.

I am also grateful for funding for conducting my research coming from the projects EC IP DECOS, EC NoE ReSIST and by research grants from Microsoft Research. A part of the research validation was accomplished over an internship at Microsoft Research, Cambridge. A special thanks to Brendan Murphy at MSR for hosting my internship and for all discussions and help writing papers.

Finally to my beautiful and supporting wife, Mia. Thank you for everything!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

What is robustness, and why is it important?

As the usage of computers proliferates, a consequence is our increasing reliance of their operations in diverse application environments. The use of computers, and especially computer software, promises many advantages compared to electronic or purely mechanical solutions, including rapid development, flexibility, effective component reuse (both software and hardware), no aging effects etc.

However, software brings about new problems. Fulfilling not only functional requirements, but also requirements on determinism, real-time and dependability properties become increasingly difficult. Software engineering tries to handle these problems by structuring the development process. However, that engineering software is difficult has long been known. Leveson notes that as software is divided into components (a well established technique to handle complexity) a new complexity is introduced in the many explicit and implicit interfaces that arise [Leveson, 1995]. Furthermore, the lack of physical constraints makes software inherently more flexible (which is positive) but also gives rise to new, unexpected and unintended interactions (which may be hard to find, quantify and master). In contrast to physical systems, small perturbations in software may give rise to serious failures without much delay. These problems require new methods for building and verifying systems based on software.

A key design model used to handle some of the complexities is to use standard platform components to build applications upon, the OS being the most significant such software platform. The OS forms the basic interface to which applications and services can be built. Consequently a reliance on continued provisioning of correct service is put on the OS, and when this is

not the case the system might not function properly.

This thesis addresses the problem of evaluating the robustness of an OS, i.e., to which degree an OS tolerates perturbations in its environment. Such evaluations can serve several purposes, such as gaining information on how the system can fail when operational, guiding verification/validation efforts towards services which are more likely to spread or be the of errors, and to guide the addition of robustness enhancing components where they are most effective.

This chapter first presents the basic terminology used in the thesis and then introduces the area of robustness evaluation. The research problems addressed are presented and discussed together with the contributions provided.

1.1 Dependability: The Basic Concepts

Dependability is the ability of a system to avoid service failures that are more frequent and more severe than is acceptable [Avižienis et al., 2004]. This definition implies that the system is well specified, together with the services it provides, such that failures of the system can be clearly defined and detected. It also requires acceptable service failure frequencies to be established and that the severities of failures are known and can be estimated.

[Avižienis et al., 2004] is a collective effort by the dependable computing community to agree on a set of standard terms. Further definitions can for instance be found the the *IEEE Standard Glossary of Software Engineering* [IEE, 1990] or in *Dependability: Basic Concepts and Terminology*, which presents the basic terminology in five different languages [Laprie, 1992]. This section provides a brief introduction to the terms most commonly used in the field and relevant to the work in this thesis.

Dependability can be seen as an umbrella, incorporating several attributes, including not only attributes directly related to *functionality*, but also attributes related to *security*. In this thesis, no emphasize is put on security related attributes. They are included and discussed shortly in this chapter for completeness. Figure 1.1 shows an overview of the attributes of dependability, the threats to dependability and the means to achieve dependability.

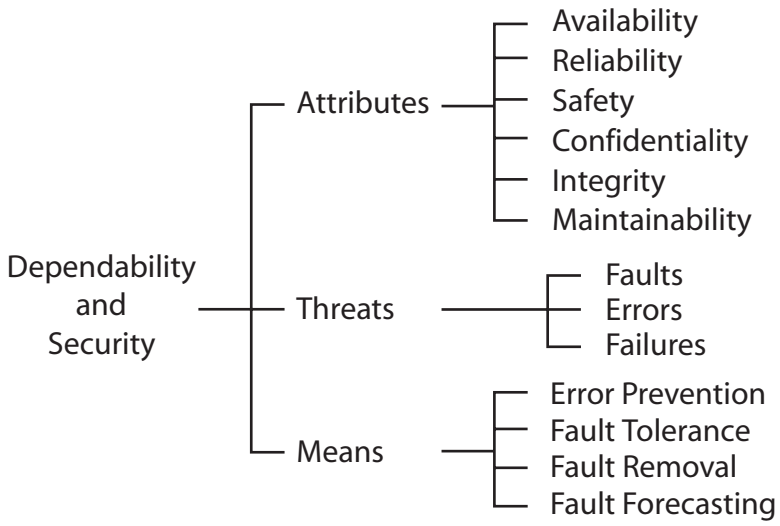

Figure 1.1: The dependability and security tree, from [Avižienis et al., 2004].

1.1.1 Dependability Attributes

Dependability is a composite term, encompassing several aspects relating both to provision of functionality, security and maintainability:

- **Availability** - The ability of the system to be ready for use when required
- **Reliability** - The ability of the system to continuously provide stipulated services for a specified period of time
- **Safety** - The absence of catastrophic consequences on users and the environment
- **Integrity** - The absence of improper system alterations
- **Confidentiality** - The absence of disclosure of confidential information to unauthorized entities
- **Maintainability** - The ability of the system to undergo repairs and modifications

Dependability and security are obviously related. Availability, for instance, is a concern both from a dependability perspective (lack of service) and from a security perspective (denial of service). Figure 1.2 shows how dependability and security attributes are related.

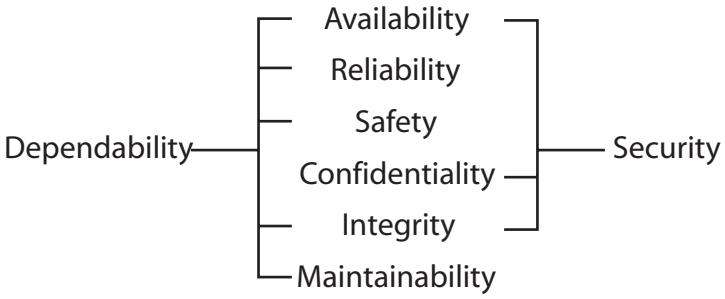

Figure 1.2: The attributes of dependability and security, from [Avižienis et al., 2004].

1.1.2 Dependability Threats

To facilitate a discussion regarding the causes, effects, detection and recovery from faults in the system a distinction is made between *faults*, *errors* and *failures*. Faults are the causes of failures in the system by being activated (becoming errors) and then propagating to the outputs of the system and there causing a failure. Figure 1.3 illustrates how a fault propagating to a failure of one component (Component A) can be the input (fault) of another component (Component B) and so on.

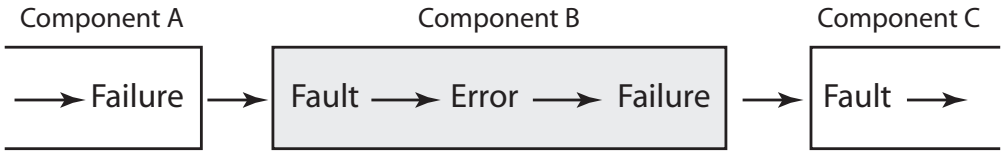

Figure 1.3: The fault → error → failure process.

Faults

Faults are the sources for errors and failures of a system or component, including faults appearing during development, physical faults in hardware and interaction faults occurring in interactions with external components. A fault by itself is not sufficient to cause a failure of the system, it must also be *activated*. Certain triggering conditions are required for the fault to be activated. For instance, the part of the hardware containing the fault must be used, or for software the code containing the fault must be executed. When the fault is activated it becomes an *error*. As the triggering mechanism for the fault activation process typically is time-dependent faults in the system can be present without immediately being activated and are then called *dormant* faults. A good example thereof is a software fault (bug) in a module that is only triggered for certain input values, which may appear at some later point in time as the module is used.

Errors

Errors change the internal state of the system in a way that may cause the system to fail. For this to happen the error must cause a series of chain reactions, where the error is *propagated* through the system by internal computations. Errors are transformed into other errors in a similar manner as faults are activated. Eventually a propagated error may cause an incorrect service output (or lack of output) violating the specification for the system,

i.e., a *failure*. Thus, also errors can stay dormant in the system, waiting for the triggering conditions for propagation to take place, before it propagates.

Failures

Failures are observed on the outputs of the system and are detectable as deviations from an assumed specification. As illustrated in Figure 1.3 a failure may itself *cause* a fault in another component. There are multiple facets to failures in a system. Some failures may be of higher *criticality* than others. Similarly, a failure must not mean a total absence of service provisions, some systems can provide a limited level of service, i.e., there is a service degradation.

1.1.3 Dependability Means

There are four ways in which dependability can be achieved and analyzed: fault *prevention*, fault *tolerance*, fault *removal* and fault *forecasting*. This thesis is mainly concerned with fault forecasting, and to some degree with fault tolerance and removal.

Fault Prevention

The main intent with fault prevention is to avoid introducing faults in the system during its development by use of mature software engineering practices and tools. Faults arising in the field are avoided by the use of high quality hardware.

Fault Tolerance

Fault tolerance is a fundamental switch in mental model compared to fault prevention. In fault prevention one avoids to introduce faults in the system. In fault tolerance on the other hand, faults are assumed to be present in the system, due to imperfect design methodologies, aging of hardware, interaction faults with components outside the control of the development team etc. Fault tolerance is based on the premise of error detection and recovery. Errors are detected and recovered from, or errors are masked using redundancy. Dependability is then achieved by tolerating the faults rather than avoiding to introduce them, which may be very hard, or too costly.

Fault Removal

Fault removal aims to remove faults either during the development stage of the system or during the operational stage. Development stage methods are broken down into verification and validation, where verification relates to verifying that an implemented system actually implements the specification given, and validation to checking the specification itself. At runtime diagnosis and compensation techniques can be used to remove faults from the system.

The contributions in this thesis relates mainly to verification, more specifically to dynamic verification, such as testing.

Fault Forecasting

Fault forecasting aims to qualitatively and quantitatively evaluate system behavior in the presence of faults. It aims to establish failure modes of the system and to evaluate other system attributes regarding the dependability of the system. The intent is not the same as in fault removal (e.g., verification) but to establish operational characteristics of the system.

The thrust of this thesis is on robustness evaluation, which is a part of fault forecasting.

1.1.4 Alternate Terminology: Software Engineering

In the area of software engineering a slightly different terminology for dependability facets exists. In software engineering an *error* represents the mistake made by the programmer that made him/her introduce a flaw in the code, the *fault* (also known as bug). The consequence of the dormant fault is that it may get activated and then propagate to the software outputs, causing a *failure* of the component.

In this thesis we will consistently use the terminology from the dependability community as presented in 1.1. It allows for a discussion on the representativeness of injected errors and is aligned with the large body of work presented in Chapter 2.

1.1.5 Bohrbugs and Heisenbugs

As stated our main focus is on faults originating from software. Furthermore, we focus on the subset of software faults that are transient in nature and require complex triggering for activation. These faults, known as Heisenbugs [Gray, 1985] are of key interest because they are less likely to be found using traditional testing techniques. They represent faults that rarely appear in

normal circumstances and contexts and are therefore harder to find. The opposite, Bohrbugs, have simpler and deterministic activation conditions and are easily repeatable. Some authors use the term Mandelbug for bugs which given the exact same conditions, sometimes appear, sometimes not [Grottke and Trivedi, 2007]. Using this terminology Heisenbugs are a subset of Mandelbugs.

1.2 Robustness Evaluation

A related term to dependability is *robustness*. Robustness is defined as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [IEE, 1990]. Robustness is therefore related to and an influence on the dependability of the system. At the same time it is more restricted than dependability, since it only relates to external perturbations and not internal ones.

The focus of this thesis is on the robustness of OS's, as it gives useful and meaningful information about the system without requiring a specific operational scenario to be in place, as would be the case for for instance reliability or availability. Robustness is concerned with the cases were external components (including human users) do not behave as expected or as stipulated/assumed by the designer of a component. As such, robustness evaluation complements traditional verification and validation techniques (also formal ones). The goal of robustness evaluation is to identify potential *vulnerabilities*[1] in the system. Such vulnerabilities may or may not be triggered in a specific operational scenario. They may or may not constitute design faults (e.g., software bugs) in the system, and they may lead to severe consequences for availability, reliability, safety or security.

As Commercial-Off-The-Shelf (COTS) components are more and more becoming standard building blocks in modern designs their composition is a key aspect of the verification process. Robustness of individual components is of great importance since components built to be re-used in multiple contexts cannot be built with any such explicit context in mind. When combined with new components, having different failure characteristics, components may be faced with unexpected and abnormal inputs. Therefore, components should be built to respond robustly to such inputs and evaluating their robustness may reveal information on how well suited they are for a particular composition.

[1] The term vulnerability does not refer only to *security vulnerabilities*, but to weaknesses in a system that might lead to robustness failures. These might also include security-related weaknesses.

Many types of users may be interested in performing robustness evaluations as described in this thesis, including developers for debugging or profiling purposes; integrators for finding possible interaction problems between components; testers for identification of vulnerabilities; or system designers and managers for suitability tests, resource guidance or identification of weak components. We will emphasize when aspects apply to a particular aspect of software development, but use the more general term *evaluator* for the person or entity conducting the evaluation.

1.3 Thesis Research Questions & Contributions

The use of COTS components, such as OS's, is becoming more and more common, also for products with stringent requirements on dependability. For such component-based designs to fulfil these requirements, one needs to establish the amount of trust that can put on these components to work in a specific environment, including how well they handle faults appearing in other components of the system. To answer such a question, the failure characteristics of the OS need to be established. This includes how the OS can fail due to faults in the environment. Are certain services provided by the OS more vulnerable? Are certain other components more likely to cause a failure of the OS and consequently a failure of the system?

Using a model where the OS is the main platform component in a system also containing applications and device drivers interfacing with the hardware, these fundamental questions regarding the OS has guided the work presented in this thesis. To give insights into how such questions can be answered an experimental error propagation and effect framework has been defined, where synthetic errors are injected in the interface between the OS and its drivers. Drivers has been identified as one of the main contributors of OS failures [Murphy and Levidow, 2000; Simpson, 2003; Ganapathi et al., 2006]. Along the same line Chou et al. found that driver code contains up to seven times as many bugs as other parts of the Linux kernel [Chou et al., 2001]. As data on how a system handles errors typically is not available as the system is built, techniques are needed to speed up this process. One such technique is fault injection, where synthetical faults (or errors) are injected and the behavior of the system is observed. This methodology raises additional questions regarding how to inject errors[2], where to inject them, when to inject them and which error model to use. These three questions are fundamental to any

[2]Traditionally, the technique is called fault injection, even when *errors* are injected.

fault injection approach and are orthogonal, as illustrated in Figure 1.4.

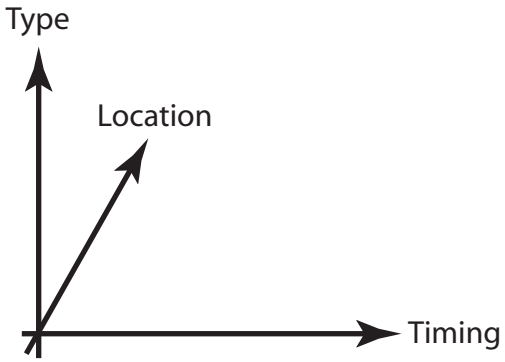

Figure 1.4: Three fundamental dimensions in fault injection.

Each injected error modifies some part of the system. The *error type* refers to how the system is modified, like flipping a bit, or assigning a wrong value. Where the error is injected is its *location* dimension, like in CPU registers, memory or in parameters to function calls. The *timing* of the injection specifies when the error is injected, relative to some system event, like boot up time. The timing can principally be time or event triggered.

One may argue that fault injection, being an experimental technique is inherently limited since it does not provide completeness. Injecting a number of faults and not finding any failures is no proof of correctness (as in the system's ability to handle all faults/errors). However, we argue that even with lack of completeness, evaluation using fault injection is still very useful since it gives information about how the system behaves in practise. Even though formal techniques and addition of several layers of fault-tolerance software may be desirable it is not always possible for large systems, such as an OS, due to performance, compliance or cost limitations.

On these premises we are interested in finding out how OS's behave in presence of errors, more specifically errors in device drivers. Consequently the following research questions are posed for this thesis:

Thesis Research Questions

The research questions posed for this thesis are grouped into two broad categories, first the conceptual definition of robustness profiling and the associated measures, and then the quantifiable experimental aspects of validating the proposed measures.

Category 1: Conceptual

Research Question 1 [RQ1]: *How do errors in device drivers propagate in an OS? What is a good model for identification of such propagation paths?*

Chapter 3 sets up the model used to evaluate and profile the OS. The model must allow for clear definition of propagation paths, and for useful, easily interpretable results to be extracted.

Research Question 2 [RQ2]: *What are quantifiable measures of robustness profiling of OS's?*

Chapter 5 presents a framework that allows for identification of error propagation paths, that help us quantify which services are more likely to spread errors in the system. It also allows us to identify for an application, which OS services used are more likely to be vulnerable to propagating errors. Additionally, device drivers can be ranked based on their potential diffusion of errors, allowing a designer to make informed choices on whether to include a driver in the system, to enhance its robustness or to find an alternate driver. Chapter 6 presents experimental results for a case study conducted for Windows CE .Net.

Category 2: Experimental Validation

Research Question 3 [RQ3]: ***Where to inject?*** *Where are errors representing faults in drivers best injected? What are the advantages and disadvantages of different locations?*

We have chosen to inject errors in the interface between the OS and its drivers. This level of injection provides flexibility and portability among other advantages. Chapter 3 presents our system model and shows where errors are injected. Chapter 4 details our injection framework, which allows for error to be injected.

Research Question 4 [RQ4]: ***What to inject?*** *Which error model should be used for robustness evaluation? What are the trade-offs that can be made?*

The choice of error model is not straightforward and there are trade-offs to be made on the amount of details provided, the time/effort required and the implementation complexity. It is shown in Chapter 6 how such trade-offs can be made and three distinct error models are evaluated, *bit-flips*, *data-type* and *fuzzing*. A novel *composite* error model is provided, combining the higher vulnerability exposure of the *bit-flip* error model with the low costs of the *fuzzing* error model.

Research Question 5 [RQ5]: ***When to inject?*** *Which timing model should be used for injection?*

When doing in-situ fault injection experiments, which are needed for robustness evaluation, the time of injection becomes an issue. The state of the system evolves as it executes and consequently also its susceptibility to faults. A novel approach to selecting the time of injection is presented in Chapter 7, based on the usage profile of the driver.

Thesis Contributions

The research presented here constitutes several important contributions to the research community. Each contribution lists the corresponding research questions it helps answer in brackets.

Contribution 1: A framework is presented for characterizing error propagation in an OS, focusing on a key source of OS failures, errors in device drivers. [RQ1, RQ2, RQ4]

Contribution 2: A series of error propagation measures are defined, which are used to profile the robustness of the OS. [RQ2, RQ4]

Contribution 3: A large scale case study for Windows CE .Net has been carried out, where fault injection is used to validate the proposed measures. [RQ3, RQ4, RQ5]

Contribution 4: A detailed investigation on the effectiveness and efficiency of several error models for use in OS robustness evaluations. Models are compared on several parameters, including number of provoked failures, service coverage, required execution time and implementation complexity. [RQ3]

Contribution 5: We show how a new *composite error model* can be used when profiling drivers, combining desirable properties of other models giving excellent coverage characteristics for a moderate number of injections. [RQ3]

Contribution 6: The impact of the time of injection is studied and it is shown that for a certain class of drivers, the impact is high. This indicates that controlling the time of injection is important. [RQ5]

Contribution 7: A novel approach to selecting the right time to inject is presented together with a large case study supporting the results. The model uses the new concept of *call block* to define the time of injection. [RQ5]

Contribution 8: A flexible fault injection framework for Windows CE .Net has been implemented and used to carrying out the fault injection experiments required. The framework allows for easy extension to new error models, drivers and services. [RQ3, RQ4, RQ5]

Publications Resulting from the Thesis

The work reported in the thesis is supported by a number of international publications:

- **Andréas Johansson**, Neeraj Suri and Brendan Murphy, *On the Impact of Injection Triggers for OS Robustness Evaluation*, Proceedings of the International Conference on Software Reliability Engineering (ISSRE), 2007.
- **Andréas Johansson** and Neeraj Suri, *Robustness Evaluation of Operating Systems*, Chapter 12 of *Information Assurance: Dependability and Security in Networked Systems*, Editors: Yi Qian, James Joshi, David Tipper and Prashant Krishnamurthy, To be published in 2007 by Morgan Kaufmann.
- **Andréas Johansson**, Neeraj Suri and Brendan Murphy, *On the Selection of Error Model(s) For OS Robustness Evaluation*, Proceedings of the International Conference on Dependable Systems and Networks (DSN), 2007.
- **Andréas Johansson** and Neeraj Suri, *Error Propagation Profiling of Operating Systems*, Proceedings of the International Conference on Dependable Systems and Networks (DSN), 2005.
- **Andréas Johansson**, Adina Sârbu, Arshad Jhumka and Neeraj Suri, *On Enhancing the Robustness of Commercial Operating Systems*, Proceedings of the International Service Availability Symposium (ISAS), Springer Lecture Notes on Computer Science 3335, 2004.

Additionally, the author has been involved in the following publications that are not directly covered by the thesis:

- **Andréas Johansson** and Brendan Murphy, *Failure Analysis of Windows Device Drivers*, Workshop on Reliability Analysis of System Failure Data, Cambridge UK, 2007.
- Constantin Sârbu, **Andréas Johansson**, Falk Fraikin and Neeraj Suri, *Improving Robustness Testing of COTS OS Extensions*, Proceedings of the International Service Availability Symposium (ISAS), Springer Lecture Notes on Computer Science 4328, 2006.
- Neeraj Suri and **Andréas Johansson**, *Survivability of Operating Systems: Profiling Vulnerabilities*, FuDiCo II: Bertinoro Workshop on Future Directions in Distributed Computing, 2004.

1.4 Thesis Structure

The structure of the following chapters follows the structure of the research questions postulated previously:

Chapter 1 introduces the research problems studied and the contributions. Also, it introduces the terminology used throughout the thesis.

Chapter 2 gives a background and context to the problems approached in this thesis by surveying related work.

Chapter 3 presents and discusses the system and error model used. The experimental environment is presented, both in terms of hardware and software.

Chapter 4 presents our experimental methodology and presents details regarding the fault injection technique used.

Chapter 5 introduces our error propagation framework and introduces the key measures used for of error propagation and effect analysis. Their use and interpretation is discussed.

Chapter 6 investigates the impact of the choice of error model by presenting a comprehensive experimental evaluation of three error models. The evaluation builds on the measures introduced in Chapter 5.

Chapter 7 shows the impact of the time of injection and presents a novel approach to choosing relevant injection times.

Chapter 8 finally puts the contributions of the thesis back into context by discussing the general conclusions to be drawn. Additionally a discussion on how the results can be applied for several other research fields is provided and future research directions are outlined.

Chapter 2

Background and Context

What is an OS, and how has its robustness been evaluated? What is the state of the art and state of the practice in OS robustness evaluation?

Over the years the OS has evolved in its complexity and roles. What started as a program to help computer operators read jobs from tapes for large mainframe computers, is today present in a multitude of computing products and responsible for serving multiple concurrent users and handling a wide range of devices. The sophistication of the services provided has increased tremendously over the years, as has the reliance on the correct and timely provision of service to applications and users. This has given rise to a whole area of dependability evaluations and enhancements.

This chapter aims to relate the work presented in this thesis to the large body of work performed by other researchers. Thus it forms the background and the context for the research questions posed and puts the contributions presented into perspective.

2.1 A Short Operating System History

The first computers were programmed per hand and the programs were given to an administrator as punch cards, which then placed them in the card reader for the computers. As computers evolved and the uses and requirements for computations increased it became evident that some form of control software was needed, both to abstract away the intricacies of the hardware and to allow for concurrent access for multiple users. The OS was born to handle multiple *jobs* that needed time on the CPU. At first these jobs were batched and the role of the OS was to read the code for one job into memory (from tapes or punch cards) and when it was finished write the output on printers, tapes etc. One major issue with batching of jobs was that while the computer, which was a horrendously expensive piece of equipment, was waiting for some external device it could not make any progress and was simple idle. This was solved when *multiprogramming* was introduced in OS's. The memory available to the computer was partitioned across multiple jobs, such that when one job was waiting for some I/O operating to complete, another job could use the processor to perform computations. Further improvements followed, such as *timesharing* where multiple users attached to terminals could share the computer, by dividing the time used on the processor across the users. As computers became smaller, faster and more user friendly, the number of computer users also increased. Several different OS's evolved, the most prominent ones being first UNIX (which comes in many flavors, including OS's like GNU/Linux and Mac OS X/Darwin), later followed by Microsoft DOS and Windows. Many special purpose OS's were developed, for instance for Real-Time systems, or for large-scale servers. Good text books on general OS related themes include the classical books by Tanenbaum [2001] and Silberschatz et al. [2004].

2.1.1 OS Design

One of the key goals for an OS is *protection*. It should prevent users and processes to gain access to data (read, modify, execute etc), devices and other processes in an uncontrolled manner. This includes both unintentional and intentional (even malicious) accesses. A common technique to enforce this is to define (in hardware) different privilege levels, where processes executing with higher privilege can access lower privilege processes, but not the other way around. For most OS's two such levels (or modes) are defined, *user level* and *kernel level*. Only at the kernel level is it possible to use some processor instructions. By executing the OS at the higher privilege level (kernel mode) it can control user processes' access to the system. Naturally

failures of kernel mode components are potentially more severe than user mode components, since few protection mechanisms exist to prevent them from corrupting important system data and components.

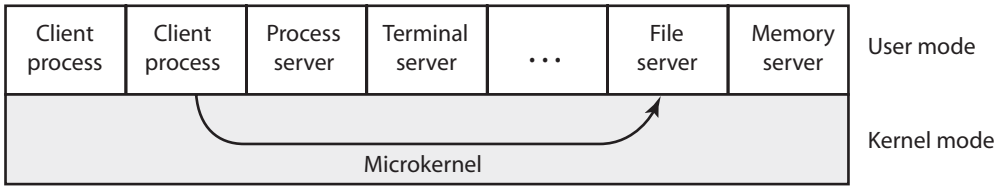

Figure 2.1: Example of microkernel design. Figure from [Tanenbaum, 2001].

There have been two main design principles for general OS's, the monolithic kernel and the microkernel-based design. In a microkernel-based design the OS kernel is kept small and provides only low level services, such as process and memory management, inter-process communication (IPS) etc. The microkernel is the only entity of the OS running in privileged mode. Other services that one wants the OS to provide, such as file systems, device drivers etc execute in user mode (and are often referred to as *servers*) as illustrated in Figure 2.1. Applications request OS services using IPC to the particular server providing the service, as shown by the arrow in the figure.

In a monolithic design on the other hand, all OS services execute in privileged mode, and applications make *system calls* to use the services[1] provided by the system. The model of the OS layered vertically, with each layer using services of lower layers, whereas the microkernel design is more of a horizontal design. This design is reflected in our system model, which is shown in Figure 3.1.

2.1.2 Device Drivers

Device drivers are, as the name suggests, responsible for interaction with devices. There are also drivers for virtual devices (protocols etc) and other software making use of the driver architecture to extend the functionality of the OS. A driver's role is to encapsulate and handle the device specific interaction needed in order for the OS and applications to use the device. As many devices have special functionalities, or use specific protocols, the drivers provide a middle layer between the OS and the devices.

[1]Throughout we will use the term *service* which is more general than the term *system call*. However, for the system used in this thesis they are synonyms.

In order to facilitate OS-driver interactions, and to make it easier to develop drivers, the interface between a driver and the OS is typically standardized. This means that a driver needs to implement certain functionality for the OS to be able to interact with it. In exchange the OS provides functionalities that make it easier to develop and maintain drivers. This way the OS can handle whole classes of drivers the same way, making it significantly easier to develop new devices (and drivers) for existing OS. Using device drivers also potentially simplifies the porting of the OS to multiple hardware architectures, as the drivers can be used handle parts of the hardware-specific features of different architectures.

2.1.3 What is the Problem?

There are several reasons why it is difficult to design and test an OS. First of all, most OS's are general-purpose, i.e., they are built to handle a wide variety of workloads and they can be highly parameterized to be used in different environments. Furthermore, the OS kernel runs in high-privilege mode, were failures easily take down the whole system. OS's often have long run-times, especially in the server and embedded areas, making them sensitive to resource exhaustion and leakage problems. Many OS functionalities are service-oriented, meaning that correctness of execution may be hard to define and limit, making testing and other means of verification and validation hard. Lastly the sheer size of modern OS's poses a problem for thorough verification and validation. To give a hint on size, Steve Jobs (CEO of Apple Inc.) was quoted to say that MacOS X contained 86 million lines of code [Jobs, 2006].

2.2 Sources of Failures of Operating Systems

This section will survey some of the sources for OS failures. To get information on common sources for failures the most straightforward technique is to collect data from deployed system in the field. Most companies collect failure data for their systems to some extent, but there are several aspects warranting consideration, such as privacy, user participation, unbiased data sets etc [Murphy, 2004].

One of the most influential papers within its field is Gray's 1985 classical paper "Why Do Computers Stop and What Can Be Done About It?" [Gray, 1985]. Studying outage reports for a large number of Tandem systems four main classes of sources for outages were identified: administration, software, hardware and environment. Administration and software were were found to

be the largest contributors (42% and 25% respectively). Another interesting finding was that a majority of the software faults investigated for a specific subsystem were Heisenbugs, not Bohrbugs, which supports the idea of using software fault tolerance through redundancy, e.g., process pairs etc. A later report in 1990 reports on a trend that software is increasingly being the source of failures (up to 60% in 1989).

The rest of the section covers different sources of faults. There are many possible classifications of faults, and in this thesis we will consider three classes: hardware, software and user-related faults. This section will review each of these in turn and relate them to OS failures. As device drivers is a major source of OS failures and of interest to this thesis, the last subsection is dedicated to faults in device drivers.

2.2.1 Hardware Related

Hardware related faults are faults stemming from physical defects or phenomena in the hardware platform upon which the OS is running. Hardware faults may have different causes, such as power glitches, wear-out/aging, radiation/EMI etc. Much work has been spent on characterizing and protecting against hardware faults. Common examples include error correcting codes on memory, redundant bus lines, redundant disks (RAID etc) and many other techniques. From an OS perspective these four classes of faults have broadly be defined in [Kao et al., 1993]:

- Memory faults, corrupting memory locations, either code or data,
- CPU faults, computation, control flow and register faults. They appear as register corruptions (PC, PSR SP etc),
- Bus faults, affecting bus lines, and
- I/O faults, external devices causing problems.

Cosmic rays penetrating the atmosphere may cause transistor voltage levels to transiently change when they hit chips. Due to their nature (transient) such errors are often referred to as *soft errors* (compared to hard, permanent errors). The rate at which they occur is referred to as the soft error rate (SER). It is expected that future chips will have an increase in SER due to the scaling of size and supply voltage [Shivakumar et al., 2002; Constantinescu, 2003, 2005].

By the principle of error containment, errors are best handled close to the source. However, many hardware errors may propagate to the software level

undetected. This is especially true for system-level software, including device drivers. Such errors may be difficult to discriminate from software faults. This relation was studied in [Iyer and Velardi, 1985]. Data was collected for an installation of the MVS OS at Stanford. The purpose of the study was to evaluate the OS's ability to detect and diagnose software errors that are related to hardware errors. The investigation showed that the OS was rarely able to correctly diagnose the error as hardware related, and less so than for pure software errors.

2.2.2 Software Related

Sullivan and Chillarege studied error reports for software errors in the MVS OS, between the years 1985-1989 [Sullivan and Chillarege, 1991]. Two main groups of software defects (termed errors henceforth) were analyzed, *regular* and *overlay* errors. Regular errors represents a "typical software error encountered in the field". Overlays are errors where memory areas have been overwritten, such as buffer overruns. The most common types of mistakes were found to be related to memory allocation, copying overruns and pointer management. Other error types identified include register reuse, type mismatch, uninitialized pointer, undefined state, synchronization, deadlock, sequence error, statement logic, data error, compilation errors and others/unknown. The study was later extended to include database management systems and further refined and related to the development process through the concept of *defect type* [Sullivan and Chillarege, 1992]. This gave rise to the classical *Orthogonal Defect Classification* (ODC) process [Chillarege et al., 1992; Chillarege, 1996]. ODC contains seven defect types: function, assignment, interface, checking, timing/serialization, build/package/merge and documentation. Each defect can be classified to belong to one of these types. These types have later been used to build libraries for injection of artificial faults in systems, for instance [Christmansson and Chillarege, 1996; Christmansson et al., 1998; Durães and Madeira, 2006].

2.2.3 User Related

Murphy and Levidow investigated the sources for outages for Windows NT, and found drivers to be a significant source of system crashes [Murphy and Levidow, 2000]. However, system outages are mostly found to be planned (installation of hardware/OS/applications etc). Xu et al. [1999] investigated the causes for Windows NT system reboots and found planned maintenance and configuration to be responsible for 31% of the downtime for a production environment network of servers. Software and hardware to also cause a

significant part of system downtime (22% and 10%).

Even though user-related faults is a key to minimizing outages of system it is not the focus of this thesis. We focus on software related faults and their consequences.

2.2.4 Device Drivers

Device drivers are typically developed by a different party than the OS vendor. However, for some general drivers (bus drivers, for instance) generic drivers may be developed by the vendor. Testing of drivers is therefore normally performed by the device manufacturer. Due to the number of devices produced and the time-to-market pressure, device drivers are often not of the same level of quality as other parts of the OS. Developers may not have time, or be skilled enough, to handle the sometimes intricate interaction required with the OS and the devices. Device drivers typically execute in kernel mode, meaning that a critical fault in a device driver may take down the whole system. Recent efforts have made user-mode drivers possible in many modern OS [Corbet et al., 2005].

Device drivers today form the largest part (in terms of lines of code) within the OS. Chou et al. [2001] reported that 70% of the Linux kernel code is device drivers. That device drivers is a major source for OS failures is therefore no surprise. Several field studies have found drivers to be a main source of system failures, e.g., [Murphy and Levidow, 2000; Ganapathi and Patterson, 2005; Ganapathi et al., 2006].

Ganapathi et al. investigated the causes of kernel crashes for the Windows OS [Ganapathi and Patterson, 2005; Ganapathi et al., 2006]. Crash reports were collected from volunteers using the BOINC (Berkley Open Infrastructure for Network Computing) platform [BOI]. Device drivers were found to be the major cause for kernel crashes, but the study is based solely on crashes, not outages, which may not be due to crashes.

In contrast to the previous studies, which were based on collecting error reports and logs Chou et al. has studied Linux kernels spanning seven years using static analysis [Chou et al., 2001]. The analysis was static compiler based, using the source code of the kernel. They found device drivers to have error rates of up to three to seven times that of other parts of the kernel. Furthermore they found that some functions have distinctly more bugs than others (clustering of bugs) and that newer files are more prone to errors than older ones.

2.3 Fault Injection

Fault injection is a technique where faults (or errors) are intentionally inserted in a system to observe how the system reacts. The technique started in the hardware area, with the specific purpose of testing fault-tolerance mechanisms [Arlat et al., 1993]. It has also been proposed to use fault injection as part of certification for high-assurance systems [Voas, 1999].

Fault injection can be performed at different levels in the system (like hardware, software, protocols) and at different stages in development (on design models, prototypes or deployed systems). The focus in this thesis is on executable systems, i.e., at least a prototype of the system needs to exist for the evaluation to be performed. Furthermore, we focus on techniques implemented in software, so called SWIFI techniques (SoftWare Implemented Fault Injection). Other techniques require use of special-purpose hardware or use abstract models of the system to inject faults. SWIFI has the advantage of being more flexible and cheaper. Surveys of fault injection techniques, covering all three classes, include Clark and Pradhan [1995], Hsueh et al. [1997] and Carreira et al. [1999]. This rest of this section covers a wide range of SWIFI tools.

FIAT (Fault Injection-based Automated Testing) is a fault injection tool for dependable distributed applications [Segall et al., 1988; Barton et al., 1990]. System dependability properties, especially error coverages and latencies were evaluated. Injections were performed by bit-level corruption of a task's data and/or code memory areas. Three types of corruptions were used, zero-byte, set-byte and 2-bit compensate. The outcome of experiments were classified on a five-grade scale, from *machine crash* to *invalid output* and *no error*.

In an early work on fault injection, Chillarege and Bowen introduced the concept of *failure acceleration* achieved through fault injection. Failure acceleration occurs when the fault → error → failure process is accelerated, by decreasing the fault and error latencies, and increasing the probability that a fault causes a failure [Chillarege and Bowen, 1989]. This makes experiments faster to perform and allows for estimations of the transition probabilities (fault → error and error → failure), which is typically not possible from field data (which focus mostly on failures). They reported on a fault injection study performed on the MVS OS, where a random (virtual) page in memory was set to 0xFF, generating an invalid address/opcode, thereby increasing the probability that a fault causes a failure. It was found that only a small fraction of the injected faults led to a complete failure of the primary service of the system (16%), whereas most (70%) led to no loss of service at all. Careful study of the latter category led to the definition of *potential*

hazard, an error which has caused damaged in the system but does not lead to failure under the current operating state. Potential hazards may lead to failure at a later stage, triggered by changes in workload, and may explain previously observed relations between workload and failures.

FERRARI (Fault and ERRor Automatic Real-time Injector) injects errors in application and OS processes simulating low level hardware faults. Injection is performed either by first corrupting the memory of the process before it is started, or by injecting faults during execution, triggered either *spatially* (i.e., after a certain code location is reached) or by a *timeout* defined by the user [Kanawati et al., 1995]. Injections are performed purely in software, using software traps. Faults are injected in the address, data or control line for the targeted instruction, resulting in for instance different instructions being executed or operands being modified. The actual injection is performed using bit-level modifications. Both transient and permanent faults can be injected.

FINE (Fault Injection and moNitoring Environment) was used to study the propagation of errors in OS's. FINE can inject both hardware (CPU, memory, bus) and software related faults (initialization, assignment, condition). FINE was one of the first fault injectors to be implemented in kernel space, making OS evaluation possible. Previous tools, such as FERRARI executed in user-mode and thus had no access to kernel memory areas [Kao et al., 1993]. A new system call was implemented (`ftrace`) used to specify injection and insertion of probes from user-space. [Kao et al., 1993] reports on experiments for SunOS 4.1.2 using randomly placed bit-flips in code memory and randomly selected global variables. Software faults were manually injected in the kernel text segment. Only 8% of the injected faults led to error propagation to another subsystem, with most of them caused by corrupted function call parameters. FINE was later extended for use with distributed systems as **DEFINE** [Kao and Iyer, 1994].

FTAPE (Fault Tolerance And Performance Evaluator) was used to compare fault-tolerant computer systems [Tsai and Iyer, 1995]. The tool combines a fault injector with a workload generator and monitor, to allow injection of faults under high stress conditions, when faults are more likely to propagate [Tsai et al., 1996, 1999]. Faults can be injected throughout the system (CPU, memory, disks). The tool can inject single, as well as multiple bit-flips and stuck-at faults.

FTAPE was later extended to NFTAPE, which is an extensible tool using generic components to perform fault injection in a distributed fashion. So called *lightweight fault injector* components are defined to perform the actual injection, monitor and trigger components can similarly be provided by the user [Stott et al., 2000]. NFTAPE has been used to study error sensitivity

of Linux [Gu et al., 2004].

MAFALDA (Microkernel Assessment by Fault injection AnaLysis and Design Aid) uses fault injection to assess the robustness of microkernel-based OS's [Arlat et al., 2002]. Injections are made into both code and data areas of the OS, as well as into the parameters of kernel calls using the bit-flip error model. MAFALDA can be used to study system failure modes and error propagation across the components of the system. The authors also showed how, using a formal description of function behavior, error detection wrappers can be defined for kernel functions. An extension of the tool, MAFALDA-RT, was designed to also handle real-time systems [Rodriguez et al., 2002].

DOCTOR (integrateD sOftware fault injeCTiOn enviRonment) is a tool for SWIFI for distributed real-time systems [Han et al., 1995]. It can inject communication faults, such as lost or duplicated messages. Hardware faults are simulated in CPU registers, bus or memory as single or multiple bit faults. The location (in memory) can be defined by the user or randomly selected. For driving the experiments various synthetic workloads can be automatically generated or user-defined programs are used. DOCTOR was for instance used to evaluate a distributed diagnosis algorithm implemented on HARTS, a distributed shared-memory-based real-time architecture [Shin, 1991].

Xception uses debugging and performance monitoring capabilities of processors to inject errors in CPU functional units. The processor is instructed to halt when faults are to be injected and low-level exception handling code performs the injection. The advantage of this approach is that interferences with the target system is minimized, it requires no source code access, or trace-based execution of applications or OS. Focus is on simulating hardware transient faults, as these form the majority of faults in modern processors [Shivakumar et al., 2002; Constantinescu, 2005]. Several triggers are supported, including address-based (fetch of opcode from specific address) and timeout-based. Faults are injected as bit level faults (stuck-at, flips and masks). Xception was later used for other studies, including [Madeira et al., 2000, 2002].

PROPANE (PROPagation ANalysis Environment) is a fault injection tool used to primarily study error propagation in embedded software [Hiller et al., 2002b; Hiller, 2002]. Data-level errors are targeted, by modifying data values, either on the bit-level or by fixed values or offsets. Injections are triggered by selecting injection locations. Additionally timers can be set, either using clock counters or counters on reaching injection locations. PROPANE can inject both transient, intermittent and permanent errors. Since instrumentation is done on the source code of the target software, propagation can be studied down to individual signals (variables) of

the components of the system. Together with the measures defined in the EPIC framework (Exposure, Permeability, Impact, Criticality), PROPANE was used to evaluate the propagation of errors in an aircraft arrestment system [Hiller et al., 2004; Hiller, 2002].

RIDDLE (Random and Intelligent Data Design Library Environment) tests application and system services/libraries on Windows NT using random but syntactically correct strings as input [Ghosh et al., 1998]. The program is observed for unexpected termination, crashes, unhandled exceptions etc. The approach taken is similar to that of [Miller et al., 1990] described in Section 2.4.

FST (Failure Simulation Tool) wraps applications running on the Windows family of OS's with an instrumentation layer, whereby failing OS functions can be simulated. On a technical level the wrapping is performed in a very similar manner to the Interceptor modules used in this thesis (see Section 4.5 for more details). Failures in the OS are simulated by throwing exceptions and returning error codes. The faults were selected from the set of outcomes from previous experiments on the system using RIDDLE [Ghosh et al., 1998]. Applications are deemed as robust if they do not hang, crash or disrupt the system in presence of perturbations.

HEALERS (HEALers Enhanced Robustness and Security) is a system for automatically increasing the robustness of C libraries [Fetzer and Xiao, 2002a,b]. HEALERS uses *adaptive* fault injection to evaluate the robustness of individual parameters in library functions. Information present in header files and manual pages is used to build fault injectors, which progressively test functions to compute the *robust argument type*, i.e., the set of values for which the function does not crash or return with an error. This information is used to automatically build robustness wrappers for the selected library functions. HEALERS was later extended (the new tool is called **Autocannon**), using an extended type system from Ballista (see Section 2.4) to further simplify the generation of robustness wrappers [Süßkraut and Fetzer, 2007]. Wrappers are defined as predicates over a set of tests on parameter values, making the approach more flexible and extensible than the original HEALERS approach.

AutoPatch reuses parts of the HEALERS system to investigate applications' handling of error codes from library functions [Süßkraut and Fetzer, 2006]. *Error injection* is used to find unsafe functions, i.e., error code return values are injected for function calls, and applications not handling them (crashing) are labeled unsafe. Unsafe functions can be automatically patched using a variety of patching techniques.

DTS (Dependability Test Suite) tests applications running on Windows NT by corrupting the parameters to library calls [Tsai and Singh, 2000]. The server in a client-server system (Apache IIS, SQL Server) was targeted and

outcomes were classified from a client's perspective, i.e., retry required, server restart required, complete failure etc. The servers were tested running standalone and using two different fault-tolerance middleware solutions. Three bit-level fault models were used for the parameters to the library calls: setting all bits, zeroing all bits or flipping all bits. Large-scale injections verified that the evaluated middleware reduced the number of failures considerably.

G-SWFIT (Generic Software Fault injection Technique) uses software mutations to inject software faults into the binary of a target program [Durães and Madeira, 2006]. A field study of real faults was used to generate mutations. First the faults were classified according to ODC (see Section 2.2.2). This classification is then refined with an orthogonal classification of missing, wrong and extraneous constructs, which allow for more precise fault injection. The binary of the target is searched for patterns relating to higher-level code constructs, where code mutations chosen from a representative set of software faults are inserted. Using failure mode analysis the behavior of three test programs is compared, when inserting low-level mutations and source code faults. Overall most of the source code level faults could be reproduced by the mutations to some extent.

Several hardware-based techniques have been developed as well, injecting faults at different levels of the system. MESSALINE [Arlat et al., 1990, 1993] injects faults at the pins to ICs of fault tolerant systems. Karlsson et. al. [Karlsson et al., 1994] use heavy-ion radiation for validation of fault-handling mechanisms.

Fault injection has mainly been used to evaluate fault tolerance mechanisms or robustness issues. However, it has also been found useful in the area of security, especially for protocols [PRO]. In [Chen et al., 2002] errors were injected in two kernel-based firewalls on Linux, and it was found that errors in firewalls can in some cases lead to security vulnerabilities. Modeling a realistic installation suggests that error-caused vulnerabilities is a non-negligible source for security concerns. Du and Mathur [2000] injected errors in the environment of an application and observed the applications for security violations. NFTAPE has been used to inject control flow bit-flips in the user authentication section of `sshd` and `ftpd` on Linux and it was found that such faults may open up the affected servers for vulnerabilities [Xu et al., 2001]. Neves et al. present the AJECT tool which perform *attack injection* for detecting vulnerabilities [Neves et al., 2006]. Attacks target protocols by varying the messages used.

2.4 Operating Systems Dependability Evaluation

Several past efforts have focused on evaluation of dependability and robustness issues in OS's, including the previously mentioned field studies. This section is dedicated to dependability benchmarks, where a standard methodologies and tools are used to evaluate and compare systems.

In benchmarking the aim is to compare competing systems using a fair and repeatable process. Benchmarks for comparing computer performance are abundant and have found widespread use, even though the interpretation of the results often is non-trivial. One of the most influential benchmarks is the Standard Performance Evaluation Corporation (SPEC) benchmark [SPE; Henning, 2000]. The most well known benchmark from SPEC is probably SPECint for measuring integer computing capabilities of CPUs, but the organization offers benchmarks in many areas, such as graphics, high-performance computing (HPC) and web-based systems. Several other benchmarks exist for specific areas, such as the Embedded Microprocessor Benchmark Consortium's (EEMBC) benchmarks [EEM; Weiss and Clucas, 1999], Transaction Processing Performance Council (TPC) [TPC] and LINPACK [LIN] to mention but a few.

Dependability benchmarks are not aimed at comparing performance (only), but how "dependable" a system behaves. Two well known projects on dependability benchmarking are the Ballista project from Carnegie Mellon University [Bal] and the EU-IST project DBench [DBE; Kanoun et al., 2001]. An introduction to the general problem of benchmarking and specific issues related to dependability benchmarking is given in [Johansson, 2001].

Ballista is a robustness benchmark [Koopman, 1999; DeVale et al., 1999; Koopman and DeVale, 1999]. The first version of Ballista targeted the POSIX interface found on many OS's. It builds a testing wrapper for the function targeted and then automatically builds test cases by selecting parameter values from a set of valid and invalid values for that particular data type. Since the number of different data types used in the POSIX interface is relatively low, the number of types for which injectors need to be specified is also low. Addition of new functions to be tested requires only to define values for any *new type* not previously used, making the approach very scalable. Extensive experimentation done on several OS's revealed multiple robustness issues [Koopman and DeVale, 1999, 2000]. Ballista was later used for testing I/O libraries [DeVale and Koopman, 2001], CORBA implementations [Pan et al., 2001] and for Win32 interfaces in Windows [Shelton et al., 2000].

Mendonca and Neves used fault injection to test functions in the Win-

dows DDK (the interface for device drivers) [Mendonca and Neves, 2007]. Since the DDK for Windows exports more than a thousand functions, only functions used in at least 95% of the drivers were tested. Each function was tested in isolation, similar to tests in Ballista, and failure mode analysis was performed. The failure modes were related, not only to the system robustness (crash, hang etc) but also to consistency of data on disk, where FAT32 and NTFS were compared. Three versions of Windows were compared (XP SP2, Server 2003 and Vista RC1) and the results showed great similarities, indicating that the tested functions have not undergone fundamental changes impacting robustness throughout the three versions tested. It was also found that NTFS, as expected, showed no filesystem inconsistencies, whereas FAT32 did in some cases.

Early benchmarking projects aimed at OS's target UNIX systems. The *crashme* program was developed to test the robustness of the OS by executing random data [Carrette]. This was achieved by first allocating an array and filling it with random data. Then several child processes are spawned that try to execute the data as if it was a code segment. The system is submitted to a large number of such processes, with the intent of testing the error detection and handling capabilities of the OS. This simple test successfully crashed several UNIX systems. The program was later extended to *CMU Crashme* which subjected UNIX system calls to random strings, thereby testing their parameter checking code [Mukherjee and Siewiorek, 1997]. This modified version could crash the Mach 3.0 OS in less than ten seconds. The authors also pointed out the usefulness of modular benchmarks, targeting specific areas of the system, such as file, memory and inter-process communication subsystems for an OS [Siewiorek et al., 1993; Dingman et al., 1995; Mukherjee and Siewiorek, 1997].

Another approach using random data was carried out by Miller et al. [1990], where a series of commercial UNIX implementations were benchmarked and compared. The target was not the UNIX kernel per se, but a set of utility applications commonly included in most UNIX OS's, such as *awk*, *diff* and *grep*. The tests consisted of supplying random strings as inputs to these utilities (which typically work on text input). The technique was named *fuzzing* and has served as inspiration to the area of Random Testing and also to the fuzzing error model used in this thesis. Robustness was measured but observing the behavior of the application, where crashes or hangs were undesirable outcomes (shows non-robust behavior). A surprising number of deficiencies were found, with large differences between the benchmarked systems. The experiments were later repeated with similar results [Miller et al., 1995]. Also studies for Windows NT [Forrester and Miller, 2000] and MacOS have been conducted [Miller et al., 2006].

A dependability benchmark for fault-tolerant systems was developed in [Tsai et al., 1996] using the FTAPE fault injection tool. Measured was the number of catastrophic incidents and the performance degradation of the system. Faults were injected into the CPU, memory and I/O components of the system. Since the systems tested consisted of redundant computers (Triple Modular Redundancy - TMR) the expected outcome is only a performance degradation.

Using PostMark^{TM}, a file system performance benchmark, as workload, Kanoun et. al. developed a dependability benchmark for several versions of Windows and Linux [Kanoun et al., 2005]. The benchmark is defined as a measure of the robustness of the OS's ability to withstand invalid API inputs. Measured is also reaction and restart times for the compared OS's. The same authors have also defined a dependability benchmark using the TPC-C transactional performance benchmark as workload [Kalakech et al., 2004a,b] for Windows.

Brown et. al. argue that the human factor must be included in a dependability benchmark by showing that most of the outages for large systems are caused by (human) operators [Brown and Patterson, 2001]. They present a dependability benchmark which includes real human operators, performing both detection and recovery actions to injected faults and to perform standardized maintenance tasks [Brown et al., 2002]. Vieira and Madeira also consider operator faults for studying recovery procedures in database management systems (DBMS) [Vieira and Madeira, 2002a,b]. A portable faultload for DBMS's is defined in [Vieira and Madeira, 2004], used to form a dependability benchmark for On-Line Transaction Processing systems (OLTP). Two kinds of measures are used, performance-related (from TPC-C) and dependability-related. Performance related measures were taken both with and without faults injected. Dependability-related measures include data integrity and availability measures. The benchmark was applied to four different DBMS's and three different OS's, and comparisons were made across them.

2.5 Other Techniques for Verification and Validation

This section presents complementary techniques for verification and validation and their relation to the fault injection-based techniques used in this thesis. In general, no one single technique is to be preferred and we emphasize that also the proposed techniques must be used as complements to other

verification and validation techniques.

2.5.1 Testing

Software testing is the most basic form of verification for software. The developer (or a designated tester) builds a *test case* which defines the context for the test and the inputs, as well as the expected outputs, based on the specification of the tested component. The test is executed and the result compared to the expected result. A plethora of testing techniques exist and in this section we highlight the more relevant ones to this work. Good introductory texts to software testing includes the seminal book *The Art of Software Testing* by Myers [2004], or *Testing Computer Software* by Kaner et al. [1999].

From an implementation point of view testing and fault injection have many commonalities. Especially fault injection focusing on interfaces, which is the case in this thesis. This type of fault injection resembles widely spread unit testing approaches, such as equivalence class testing or boundary value testing [Myers, 2004].

Conceptually, software testing's goal is to identify faults, i.e., bugs, whereas the goal of a robustness evaluation is to identify weaknesses. A weakness in this sense may not be a bug (although it might), because it may lie outside of the scope of the specification, or may arise only in a certain context.

In equivalence partitioning testing of a function one typically focuses on both valid and invalid classes of inputs. The input space is split into a set of equivalence classes where it is assumed that the function behaves similarly (in terms of correctness) for all values within a class. The specification for the input is required for performing the partitioning. Boundary-value testing can be seen as an extension of equivalence partitioning testing where one focuses on the values around the boundaries of the equivalence classes.

2.5.2 Formal Methods

Any form of fault injection is inherently a dynamic testing method [Myers, 2004]. We consider dynamic testing techniques as ours, and more formal techniques, including static analysis (like [Ball and Rajamani, 2002]) as complements. Both are useful for building more dependable systems and both have their strengths and weaknesses. Formal proofs (like theorem proving) are used to *prove* that the implementation is made according to the specification, which also needs to be expressed formally. Another approach is to using a model of the system, build and check all the states of the system and

verify that they do not violate the specification, for instance lead a violation of the safety specification for the system [Kumar and Li, 2002].

[Hayes and Offutt, 2006] uses static analysis of the user input specifications to programs to both identify inconsistencies in the specification itself and to generate test cases for testing. A large empirical case study showed that the automatic tool found defects faster than expert testers, but not necessarily more. It also found defects not found by human testers at all. The results supports the complementary use of automatic tools with domain expertise.

Even though formal methods are theoretically attractive, since they offer means to reach completeness, testing techniques and related experimental techniques like fault injection are likely to prevail for many years to come, due to their ease of use and understanding. However, test automation is a necessary evolution in testing, as systems grow larger and more complex.

2.6 Summary

This chapter has presented background information and reviewed related research within the areas of OS robustness evaluation and fault injection. On this background we have identified the OS as being the key to system dependability and robustness since it is the platform on which applications and services are built. Furthermore, device drivers were identified as the main source of software-related causes of system failures. Several previous studies have focused on interfaces (OS-Application and OS-Driver) as it facilitates portability and fair comparison, important aspects of benchmarks. Fault injection has in multiple previous studies been shown to be an effective means for evaluation of dependability attributes of OS's.

Our report on related work does not stop with this chapter. Throughout the thesis we will give pointers to relevant studies where appropriate.

Chapter 3

System and Error Model

What are the system boundaries, and what is an error?

OS's are key building blocks in virtually all computer based system, ranging from small deeply embedded control systems, to desktop workstations and large servers for online transactions. Consequently OS dependability is an important objective and a prerequisite for dependable provision of services.

This chapter builds the foundation for the following chapters, starting by presenting the system model used. Then a general error model is defined, in terms of location, type and trigger, followed by the presentation of the three error models used in the following chapters. The experimental setup used is presented, both in terms of hardware and software. The chapter is concluded with a summary containing a table of the symbols introduced for reference in later chapters.

3.1 System Model

Most modern OS's are monolithic, i.e., the OS kernel providing the most basic functionalities runs in kernel space, as illustrated in Figure 3.1. This is in contrast to, for instance, micro kernel-based OS's, where the functionality of the OS kernel is spread across multiple subcomponents with well specified interfaces.

We use a generic model of the OS. Similar to many other studies, e.g., [Albinet et al., 2004; Durães and Madeira, 2003], we model a monolithic system. The model consists of four layers: applications, OS, drivers and hardware platform. We have chosen this model as it is generic enough to apply to several commercial OS's, like Windows or Linux. It is also sufficient for measuring the robustness of the system due to errors in drivers by studying error propagation.

Each layer consists of one or more subcomponents (like different applications in the application layer, or different drivers in the driver layer). Our model does not specify the subcomponents required in each layer since they differ for each specific OS. Each layer provides *services* to be used by neighboring layers. A service can be realized in many ways. Common is for instance function calls (like API's, Application Programming Interfaces or system calls), but in general other mechanisms could be used like the message passing paradigm used for communication between the OS and the drivers defined in the Windows Driver Model (WDM) found on Windows XP [Oney, 2003]. The nature of the communication is not important for the model, important is that the service is syntactically specified, and that the flow of information can be intercepted and modified. This is required to be able to inject errors and to observe the outcome of each injection. The specification is defined in an *interface*. The two interfaces of interest here are the *OS-Application* and *OS-Driver* interfaces indicated in Figure 3.1.

The system has a set of n applications, $\mathrm{APP}_1 \ldots \mathrm{APP}_n$. The application set includes all applications running on the system which are not required for the OS to function properly. This includes applications added for the purpose of the evaluation, called benchmark applications, or test applications. Applications make use of OS-level libraries to implement their functionalities. Typically, applications run in user space and the OS and the device drivers execute in privileged mode.

The OS layer includes the OS kernel and all required libraries delivered as parts of the OS. An example of such libraries are libraries used by applications to interface with the OS (POSIX, C-runtimes etc.).

The OS provides a set $\mathcal{S}$ of services to be used by applications (s_i's in Figure 3.1). The OS-Driver interface consists of services provided both by

the OS ($os_{x.y}$'s in Figure 3.1) and services provided by drivers (the $ds_{x.y}$'s in Figure 3.1). Collectively they are referred to as the set $\mathcal{O}$ of services in this interface. Each application APP_x uses a set of OS services, termed $\mathcal{A}_x$, where $\mathcal{A}_x \subseteq \mathcal{S}$

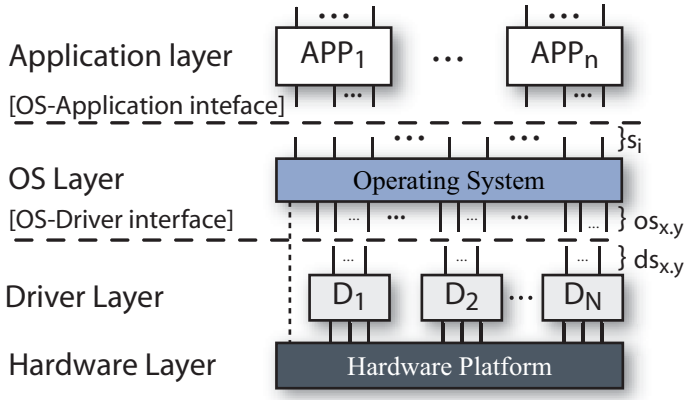

Figure 3.1: System model.

A driver is modeled as a component having both import and export interfaces as illustrated in Figure 3.2. The exported interface consists of a set of services that the OS calls to request the driver to perform operations. These services are termed $ds_{x.y}$ for the y^{th} service provided by driver D_x. The imported service interface is used by the driver to accomplish these requests and can be from the OS itself or other libraries in the system. These services are termed $os_{x.y}$ for the y^{th} service imported by driver D_x. When no distinction is made between imported and exported services we term a services at this interface $s_{x.y} \in \mathcal{O}$. In the target environment used in this thesis (Windows CE .Net) a service corresponds to a function call.

To perform error propagation analysis we require sufficient access (with specification) to the system to be able to intercept information flow in the two interfaces defined (OS-Driver and OS-Application). In most cases this can be achieved without requiring access to source code, neither for the drivers, nor for the OS itself. For Windows CE .Net no such access is required. However, access is needed to the source code of the benchmark applications, for instrumenting them with assertions used to track the outcomes of injections. The availability of interface specifications is a basic requirement for any OS open for extensions by new types of drivers/applications.

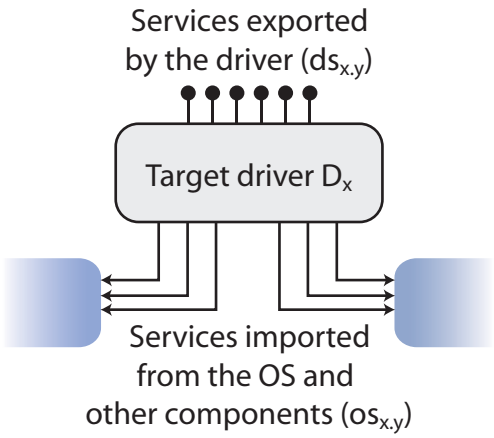

Figure 3.2: Driver model.

3.2 Error Model

In order to conduct fault injection based experimental stressing, three questions arise, namely:

- Where to inject?
- What to inject?
- When to inject?

The answers to these questions correspond to three properties of an error model, referred to as the *error type*, *error location* and *error trigger*. Another, fourth property, related to the error trigger is for how long to inject. Each of these properties of an error model is discussed in the following subsections.

Throughout this thesis we do not make any distinction between the terms *errors* and *faults*. Consequently, we will use *error* when discussing the perturbations inserted in the system. When the distinction is needed, we will explicitly use the term *fault*.

3.2.1 Error Type

The error type constitutes the *nature* of the error. The error type relates to the origin of the error, i.e., the fault, but also to the manifestation of faults as errors. The error type describes how an error changes some internal state of the system, from the originally (assumed) correct state, to another, possibly erroneous, state.

Depending on the goal of the evaluation, error types are chosen either to as closely as possible match errors expected to appear in the system as it is deployed in the field, or generic error types are used, based on their ability to provoke the system such that weaknesses in handling perturbations are discovered. As our interest is on robustness , i.e., how the system handles external perturbations, our goal is to use models that provoke as many and as diverse vulnerabilities as possible. It can also be argued that when the purpose of the evaluation is comparative, as is the case here, the value of using a realistic error model is decreased, assuming that the relative effect of different models is the same [Hiller, 2002]. Chapter 6 studies the selection of different error models explicitly. As previously noted, robustness evaluation can also be a means for finding security-relevant vulnerabilities.

Fault injection originates from the desire to estimate the effectiveness of error detection and recovery mechanisms (EDRMs) built into a system. For this purpose one chooses to use the error model used for the design of these mechanisms. The second type of evaluation is explorative in nature. Without knowledge of presence or coverage of any EDRMs, the system is evaluated to see how it handles the perturbations injected. This type of evaluation can be guided by the need to explore extra-functional behavior of the system (or lack thereof) or by lack of operational scenarios. The second case is especially true for general purpose system components, such as OS's, which may be used in many, fundamentally different, operational contexts.

The focus of this thesis is on exploration of robustness vulnerabilities of OS's. To this end we have chosen error types based on their usefulness in other research projects as well as real-world projects as reported in literature. Three main error models have been used: data type-based errors, bit-flips and fuzzing (random values).

An error appearing at the interface of a component (such as a device driver) appears as a *data level error*, i.e., the (data) value of some parameter used in the interface has an erroneous value. What constitutes erroneous depends on the interface/parameter in question and the state the system is in. For instance, a driver returning an erroneous "busy" value may only cause a small delay for the overall system (provided that a retry mechanism exists). A driver responding "ready" when it in fact is not ready to receive commands may cause severe failures in the system.

There are many possible sources for erroneous values to appear at the interface, such as propagating hardware errors, faulty assignments of variables in the driver code, wrong user inputs or concurrency problems. ODC is a framework to classify software defects and many of them can manifest as interface errors (one class in ODC refers specifically to interface defects) [Chillarege, 1996]. As we model the effects of faults, i.e., errors, the interface

level errors model the effects of many of the underlying faults (and consequently ODC classes of defects) as propagating errors. However, as we focus solely on data level errors for device drivers at the interface to the OS we do not offer complete coverage of all operational faults. Therefore, our approach should be treated as a complement to other techniques.

All three models used represent data level errors, but differ in the implementation and the level of semantic expressiveness. The three models will now be presented one by one.

Data Type Error Model

The manifestation of a data error also depends on the data type of the parameter in question. Since modern compilers contain type checkers, not all assignments are possible to do for a parameters, restricting the set of possible errors for the parameters. The erroneous value for data type (**DT**) errors are therefore selected depending on the data type of the parameter in question. As most device drivers are written in the C programming language we will consider C-style data types. This excludes high level abstract data types supported in other high-level languages, such as classes/objects in object-oriented programming languages.

Some OS's do provide the possibility to write device drivers in other programming languages, for instance C++. However, since most drivers are still written in C we focus on such interfaces. In principle, object-oriented interfaces can be seen as extensions of the data structures used (we already support the `struct` data structure) in C.

For each data type used a set of injection cases are defined. These are predefined, before injection, and are chosen based on their effectiveness in exposing vulnerabilities in the system [Koopman et al., 1997]. Values include predefined (no randomness involved) test values, offset values and boundary values. Offset values modify the original value, for instance using addition or substraction operations on the original value. The number of injections defined is typically relatively low, allowing this error model to incur fewer injections (on average) than, for instance, the bit-flip model [Koopman et al., 1997]. Chapter 6 discusses the number of injections required compared to other error models.

Since the injection cases are defined on a data type-basis the number of such data types used becomes the scaling factor for the data type error model. Typically, multiple services in such interfaces use the same data types. For instance, in [Kropp et al., 1998] only 20 data types were used for the 233 POSIX functions targeted. Each of the 233 functions used a combination of the 20 data types, making this error model scale very well with the number

of functions. No specialized injection scaffolding is required for each tested function. However, one caveat is that information on the data type used is required to select the right injection cases.

Table 3.1: Overview of the data types used.

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 3.1 shows an overview of all the data types used and the number of cases implemented for each of the types. The injection cases were chosen based on their reported used in literature, such as the Ballista project [Bal] and to include cases modifying the original value. Not relying solely on statically defined values, like boundary values and special values allows exploration of "close to correct" values, which may be very problematic to detect and recover from. Furthermore we have only selected values which *can* occur in real code. It is important to be able to match the injected error to a hypothetical fault in the code. Since we simulate mostly software faults, each error injected must have been possible to introduce by an implementation fault. Consequently, each injected error must be compilable, i.e., it must pass the type check by the compiler. By having a specific injection case for each data type this property is maintained.

Data type errors also treat pointers as a 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. This done for *explicit* pointers, but not for *implicit* pointers, such as strings, which have this case

Table 3.2: Data type error cases for type `int`.

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

Table 3.3: Data type error cases for strings.

Case #	New value
1	Overwrite end of string ('\0')
2	Increase reference pointer
3	Replace with empty string
4	Set reference to NULL

defined as a special injection case. To further illustrate how data type errors are defined, Table 3.2 shows the cases for the type `int`. Cases 1 and 2 modify the original value by adding an offset to it. Cases 3-7 use commonly difficult values and boundary values. Table 3.3 shows the errors injected for string parameters (both for Unicode and ASCII strings). The first case effectively evaluates the reliance on the end character for strings. The second case shortens the string by disregarding the first character and the third case replaces the entire string with an empty string. The last case sets the pointer to the string to `NULL`.

The choice of values was done based on known problematic values, and previous studies; and it was kept relatively low. The choice of effective values (those exposing vulnerabilities) is difficult and context dependent, and is similar to the problems arising when selecting suitable equivalence classes for functional testing [Hamlet, 2006]. For this study we have therefore opted for a simple and lightweight data type model. Our model does not have the same expressive power as the ones used for instance in [Koopman et al., 1997] or [Fetzer and Xiao, 2002b], as it is based solely on the data type of the parameter. The in-situ injection strategy effectively limits the possible types of injections that can be carried out. The model used is chosen for its simplicity and low number of injection cases.

Bit-Flip Error Model

When hardware elements are exposed to, for instance, radiation or EMI the voltage levels in transistors can change, causing the logical ones and zeros to change values or even get stuck at a certain value. The bit-flip model (**BF**) simulates these types of faults in hardware, by selectively flipping certain bits, changing the value from one to zero or vice versa.

The **BF** model was first introduced to simulate hardware errors as above, and was used in multiple fault injection tools (see Section 2.3 for several examples of such tools). At first, hardware-based injection tools were used, but SoftWare Implemented Fault Injection (SWIFI) soon emerged, where hardware faults are injected using software mechanisms. SWIFI improves flexibility and ease implementation (no special hardware components needed), but may be limited in which areas of hardware can be targeted. Once injection could be conducted using software, bit-flips were soon also used to simulate software faults [Voas and Charron, 1996]. There is still a debate whether the **BF** model accurately reflects software faults. Some authors argue that this relation is of lesser importance, especially for robustness evaluation, and that the important question is whether the *effects* of the injected faults (the errors) are the same as those of real faults [Jarboui et al., 2002b].

In the **BF** model each parameter is seen as a data word, where selected bits are flipped to simulate faults in the module. A difference is made between single event upsets (SEU, also referred to as *soft errors* in discussions on hardware reliability) where only one bit is flipped, and the case where multiple bits are flipped. In this thesis focus is on the simpler SEU model, where one of the bits is selected as target and flipped. For a 32 bit architecture this typically results in 32 injections per parameter. Some data types do not use the full 32 bits and thus a smaller number of bits can be used.

The greatest advantage of the bit-flip model when used on interface parameters is simultaneously its greatest weakness, namely simplicity. While being very simple to implement it lacks in expressiveness for more complex errors in abstract data types, such as strings etc.

Fuzzing Error Model

The first use of the fuzzing error model (**FZ**) in the context of robustness evaluation was reported in [Miller et al., 1990]. Here UNIX utility programs were fed random input data and their behavior was observed. The technique of random input for robustness evaluation was further developed in [Ghosh et al., 1998; Oehlert, 2005; Godefroid et al., 2007] and is advocated for instance by Microsoft as part of their Secure Development Lifecycle [Howard

and Lipner, 2006] and is mainly focused on files and network protocols.

For interface fault injection, fuzzing translates into replacing the value of a parameter in the interface with a random value. The random value is uniformly selected across all 32-bit values. The uniform distribution is selected since no knowledge regarding operational profiles or equivalence classes of parameter values are present (or assumed), which could justify using a different distribution. More sophisticated techniques can also be applied, as in the area of random testing (see Section 2.5.1). As true randomness is difficult to achieve, pseudo-random generators are used. Care needs to be taken to make sure that the seeds used to these generators are selected differently for each experiment, else the same value will be chosen for each injection.

It is important to note that whereas **BF** (and in some cases also **DT**) modifies a given value, i.e., the new erroneous value depends on the original (presumably correct) value, fuzzing completely replaces the value with a new one. This means that **BF** can be expected to more effectively test "close to correct" values, whereas **FZ** is better (thanks to the random selection) at finding more rare values causing vulnerabilities.

3.2.2 Error Location

As location and distribution of actual faults may be unknown, or too costly to fully explore, a common approach to inject *errors* instead of *faults*, i.e., to inject the consequences of activated faults rather than the faults themselves [Barton et al., 1990]. Many faults may manifest as the same error, i.e., at the same location/level in the system. This concept is illustrated in Figure 3.3. An injected error may represent multiple faults (at 2a or 2b), originating at different locations. An error injected at the interface (at 3.) may therefore represent multiple errors having propagated to the same location.

Jarboui et al. [2003] makes a distinction between the level in the system where a fault is injected and the reference location of the originating fault, *dr*. In Figure 3.3 this corresponds to the distance between point 1 and 2. Furthermore, a distinction is made between the location of the injected error and the level where the failures of the system are observed, *do*. In Figure 3.3 between points 1 and 4.

In this thesis we have focused on errors appearing in the interface between the OS and its device drivers. For the purpose of robustness evaluation of an OS, this interface represents a good location for injecting errors for the following reasons:

- This is a standard interface defined for the OS. Using a standard interface facilitates fair comparisons across drivers for the same OS.

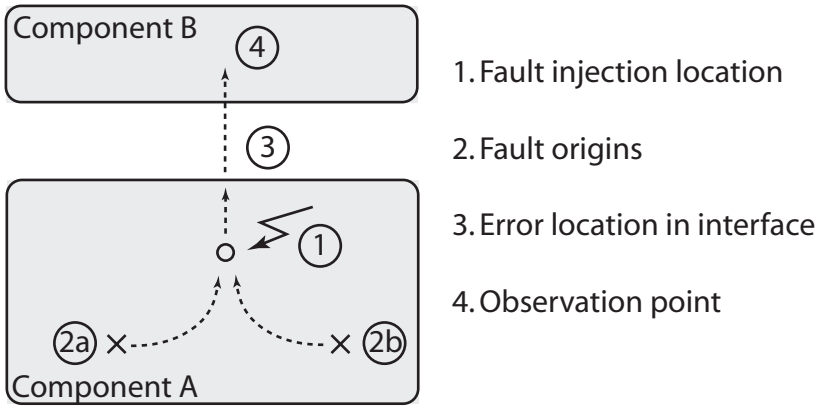

Figure 3.3: Error manifestation example.

- The interface allows for low-intrusion interception of the calls being made across the interface. Low intrusion means that no source code changes are needed, neither to OS, nor to drivers, which may not be available for commercial products.
- As described above, injecting errors in the interface between the driver and the OS allows for simulating multiple faults within the driver and in the hardware it controls.
- Injecting errors at this level allows for achieving 100% activation ratio for the injected errors using pre-profiling. This process is described in Section 4.6.
- No driver-specific knowledge is needed, making the approach readily available also for non-driver experts.
- The interface is an *open* interface, in that other 3^{rd} party developers are given access to the full interface, making robustness a key issue for the vendor of the OS.

3.2.3 Error Trigger

An error can be a permanent defect present in the system, or a combination of defects and external perturbations that together lead to an error. This gives rise to two distinct properties of an error, relating to timing: the event triggering the appearance of an error and the duration of the error once it appears.

For hardware related errors, permanent errors are not uncommon, even though recent research indicates that the ratio of transient errors is increasing. For software, permanent errors (Bohrbugs) are targeted using testing. For robustness evaluation the main target is Heisenbugs. Furthermore, the very nature of Heisenbugs make them difficult to find since a simple relation between inputs and failure cannot be established. Therefore, they are instead simulated by the injection of errors in the system. In this work we have focused on a transient error model as we believe this to more closely represent behavior by the drivers not found through standard testing routines. Multiple other fault injection tools allow for injection of both transient, intermittent and permanent faults, e.g., [Han et al., 1995; Stott et al., 2000].

The trigger used for an injection is studied in depth in Chapter 7. Which type of trigger to use (event- or time-driven) and which parameters to use is a non-trivial task. This thesis proposes a novel event-driven approach, resulting in an effective, yet simple, process for the selection of triggering events. The proposed approach is based on the usage profile of the drivers, which allows injection in different states of the system.

3.2.4 Other Contemporary Software Error Models

The work reported in [Albinet et al., 2004; Durães and Madeira, 2003; Arlat et al., 2002; Gu et al., 2004; Jarboui et al., 2002a] explored the use of various error models and injection techniques for OS robustness evaluation and benchmarking. In [Jarboui et al., 2002a], for instance, error models similar to ours are used, but are injected at different levels within the Linux kernel. Durães et. al. use a code mutation error model, where code segments of device drivers are targeted [Durães and Madeira, 2003]. Mutations have long been used to assess the effectiveness of testing. For instance, DeMillo used code mutations for investigating the efficiency of a set of test cases in discovering flaws of a piece of software [DeMillo et al., 1978]. The authors develop a theory regarding the coupling effect, namely that if a set of test cases (input values) can distinguish all (simple) mutations of a program from the correct one, then it will also detect more complex faults in the code. Mutations were later used in fault-based testing (for instance [Zeil, 1983; Morell, 1990]) to verify that certain code level errors are not present in a piece of software.

Finally, in [Moraes et al., 2006] the authors note that errors appearing at interfaces of components, though being useful for robustness evaluation, do not necessarily represent faults in the code. Since we are indeed focusing on robustness, interface errors are relevant. However, continued research on error propagation can hopefully reveal which errors can be represented at the interface, and which not.

3.3 Experimental Environment

The experimental environment detailed in this section was chosen to represent a commonly used OS. We chose Windows CE .Net because it provides the possibility to customize the whole system image in an easy manner and it is a wide-spread OS, with uses in a wide range of products. Furthermore, its architecture resembles that of most modern OS's, making it an excellent case study. This section first introduces Windows CE .Net , its architecture and tool support. Then the hardware setup used is presented together with a description of the software setup.

3.3.1 Windows CE .Net

Windows CE .Net is an OS from Microsoft, targeted mainly at the embedded market. It is highly configurable, making it widely used in different configuration in diverse products, such as mobile phones and PDAs, Internet surf stations, Point-of-Sale stations, GPS navigators etc. Windows CE is the foundation for more specific embedded OS's from Microsoft, such as *Pocket PC* and *Windows Mobile*.

The first released products based on Windows CE were released in 1996 as so called *Handheld PCs*. Further revisions of the first version has led to the currently latest version 6.0 at the time of writing this thesis. The work in this thesis is based on version 4.2 of the OS, called Windows CE .Net, which was released in 2003. For reasons of continuity we have chosen not move to a newer version. Therefore, the rest of this thesis describes version 4.2 of the OS. A good introductory text book on programming for Windows CE is Douglas Boling's book *Programming Microsoft Windows CE .Net* [Boling, 2003].

Figure 3.4 shows an overview of the architecture if Windows CE .Net. It shows how the OS layer is split in two parts, the generic OS layer provided by Microsoft, which forms the interface used by applications, and the *OEM layer* which is provided by the OEM (Original Equipment Manufacturer) embedding Windows CE in a product sold to customers. The OEM layer makes it possible to use Windows CE for many hardware architectures and for a multitude of peripheral devices and technologies. Although device drivers are the main responsibility of the OEMs, some generic drivers are also provided by Microsoft as part of the OS package.

In relation to our system model (Figure 3.1) the Driver Layer is formed by the drivers provided either by the OEM or Microsoft. The rest of the OEM layer is considered as part of the OS Layer.

Windows CE .Net, although being a completely different OS than other

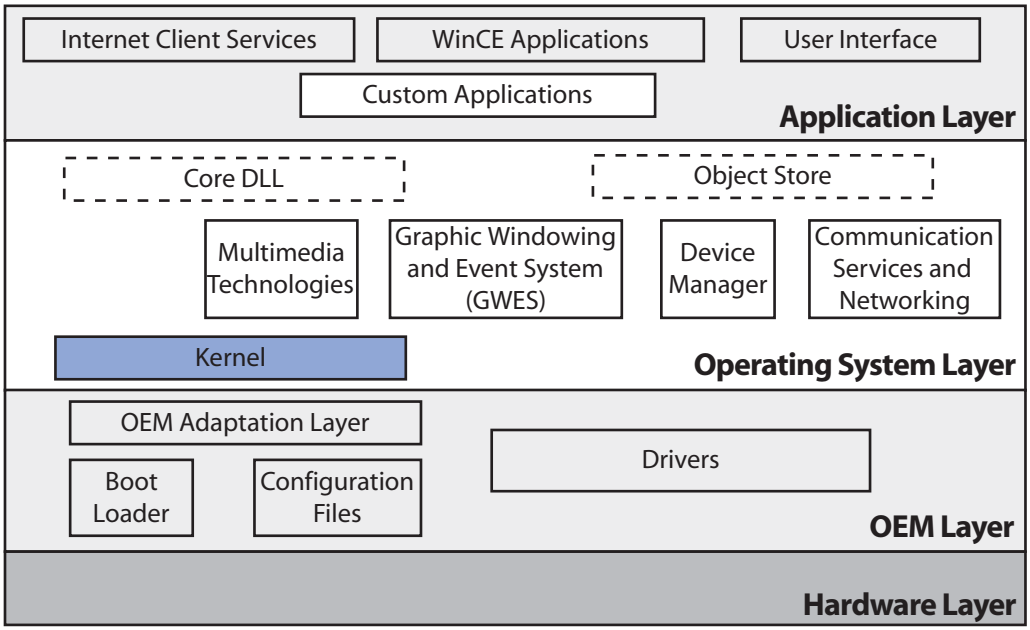

Figure 3.4: An overview of the architecture of Windows CE .Net. Figure adopted from [MSDN].

OS's produced by Microsoft, offers many of the same services and interfaces used on the Windows platform. Examples include the .Net platform (as a subset known as .Net Compact Framework), Win32, MFC etc. At design time the designer can choose which components to include in the OS image. A component is a separate piece of functionality that can either be included in the OS image or not. The designer uses the Platform Builder tool to build the OS image and to download it to the target machine.

3.3.2 Device Drivers in Windows CE

A device driver for Windows CE .Net is a dynamic link libraries (Dll). It is dynamically linked into another process at load time. This host process can then use the services provided by the driver. Most drivers in Windows CE .Net are loaded by the device manager (`device.exe`). The Registry[1] is used to specify which drivers are to be loaded in the system, and in which order (if there are dependencies across drivers the order might be important).

The interface used for communication between the OS and the driver

[1]The Registry is a Windows specific technique to centrally store configuration information, both for the system and for applications.

is defined in the C programming language. Each driver exports a set of services and uses (imports) service from the OS to perform service requested. Applications access devices through the OS, for example through the file system interface. These calls are translated to corresponding calls into the driver.

Since Windows CE .Net is supported on many hardware platforms and used mainly for embedded systems it supports many different peripheral devices. Windows CE .Net supports three basic types of drivers, native, bus and Stream interface drivers. Native drivers are built-in drivers provided by the hardware vendors. They are typically tied to specific hardware and OS versions, for controlling things such as keyboards, touch screens etc. They might use completely custom interfaces to the OS and therefore often require changes when new versions of the OS are released. Native drivers can be seen as extensions of the OS for the required hardware, rather than supporting add-on devices.

The most common type of interface is the Stream interface, which provides standard entry points for a driver. Since the interfaces is well specified it allows for 3^{rd} party developers to build drivers of the OS. Table 3.4 shows an overview of the entry points provided. The prefix COM is by convention used for serial drivers, other drivers use other prefixes, such as CON for console drivers or WAV for audio wave drivers.

Table 3.4: Stream interface for serial driver.

Number	Name
0	COM_Init
1	COM_Deinit
2	COM_Open
3	COM_Close
4	COM_Read
5	COM_Write
6	COM_Seek
7	COM_IOControl
8	COM_PowerDown
9	COM_PowerUp

We focus mostly on Stream interface drivers, as these use a standard interface allowing fair comparison across drivers, are typically developed by 3^{rd} parties and represent add-on peripherals where a choice among competing products can actually be made.

3.3.3 Hardware

The hardware setup used for the experiments presented in this thesis is an XScale-based reference board produced by Intrinsyc Ltd [Int]. Multiple boards were acquired to allow for parallel execution of the injection experiments.

The boards are based on the Intel PXA250 architecture, with an Intel XScale processor chip. Each board carries 64 MB RAM and 32 MB Flash-based ROM. A boot loader is present in flash and allows for simple download of new OS images, either to the ROM, or to RAM for immediate boot. A dedicated flash chip is also present, with access from user space applications.

The boards are equipped with a set of serial port connections (standard RS232), where one acts as debug port. Two standard Ethernet sockets (RJ45) allow for network connection. Each board is also equipped with a CompactFlash socket, where peripheral devices can be attached onto the PCMCIA bus.

3.3.4 Software

Using the provided Platform Builder tool, a small-footprint image containing the OS and the associated software modules described in Section 4.5 is built and downloaded to the target board using Ethernet. Starting with the smallest supported image, only components for the desired drivers and the hardware specific components supplied by the vendor were included. This resulted in an image with a footprint of less than 3 MB. Since the boards used are headless, only minimal graphics and windowing components are needed. Also media related (e.g., readers and viewers of different file formats) and Internet components (web, telnet, and ftp servers etc.) are left out of the image. This way we get a system which contains a minimum number of components that may influence the result of the experiments.

3.3.5 Selected Drivers for Case Study

Three drivers were chosen as representative for a case study: a serial port driver (`cerfio_serial`), a network card driver (`91C111`) and a driver for accessing a CompactFlash card connected to the PCMCIA bus (`atadisk`). These drivers were chosen as they represent different classes of drivers, the first two are common types of communication and the third access to external sources attached to the system. The first two are supplied, in this case, by the vendor of the development board (not the same as the producer of the hardware circuits), whereas the CompactFlash driver is delivered as part of

the OS. They also represent drivers typically found on many systems and platforms. All three provide the required Stream interface entry points and are loaded by `device.exe` at load time.

3.4 Summary

This chapter has introduced the preliminaries needed for the discussion in the following chapters. The system model used was introduced and for reference Table 3.5 provides an overview of the symbols defined. The error models used were introduced and discussed followed by a description of the experimental environment used, including both hardware and software aspects.

Table 3.5: Summary of symbols introduced.

Symbol	Description
APP_i	Application i out of a total n applications running on the system
D_x	Driver x from a total of N drivers in the system
s_i	A service provided by the OS, to be used by an application.
$os_{x.j}$	A service provided by the OS, used by driver D_x.
$ds_{x.j}$	A service provided by driver D_x to be used by the OS.
$s_{x.y}$	Any service in the OS-Driver interface, disregarding the difference between imports and exports.
$\mathcal{S}$	The set of OS services provided in the OS-Application interface.
$\mathcal{A}_x$	The set of OS services used by APP_x.
$\mathcal{O}$	The set of services in the OS-Driver interface.

Chapter 4

Fault Injection Framework

How to perform fault injection for device drivers?

In order to provide the infrastructure and support needed to perform the required experiments a fault injection framework has been implemented for Windows CE .Net. The framework allows for injection of a varied set of error models in the interface between the OS and its device drivers. This chapter first discusses the requirements on the injection framework and then describes the architecture of the framework, its implementation for Windows CE .Net and the extension possibilities it provides.

4.1 Introduction

When performing research using fault injection a flexible environment is needed, such that new ideas can easily be pursued, without extensive redesign of the underlying tool[1]. The framework should support automatic configuration, such that management of configuration settings is simplified and the risk of mistakes is minimized. It should also simplify the collection, storage and analysis of data.

A plethora of fault injection tools exist in literature (see Section 2.3 for a comprehensive list). However, even though some of the tools may have been able to adapt we decided to implement a new injection framework to better suit our needs. For the implementation several requirements were postulated, which allow for a flexible fault injection environment:

- **Extensibility:** Injection of multiple drivers, using multiple error models should be supported. The per-driver scaffolding should be minimized.

- **Black-box:** No access to source code should be assumed, to allow external evaluators the possibility to use the tool.

- **Data handling:** The data extracted from the experiments should be processed and stored without loss of information, and allow for easy extension and interoperability with external tools.

- **Automation:** Automation is key to a) reduce the time overhead associated with configuring and running fault injection experiments; b) to minimize the risk of user mistakes in configuring the experiments; and c) to more easily adapt to changes in the setup requiring configuration changes.

These design requirements were the basis for the design of our fault injection framework. Even though one of the goals is extensibility we have chosen to implement the framework for a specific OS, Windows CE .Net. Extending the framework for other OS's is part of future directions. The rest if the chapter describes the overall design of the framework, the physical setup used and the functionality of the system components.

[1] As our tool is flexible and supports extensions we will refer to it as a *framework* for fault injection.

4.2 Evaluation, Campaign & Run

To simplify the discussion we make a distinction between an *evaluation*, an injection *campaign* and an injection *run*. An injection run is the injection of a specific error and the observation (and logging) of the effects of the injection. An injection campaign is a collection of injection runs, pertaining to, in our case, a specific driver, error model and interface (Dll, imported/exported). A set of injection campaigns form the basis for the evaluation of a system, including possibly multiple drivers and error models.

4.3 Hardware Setup

The target system for the evaluation runs on a dedicated computer, the *Target Computer*. As already mentioned we use special development boards for the evaluation. The evaluator controls the evaluation from a normal PC workstation (in our setup running Windows XP SP2).

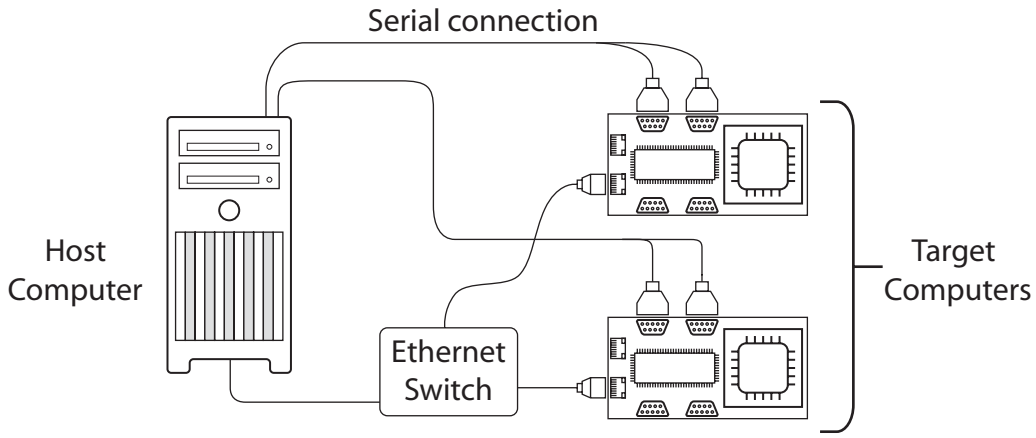

Figure 4.1: The hardware setup. Using an Ethernet switch to connect each development board to the private network. Each board is also connected directly to the Host Computer via serial cables.

Figure 4.1 shows the hardware setup, with two target boards connected. Each board is connected to a private network using an Ethernet switch, which provides dynamic IP addresses using a built-in DHCP server. Each board is connected to the Host Computer over a debug serial connection to access the

boot menu of the boot loader and to read debug output. Additional serial connections are used by each board's workload, as described below.

4.4 Software Setup

The binary image downloaded to a board for each experiment campaign contains all necessary software components required for performing the experiments. This includes all components comprising the system depicted on the target computer side of Figure 4.2.

Platform Builder together with the Embedded Visual C++ 4.0 tool were used to compile and build the applications and OS images used. Most of the code for the components detailed in Section 4.5 was written in C++, or in some cases the C programming languages. Furthermore SQL Server was used on the Host Computer (Windows XP workstation) to store the experiment data. Applications running on the Host Computer are written either in C# (.Net) or C++.

4.5 Injection Setup

Each injection is specified using three (integer) parameters: service ID, parameter number and injection case number. The service IDs can be selected in any order, as long as each service for each campaign is uniquely identified. When storing data in the database on the Host Computer, each service obviously has to have globally unique identifier. Parameters (including return values) are simply numbered as they appear in the argument list. Finally injection cases have to be uniquely identifiable for each parameter and are defined for the data type and error model used.

To support extensibility and to have a flexible injection framework we have opted to use SWIFI. No specialized hardware support is required and injections are performed using software only. Figure 4.2 shows an overview of the main software components of the system, showing both the target and Host Computer. Apart from the OS itself and its drivers, the system contains the following main modules:

- **Host Computer:** The main responsibility of the Host Computer is to receive and store log messages sent by the Experiment Manager.

  The Host Computer is additionally used to build and download new OS images to the target computers.

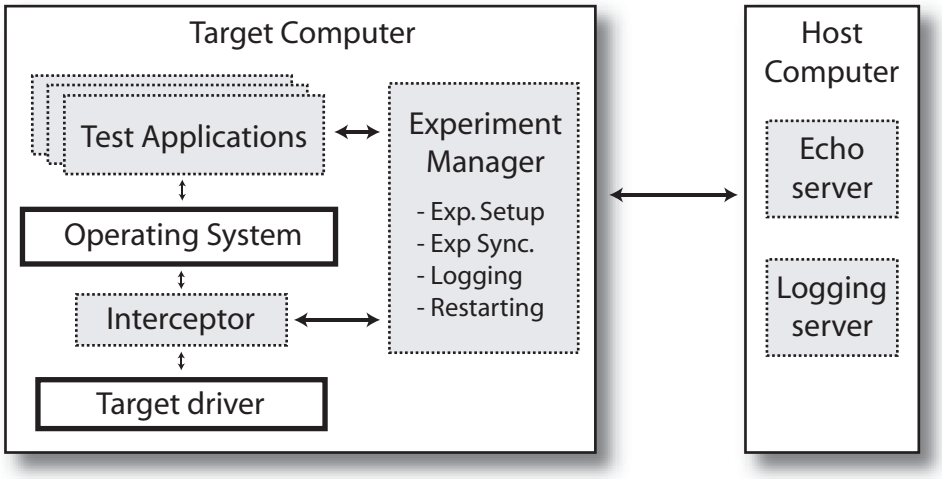

Figure 4.2: An overview of the experimental setup.

- **Experiment Manager:** Responsible for setup, control and logging of experiments. It communicates with the Host Computer, which stores log messages.
- **Interceptor:** The Interceptor is a module used to intercept communication between the OS and the targeted driver. Two types of Interceptors are used, one for tracking calls and one for injecting errors.
- **Test Applications:** The workload consists of a set of test applications exercising the system and the drivers in a multitude of ways.

The Injector and the Experiment Manager interact to coordinate each injection run and to send log messages to the Host Computer. Similarly, the test applications report their progress to the Experiment Manager, which forwards log messages. Information between modules is exchanged using message queues, a message passing primitive native to Windows CE .Net.

The components of the target computer are built into a new OS image as described in the previous section. The OS image is downloaded to the onboard flash memory and loaded into RAM each time the OS (cold) boots. A possible risk when conducting fault injection is the presence of dormant faults, i.e., faults from previous injections that are left dormant in the system. This can lead to unpredictable and non-reproducible results, as the outcome of an injection may be affected by such dormant faults. To minimize this risk the system is (cold) restarted between each injection, resulting in a

fresh copy of the OS image being loaded for each injection run. Logs are sent and stored on a different machine (the Host Computer) and a minimal set of configuration information is stored in flash memory. This process is conservative and common for fault injection experiments, e.g., [Chillarege and Bowen, 1989; Gu et al., 2003]. However, the restart before for each injection incurs a substantial run-time overhead when errors are masked or overwritten.

The process of producing an injection-ready OS image is illustrated in Figure 4.3. First the binary of the original driver is scanned to identify exported and imported services. Together with information in system header files and the online documentation the Interceptor module is constructed (**A** in Figure 4.3). For intercepting imported functions the binary of the original driver is modified to import the functions from the Interceptor module instead of the original services (**B** in Figure 4.3). For exported services the system is configured to use the Interceptor module instead of the original driver, by modifying the configuration of the loading process of drivers in the system Registry (**C** in Figure 4.3). Lastly the (modified) driver, Interceptor, configuration and other system components are merged into a single OS image to be downloaded onto the target computer (**D** in Figure 4.3).

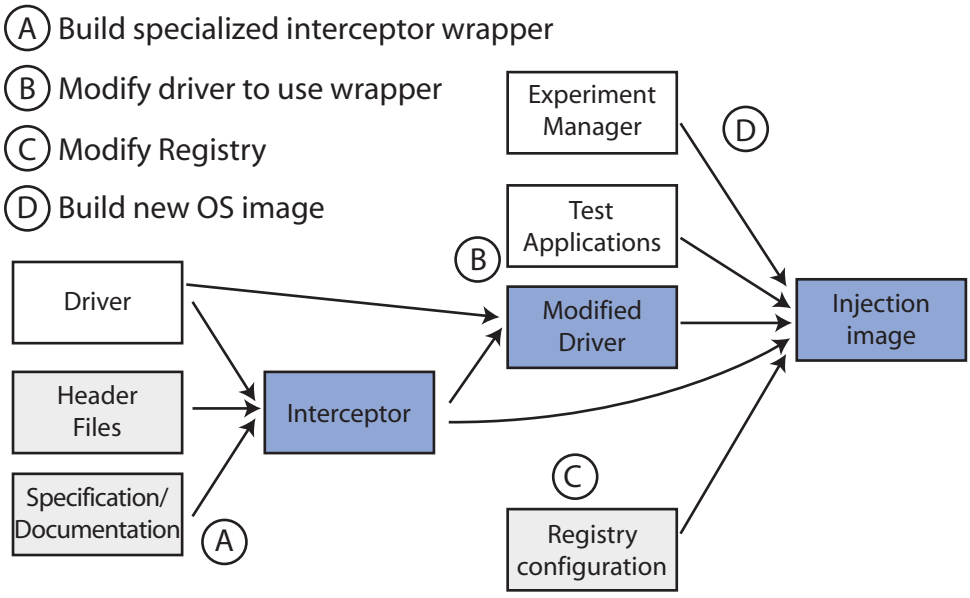

Figure 4.3: Building an OS image for injection.

4.5.1 Experiment Manager

The Experiment Manager runs as a separate process on each board and is responsible for setup, monitoring and logging. The parameters needed to configure the system are set up using the system Registry, which is part of the binary image of the OS built offline. These settings remain static throughout the experiment. Dynamic information concerning which injections have taken place and the configuration for the next injection is stored in a plain text configuration file in persistent (flash) storage on the device.

At boot time it starts by setting up a connection for sending log messages to the Host Computer, either using Ethernet or serial communication. From this point on log messages can be sent to the Host Computer. Depending on the targeted driver either serial or Ethernet communication is used to send log messages to the Host Computer.

Next it reads the configuration file to find out which service is to be targeted for the next injection (see Figure 4.4). When the injection data has been read they are marked as *pending* and the change is flushed to be made persistent. Once an experiment has finished, and before rebooting, the pending flag is changed to *finished*, indicating that the Experiment Manager could reboot the system in a clean way. This simple mechanism allows us to detect hangs and unclean reboots by the system. If, at boot time, a pending flag is found for the next injection a special log message is sent with this information.

The Experiment Manager then waits for the Interceptor to send a "Ready for injection" message, after which it sends the injection data (service, parameter and injection case). The Interceptor then handles the rest of the actual injection.

After an injection has been configured the test applications are started and monitored. Each test application updates the Experiment Manager, on any assertion violation. Their exit status is also monitored by the Experiment Manager and if they exit abnormally this is logged. If they have not exited within a given time period (about three times normal execution time) they are considered hung/crashed and this fact is logged.

Once the outcomes for each of the test applications is known the system is automatically cold restarted. The cold restart ensures that any data left in RAM is replaced and that a clean image of the OS is read back in from flash storage. For the case when an error causes the system to not respond to the Experiment Manager's attempts to reboot the system, a dedicated reboot process is used. This process is started automatically at boot time and simply tries to reboot the system after a specific timeout has triggered (currently four minutes). To define such a timeout is common practice for

fault injection [Arlat et al., 1990; Durães and Madeira, 2006]. If this also fails a manual hardware reset is required by the evaluator.

4.5.2 Host Computer

The Host Computer is used to manage and support the experiments. The Host Computer runs a set of experiment servers. Each server is responsible for communicating with one Target Computer. The server can be configured to respond both on network and serial communication. It is responsible for both echoing test application data sent as part of the workload on each Target Computer and for receiving log messages. It also keeps a timer for each log stream and if no message is detected within a given time the operator is alerted that the board may have hung, requiring a hardware reset.

Log messages are stored sequentially in a text file, one file per managed board. The operator can also add custom log messages to the log file, for instance that a board was manually restarted. The log files are processed off-line and the results are stored in a relational database. Currently we use SQL Server 2005 from Microsoft, but other databases could be used as we do no rely on specific functionalities of the underlying server. Header files are processed to match services to function names for easier handling of the log data. The use of a database significantly improves working with the data. Queries can be posed in a structured way and saved for later use using the SQL query language, in our case Transact-SQL [TSQ].

Experiments are classified into failure classes as presented in Section 5.2. This is done automatically on the data stored in the database for each experiment. Failure classes are defined as disjoint predicates on the experiment data, and are implemented as views in SQL. This is a very flexible approach as improvements to the failure classification scheme can be introduced very quickly by modifying the SQL definitions for each view. It is also very easy to run consistency checks on the data to check that the failure classes are indeed disjoint and complete (each experiment is in one, and only one class).

4.5.3 Interceptors

Communication between the targeted driver and the OS is tracked using *Interceptor* modules. There are two types of Interceptors, *trackers* and *injectors*. The Tracker is used only in Chapter 7 to track the calls made to a driver. The Injector is used to perform the actual injection.

Each service exported and imported by a driver is a function call to/from a dynamic link library (Dll). As previously described in Section 3.2.1 some error models require that the data types of the parameters used in function

calls to be tracked. Tracking the data type used serves two purposes, first and foremost it is used to select the specific error to inject, but it is also used to reduce the number of injections for error models based on bit strings (like the **BF** model described in Section 3.2.1) by restricting injection to the bits used.

The C-language does not provide any reflective mechanisms whereby the data type of a parameter can be discovered at run-time. This information is pertinent for data type-based injection. For most functions the information is available in the form of header files present on the system. In some rare cases online product documentation is used to resolve parameter definitions, in this case Microsoft's online developer documentation [MSDN].

Injection is done on one interface at a time. Exported functions are targeted separately from imported functions, thereby defining a single injection campaign. Functions from one imported Dll are targeted separately. The injection module wraps the driver and acts as a "trojan horse" to both the driver and the OS, by imitating the behavior of the other party. Similar strategies have been used in previous fault injection tools, for instance, [Segall et al., 1988].

For each injection campaign (driver/Dll) a separate Injector module is built, where an injection wrapper is built for each function in the service interface. The Injector interacts with the Experiment Manager and activates the targeted injection wrappers and can be configured to make multiple wrappers active for an injection run. However, for the experiments carried out only one was made active and the non-activated wrappers act as passthroughs, without touching the the parameter values used.

To make sure that the system itself does not modify or perturb the behavior of the system each experiment campaign starts with an error-free run, i.e., a run where all wrappers are in place but act as passthroughs. This run allows the evaluator to verify that communication with the Host Computer is setup properly and that no unexpected problems have arisen, before any actual experiments are carried out. During this error-free run the system is profiled to minimize the number of injections that need to be carried out. This process is further detailed in Section 4.6.

When targeting imported services the binary of the driver is modified. The binary format of drivers on Windows CE .Net follow the Portable Executable (PE) format [Mic, 2006], where Dlls being dynamically linked to the driver are specified together with the services used. By modifying the name of the library being linked, the Interceptor Dll can be linked instead of the original Dll. A dedicated application has been implemented for performing the modification of the binary image of a driver. It runs on the Host Computer and is used before building a new OS image. The Interceptor is

implemented to export all functions that the driver uses in the original Dll. The Interceptor then in turn loads the original Dll and can pass any calls along.

The Injector can work in three modes: a) for testing purposes it can act as a complete passthrough, b) it can wait for injection instructions from the Experiment Manager and then activate the appropriate injection wrapper, and c) it can build all injection wrappers with passthrough functionality and query their injection cases. The latter mode allows the creation of injection cases on the fly. This is important, as when a new error model is implemented/enhanced or when new functions are wrapped, the injection cases are automatically generated without human assistance, saving time and reducing the probability for mistakes. The two latter modes are illustrated in Figure 4.4

In Figure 4.4, when the system boots it check for the presence of a configuration file, specifying which errors are to be injected and which injections have already been performed, as previously described. If one exists it is read and the Interceptor builds the required injection wrapper. Next the Interceptor sends the "ready for injection" message to the Experiment Manager, which then starts the test applications (workload). The Interceptor is now ready to inject the error when the specified trigger fires. When injection is finished (intermittent and permanent error models may inject multiple errors) the system waits for the test applications to exit before updating the configuration file and then rebooting. Note that if the system is still prepared to inject errors when the test applications exit the system is rebooted anyway, else permanent errors, or errors not being triggered by a certain workload lead to livelock (the system waiting for the error to be triggered, which will not happen).

In parallel to the injection process a watchdog timeout process is started which reboots the system after a set timeout from boot time has elapsed. The purpose of the watchdog is to reboot the system in case the rest of the system fails due to an error and the normal reboot step is not reached. Currently the timeout is set to 200 seconds, more than twice the normal execution time of the system.

Building injection wrappers is currently a manual process, but since the information required is (mostly) available in parsable header files an automatic processing is possible, similar to the approach used in [Fetzer and Xiao, 2002b]. As the wrapper only needs to be implemented once for each function, and can thereafter be used for any error model, the approach still scales reasonably well. However, for a large scale deployment of the approach this step needs to be automated as much as possible.

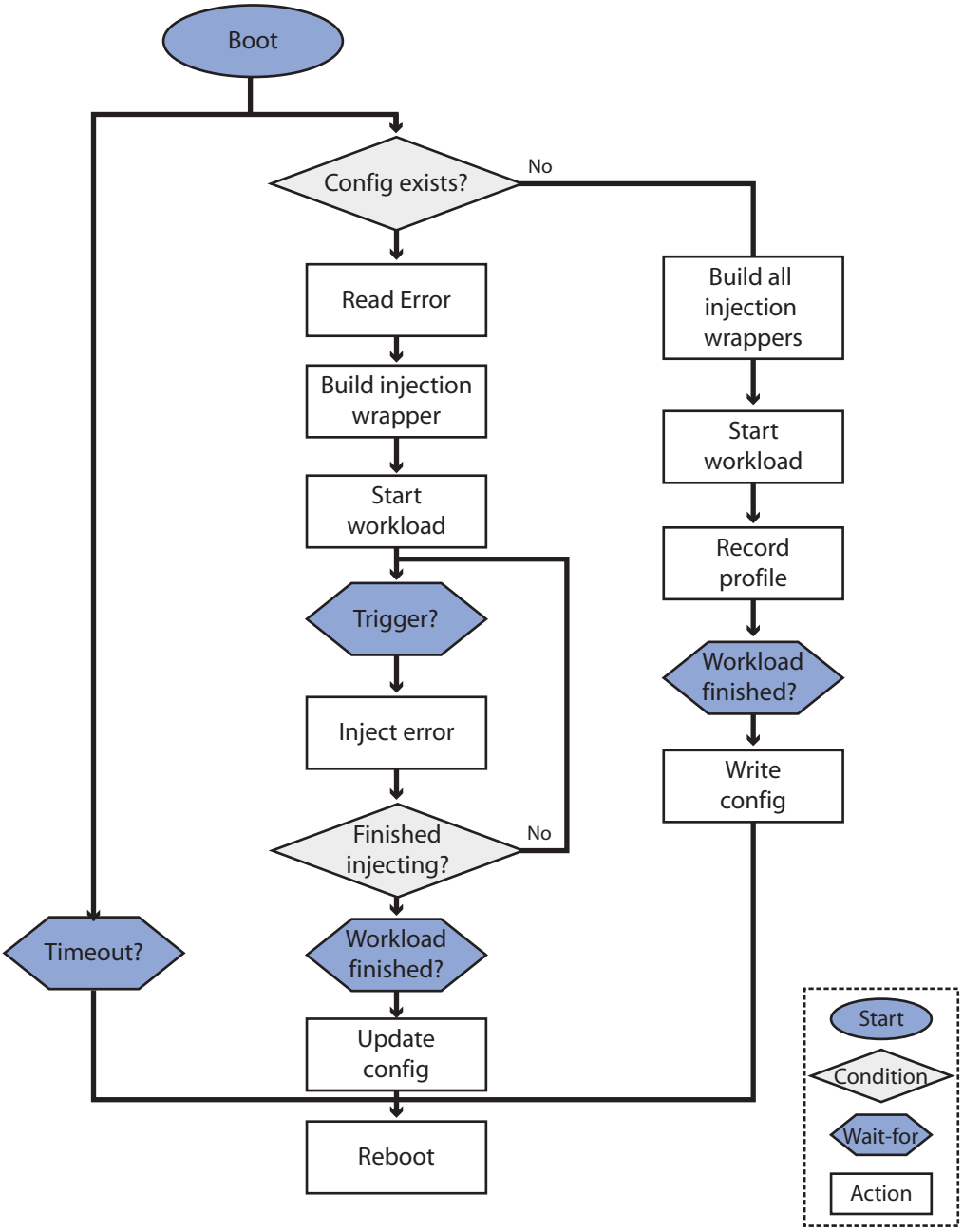

Figure 4.4: An overview of the injection process.

Specifying Error Model

Each Injector builds an in-memory data structure (a C++ object) containing information regarding the data types used, and pointers to the values passed

for the targeted service. For each service one such object is built. The data types used for a specific service are hard coded into the code. This is no real limitation as the information is static (function definitions) and only need to be defined once for each function. The error model is specified using a plugin model, where a model is specified using the error type, timing (duration) and trigger, analogously to the description in Section 3.2.

For each parameter targeted for injection the data type needs to be recorded and the error selected accordingly. The data tracking mechanism is implemented in C++, but the interface between device drivers and the OS is defined using C. We make a distinction between three major classes of parameter types, namely:

- **Basic types** - The basic types include the built-in types provided by the programming language, like integers and booleans, as well as specialized types where it makes sense to use specific injection cases, for instance `HKEY` representing a handle to a Registry key.
- **Structures** - This is the `struct` type used in C. It contains a set of members which themselves could be of any of the three classes.
- **Pointers** - These are references to other values, which could basic types, structures or other pointers.

The tracking mechanism is built around the concept of a *Parameter*. A *Parameter* is of any of the three classes: basic type, structure or pointer. Figure 4.5 shows the relation between the three classes of types found. Structures and pointers refer to other parameter values, which in turn can be any of the three classes.

The three classes are implemented as C++ classes inheriting from a Parameter base class. For each data type tracked a new specialized class is implemented. A Parameter provides methods for specifying error models, querying for injection cases and to inject errors. Structures and pointers furthermore contains methods for adding members and references.

Figure 4.5 illustrates our implementation of the data tracking mechanism. For each targeted parameter an object is defined, whose class inherits from the Parameter class. Using inheritance, new data types can easily be added, provided they inherit from the Parameter class and implement the required methods for injecting errors etc.

As previously explained in Section 4.5.3 an injection wrapper is built for each targeted function. The wrapper builds a model of the interface using the classes explained above together with information regarding the error model. Using this information, the Experiment Manager can query the wrapper for

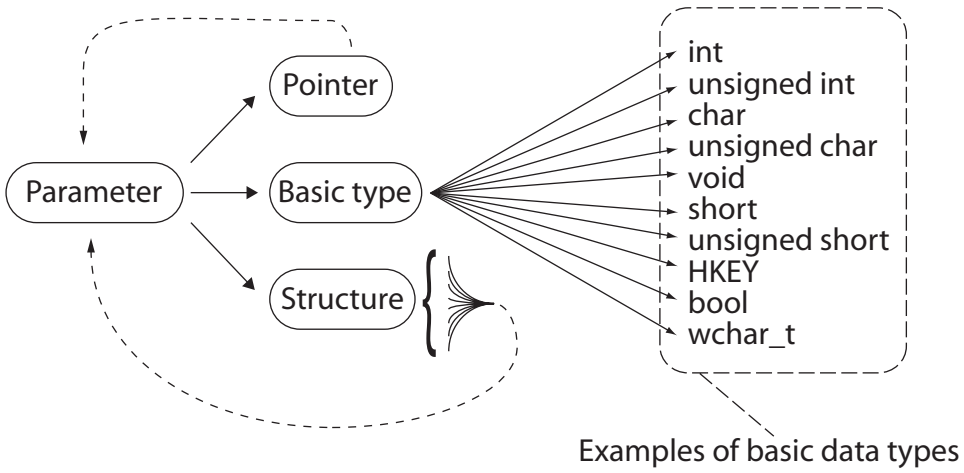

Figure 4.5: Data type tracking mechanism.

the injection cases possible for the given error model. This enables automatic configuration and generation of injection cases, illustrated in the right branch of Figure 4.4.

The plugin model makes the injection framework considerably more flexible, compared to hard coded injections on a service/driver basis. Once the injection wrapper is defined several injection campaigns can easily be performed by specifying different error types, durations and trigger models. The error type, duration and trigger are specified as keys in the Registry, which are extracted by the Experiment Manager at boot time. The injection wrapper is then instructed to build the corresponding injection object online. This facilitates a very flexible architecture, that automatically extract the injection cases to be performed. The number of injections required is extracted from the error type object. The first time the system is booted all injection objects are built and the injections to be performed are stored in a file.

Three error type plugins have been implemented (**BF**, **DT** and **FZ**), but additional models can easily be implemented, including timing errors (delays). Three error durations are implemented, transient (occur only once), intermittent (occur x times) and permanent. As previously described only the transient model has been evaluated. The triggering mechanisms implemented include first-occurrence, call block-based and timeout-based models. One can also specify more advanced triggering mechanisms, where errors are triggered after x calls to a service or only after calls to a certain (other) service.

4.5.4 Test Applications

The workload for a robustness evaluation is typically the set of user applications running on the system, together with their inputs. The purpose of the workload in this context is two-fold: a) to drive the use of the system and make sure that all relevant parts of the OS are used, including devices attached to the system; and b) to detect any robustness violations in the OS services used.

When robustness evaluation is carried out on a finished product (or prototype) the workload should as closely as possible mimic the use of the product in its operational setting. However, for generic components, like an OS, information on operational settings might be unknown. For such cases generic workloads are typically used. Examples include standard performance benchmark applications [Barton et al., 1990; Kanawati et al., 1992; Carreira et al., 1998]. The same problem arises when testing components, which may be reused in the future. The future uses may not match the operational profile available at development time and thus testers are forced to anticipate "typical" use of the component [Weyuker, 1998].

Another approach is to test each service individually by defining a test harness simulating operational conditions. The harness needs to setup any specific context (such as held resources, open files etc) needed for the service to be called in a realistic setting. This approach was used for instance in Ballista [Koopman and DeVale, 1999] and HEALERS [Fetzer and Xiao, 2002b]. We have opted to use a realistic workload instead, thus avoiding the problem of defining appropriate context scenarios.

The workload used in this thesis consists of a set of test applications. The purpose of the test applications is to exercise a wide variety of OS services and specifically the device drivers evaluated. Each test application is enhanced with additional assertions that verify that each call to an OS service returns the expected result. The expected result is derived from the documentation of the OS and a golden run of the application. The test applications are kept as simple as possible to make them deterministic. This allows assertions to be manually inserted into the code of each test application to verify the correctness. Three types of test applications are used, using the OS in different ways:

[Memory Management]: The application allocates memory and access it. The memory is then released.

[File System Operations]: Normal text files are created and opened. Some text is written to the file and read back. File attributes are set and checked. The file is finally deleted.

[**Driver Specific**] The driver specific test application uses the driver by issuing driver specific operations.

For each driver tested a specific test application is built to test the driver's functionality. For a network card driver packets are sent and received using a connection to the Host Computer. Similarly, the serial driver reads and writes on the serial port connected to the Host Computer. Specific consistency checking assertions are added to check for any errors in the received echo strings. Similarly, the echo server on the Host Computer also checks for incomplete or otherwise erroneous messages. The CompactFlash driver is tested in a similar fashion to the file system tests above.

To get a consistent system for all injections, and to establish a common ground for comparing drivers, all driver specific test applications are executed for each test, even when the specific driver is not targeted.

4.6 Pre-Profiling

A key concern with any fault injection is to keep the level of activation for the injected faults as high as possible, i.e., as many as possible of the faults should be activated and become errors. Since typically the goal of fault injection is to exercise the system's fault/error handling mechanisms and observe how it behaves in the presence of faults, the activation rate is a measure of how effectively these mechanisms are tested. Note that the goal of injecting faults can also be to assess the activation rate itself, i.e., how easily faults are activated and become errors. However, this is not the case in this thesis. In general, one wants to achieve an *acceleration* of the fault $\rightarrow$ error $\rightarrow$ failure process [Chillarege and Bowen, 1989].

Robustness evaluations of large systems, such as OS's do not target evaluation of specific fault tolerance mechanisms. Therefore, one cannot generally know if a non-activated injected fault is still dormant, has been overwritten, or detected and corrected. To cope with this an experimental timeout is set, after which the fault is declared to have disappeared, either by being overwritten or handled by the system. The timeout needs to be long enough to justify this assumption, but short enough to make experimentation feasible. A high activation level naturally helps speed up the experimental process, as fewer experiments need to run until the timeout elapses. For the experiments presented in subsequent chapters a timeout of four minutes was used. This was set to be more than 100% longer than the execution time of the test applications in fault-free scenarios. Initial experiments with significantly longer timeouts did not reveal any dormant faults surfacing.

By injecting errors at a high-level interface which is readily accessible one can achieve 100% activation ratio of injected errors, by employing a pre-profiling stage before the experiments starts. This stage profiles the component in question and records each invocation made in the interface. This information is then used to filter out any injections that would never take place (because the function is not used). This technique assumes a deterministic invocation pattern in the sense that the same set of services are invoked for each run of the system. This property is also required to get repeatable results, an important aspect of system evaluation. The use of test applications that give rise to a deterministic workload makes this assumption justified and indeed for the experiments carried out for this thesis, no such deviations were observed. However, it is important to re-profile the system as any changes are made, especially regarding the test applications, as they might give rise to new invocation patterns for the drivers.

For the injection experiments carried out a pre-profiling stage is run for each experiment campaign. The first time the system boots the test applications are executed as they are when errors are injected. In this case, an invocation profile of the targeted driver is *automatically* collected instead of an error being injected (right branch of Figure 4.4). Based on the services that are marked as used the injection cases generated can be filtered, such that errors are only injected for services that are *actually* used. By automatically performing the profiling for each new configuration any changes made to the system configuration are always considered.

Additionally to recording which services are invoked, also the number of invocations is stored. This is used to further eliminate non-activated injections when investigating the time of injection in Chapter 7.

In the DTS tool a similar approach to ours is used [Tsai and Singh, 2000]. Library calls made by an application are targeted. Only calls actually performed are targeted, reducing the potentially large set of functions considerably. We use the same strategy, reducing the number of targeted functions by 36.7% on average. Table 4.1 shows the number of services specified and the number of services actually used. The table shows that many services are not used, indicating that more complex workloads may potentially incur more services to be targeted and thus for more injections to be performed.

In [Süßkraut and Fetzer, 2007] static analysis of library code is performed to reduce the number of injections. Using **DT** injections similar to ours, the injection cases to use are selected depending on how each parameter might be used. This information is found by statically analyzing the code to find out which (other) library functions are called for the targeted function and based on this restrict the number of injections. This does not limit the number of services targeted, but the number of injection cases required for each function

Table 4.1: The number of services specified in the OS-Driver interface and the number of services used for the specified workload.

Driver	Specified	Used	[%]
cerfio_serial	60	46	76.7
91C111	54	26	48.1
atadisk	47	30	63.8

parameter.

4.7 Summary of Research Contributions

This chapter presents the injection framework used for performing the fault injection experiments performed for this thesis. The framework is implemented for Windows CE .Net specifically focusing on extensibility and automation. Using a plugin model three error models have been implemented and the framework is easily extended to include more error models. Several aspects of the process have been automated, to minimize the risk of user mistakes and to expedite experimentation. The framework provides the following benefits for the evaluation of OS robustness:

- A *flexible* and *extensible* plugin model for easy adoption of new error models.
- A *low-intrusion, black-box* injection methodology, not requiring access to source code.
- A high degree of automation, not requiring manual actions to be taken when changes to the used error model or workload are performed.
- An efficient selection of injection cases, eliminating injections that would not have been activated for the used workload.

Chapter 5

Error Propagation in Operating Systems - Where to Inject

How to measure error propagation in OS's? What are quantifiable measures of error propagation?

That software components contain faults that lead to undesirable behaviors in the form of errors and failures is a fact that will not disappear in the foreseeable future. In a system design it is therefore important to quantify how such errors may affect the system as a whole. An important aspect of this is how errors spread throughout the system, i.e., how they propagate. Another aspect is the effect they have, i.e., the failure modes they give rise to. Both are important as they allow a designer to a) quantify potential dependability bottlenecks in the system, b) assist in the designing higher quality components and c) to guide addition of enhancements in the system in an effective manner.

This chapter introduces a series of measures that can be used to quantify error propagation, **RQ2** - quantifiable measures of robustness. They are based on the previously described system model, and show that by introducing errors in the interface between device drivers and the OS can be used to study how the OS treats faulty drivers. As such it also considers **RQ3** - the question of where to inject errors.

5.1 Introduction

This chapter defines a framework for the evaluation of error propagation with respect to robustness in OS's. As detailed in Chapter 1 an important aspect when increasing the robustness of a system is identifying potential sources and sinks for error propagation (**RQ1**). The goal of this chapter is to define measures that help identifying such services. To achieve this we use failure mode analysis and four separate measures: *Service Error Permeability*, *Service Error Exposure*, *Service Error Diffusion* and *Driver Error Diffusion*. After their definition a discussion on their use and its implications is presented.

As previously discussed, the use of the OS-Driver interface for measuring error propagation and effect is suitable for many reasons, such as portability across drivers, low intrusion (as no source code changes are required) and it allows injection of multiple driver faults. The measure presented in this chapter are defined for errors appearing at this level, and therefore further substantiate the chose of error location, i.e., answers the question of *where* to inject errors.

This chapter presents the analytical foundation upon which the following chapters build. Further discussion on using the framework presented here and its implication from a quantitative point of view will be discussed in subsequent chapters.

5.2 Failure Mode Analysis

Robustness evaluation is similar to Failure Mode and Effect Analysis (FMEA), a well known technique in reliability engineering [Leveson, 1995]. In FMEA the failure modes of individual components are postulated and their effect on other components and the whole system derived. Robustness evaluation differs, as it is experimental in nature, treating a system not as a static entity, but a dynamic system in context.

Failure modes for use in robustness evaluation are typically postulated beforehand (they may be iteratively refined of course) as a set of failure modes, or classes. For safety critical systems, FMEA may also be performed to investigate which failure modes the system or component shows in operation. Even though the techniques developed here may help towards this goal as well, it is not the prime focus of this thesis.

The failure severity scale used in this thesis is similar to several previous severity scales. Siewiorek et. al. used a five grade scale for their robustness benchmark [Siewiorek et al., 1993]. Many fault injection tools have used

similar scales as well, like MAFALDA [Arlat et al., 2002; Rodriguez et al., 2002], NFTAPE [Gu et al., 2003] and others, for instance [Chillarege and Bowen, 1989; Barton et al., 1990; Marsden and Fabre, 2001; Durães and Madeira, 2006]. As a representative example of failure modes defined from an application perspective, the CRASH severity scale presented in [Koopman et al., 1997] is shown in Table 5.1. The API of the OS is tested by creating a specific task that calls the targeted function and the outcome is classified according to the CRASH scale.

Table 5.1: The CRASH severity scale from [Koopman et al., 1997].

Failure Mode	**Description**
Catastrophic	System crash
Restart	The task is hung and requires a restart
Abort	The task terminates abnormally
Silent	No error report is generated by the OS, even though the operation tested cannot be performed and should generate an error
Hindering	Incorrect error returned

The failure classes used in this thesis are listed in Table 5.2. We use the term failure class as each failure class may correspond to multiple failure modes depending on the desired level of granularity. The chosen classes represent generic classes that apply to general purpose systems. The failure classes are defined to be disjoint, such that the outcome of an experiment can be unambiguously determined to be a member of a specific class. Whenever the outcome fits the description of one or more classes it is assigned the more severe one. For instance, an error that first causes an application to crash and then the rest of the system would only be considered in the latter class.

5.3 Error Propagation

Error propagation in software happens when a fault is activated (becomes an error) and then subsequently used in a computation, leading to a new error at a different location [Lee and Iyer, 1993; Voas et al., 1996]. As an example, consider a faulty line of code where the wrong value is assigned an integer variable. This value is read and used in a condition statement and the wrong decision is taken, leading to a set of statements being executed in error. The error has now propagated from the assignment to another part of the component. The error might continue to propagate and may propagate to

Table 5.2: The failure classes used.

Failure Class	Description
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	The error propagated, but still satisfied the OS service specification as defined in the documentation. Examples of **Class 1** outcomes are when an error code 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	The 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.
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.

other components by function calls, message passing, shared memory areas etc. Eventually, the error might propagate to the outputs of the system, there causing a failure.

Knowing which errors propagate and where is important because it enables counter measures to be taken. A general design principle in the design of dependable systems is the concept of *error containment*, i.e., that a component masks any errors, not exposing interacting components to propagating errors [Pradhan, 1996]. For software this is difficult to realize in practice for every type of errors. Instead, the failure modes and the propagation paths need to be found, such their impact can be characterized.

At least three main uses for knowledge regarding error propagation can be envisioned:

- *Identifying robustness bottlenecks in the system.* Some components may be more likely to spread errors or more likely to be the sink for errors

propagating in the system. These components should be the focus of other verification and validation efforts.

- *Expose flaws and and their impact on system dependability.* Error propagation may reveal real flaws in the software and may thereby assist in the designing higher quality components. The impact of such flaws can be characterized using for instance failure mode analysis, which helps a designer focus attention to the components which cause severe damage.

- *Locating error detection and recovery mechanisms.* The error propagation paths identify locations where specific error detection and recover may be added. By placing them along such paths their effectiveness is increased.

The framework presented in this chapter aims at all of these three goals. A discussion is provided on how the measures defined can help achieve this. Chapter 6 presents an implementation of these measures on a real OS and discusses their suitability.

5.3.1 Failure Class Distribution

The simplest way to compare a system's ability to withstand errors in drivers is to compare the number of severe failures the system incurs as a result of injected errors. Since the number of failures depends on the chosen error model, i.e., the number of injections performed, one can use the ratios of failures into different failure classes, the failure class distribution.

The failure class distribution highlights key differences between drivers and gives a fast overview of different driver's and/or error model's ability to provoke failures in the system. However, when more detailed results and guidance is needed more refined measures should be used, as the one presented in the next section.

5.3.2 Error Propagation Measures

In the context of this thesis we are interested in how errors in device drivers propagate throughout the system. To do this, we need to clearly specify the observation points where error propagation is measured. Errors are injected in the interface between the driver and the OS. Observations are made from a user perspective by observing the behavior of user-space applications. This gives us the ability to characterize the relation between drivers' ability to spread errors and an application's use of OS services.

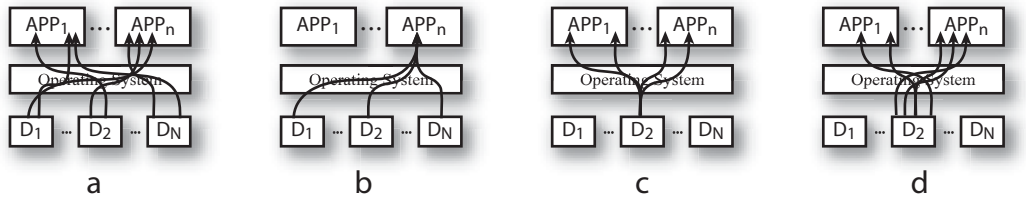

Figure 5.1: The four propagation measures introduced: a) Service Error Permeability, b) Service Error Exposure, c) Service Error Diffusion, and d) Driver Error Diffusion.

With the intent of finding robustness bottlenecks in the system three main goals are identified: a) to identify services in the OS-Application interface that are the likely sinks for propagating errors, b) to identify services in the OS-Driver interface that are more likely to spread errors, and c) to identify drivers that are more likely to spread errors in the system, given that errors are present. To facilitate such an identification, a basic propagation measure is defined, the *Service Error Permeability*, capturing the likelihood that an error in the OS-Driver interface will spread to an OS-Application service. The objectives for our measures are illustrated in Figure 5.1 and are summarized as follows:

(a) Measure for degree of error porosity of an OS service: *Service Error Permeability*,

(b) Measure for error exposure of an OS service: *Service Error Exposure*,

(c) Measure of a driver service' proneness to spread errors, *Service Error Diffusion*, and

(d) Measure of a driver's ability to spread errors in the system through the OS-Driver interface: *Driver Error Diffusion*.

Service Error Permeability

The Service Error Permeability is the probability that an error propagates from a specific service in the OS-Driver interface to a specific service in the OS-Application interface. It is conditioned on the presence of an error in the first place. That it is a conditional probability is significant, since else we would have to know the probability of error occurrence, which is inherently difficult and system specific. With the conditional probability we still get an assessment of the system's ability to contain error propagation and when

a error occurrence probability is know it can be combined with the Service Error Permeability.

Two classes of services are identified in the OS-Driver interface, as shown previously in Figure 3.2 (page 36). Each driver D_x exports a set of services $ds_{x.1} \cdots ds_{x.N}$. These are the services that the OS calls to instruct the driver to perform a certain operation. To implement its functionality a driver also use a set of OS services $os_{x.1} \cdots os_{x.M}$.

For a given driver or project, only one of the classes may be of interest, for instance, for drivers that do not make extensive use of the export interface. The Service Error Permeability is therefore defined for each class explicitly. $PDS^i_{x.y}$ is defined for the exported services and $POS^i_{x.y}$ for the imported.

The Service Error Permeabilities is defined for a driver D_x, an OS-Application service s_i and an OS-Driver service (either $ds_{x.y}$ or $os_{x.y}$):

$$PDS^i_{x.y} = Pr\left(\text{error in } s_i | \text{ error in } ds_{x.y}\right) \tag{5.1}$$

$$POS^i_{x.y} = Pr\left(\text{error in } s_i | \text{ error in use of } os_{x.y}\right) \tag{5.2}$$

Typically, propagation is evaluated to a service in a *specific* application, i.e., $s_i \in \mathcal{A}_x$ is defined for a specific application APP_x, and this is the way it is interpreted here.

Service Error Permeability gives an indication of the permeability of the particular OS service, i.e., how *easily* the OS lets errors in a specific service in the OS-Driver interface propagate to a service used by an application. A higher permeability implies that precautions need to be taken for the services involved. Such precautions could entail either ensuring that the services are properly used (fault prevention and removal methods), including handling exceptional situations, or addition of error handling code. Note that Equation 5.2 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.

Note that the Service Error Permeability is defined with respect to subsets of the services at the OS-Application interface, $\mathcal{S}$, and OS-Driver interface, $\mathcal{O}$. For service pairs not members of this subset, no assertion can be made about their permeability. It is therefore desirable to make these subsets representative of the set of services used when the system is operational.

Service Error Exposure

To the find OS services that are more exposed to errors propagating through the system, a set of relevant drivers needs to be considered. The propagation

from this set of drivers can be combined into a composite measure, namely the *Service Error Exposure*[1] (E^i). Service Error Exposure considers each driver's contribution to the propagation of errors to a specific OS-Application service s_i (see Figure 3.1). Thus it is an estimation on how *exposed* this service is to propagating errors from these drivers. Both PDS and POS contribute to the Service Error Exposure, and consequently both are part of Equation 5.3.

We use the measure Service Error Permeability, to compose the *Service Error Exposure* for an OS service s_i, namely E^i:

$$E^i = \sum_{\forall x} \sum_{\forall j} POS^i_{x.j} + \sum_{\forall x} \sum_{\forall j} PDS^i_{x.j} \tag{5.3}$$

Service Error Exposure considers all drivers influence on one OS service. Thus its use is mostly to compare OS services and rank them based on their exposure to propagating errors. It can therefore be used to guide verification efforts of applications or placement of error handling mechanisms on the application level. Note that this expression implies aggregating all imported and exported Service Error Permeabilities (5.1 & 5.2 above) and all considered drivers. When comparing services on a driver per driver basis the *driver specific* Service Error Exposure can be applied:

$$E^i_x = \sum_{\forall j} POS^i_{x.j} + \sum_{\forall j} PDS^i_{x.j} \tag{5.4}$$

The driver specific service exposure allows study of driver attributed differences in exposure of propagating errors. It also makes assessment of exposure independent of the selected set of drivers.

Service Error Diffusion

As Service Error Exposure considers a specific service at the OS-Application level, Service Error Diffusion ($SE_{x.y}$) focuses on specific services on the OS-Driver level. This allows us to pin-point services which are more likely to spread errors through the system. $SE_{x.y}$ for an imported services is defined for a driver D_x and a specific $os_{x.y}$:

$$SE_{x.y} = \sum_{\forall i} POS^i_{x.y} \tag{5.5}$$

Service Error Diffusion for exported services can be calculated analogously to 5.5 using $PDS^i_{x.y}$.

[1] We will use the term *Service Error Exposure* and *Service Exposure* interchangeably.

Service Error Diffusion can be used to rank driver services on their ability to spread errors in the system. Values can be compared either globally (across all drivers) or locally, for a specific driver D_x.

Driver Error Diffusion

Driver Error Diffusion is used to rank drivers on their ability to spread errors in the system. To do this, the Service Error Permeability values for one driver are aggregated. A higher value means that the driver may more easily spread errors in the system. For a driver D_x and a set of services, the Driver Error Diffusion, D^x is defined as:

$$D^x = \sum_{\forall i} \sum_{\forall j} POS^i_{x.j} + \sum_{\forall i} \sum_{\forall j} PDS^i_{x.j} \tag{5.6}$$

Analogous to Service Error Exposure, a higher Driver Error Diffusion value is an indication of where efforts need to be spent on verification or where error detection/recovery mechanisms should be placed in the system. Since Driver Error Diffusion focuses on the driver level, locations are identified on this level as well.

Once a ranking across drivers exists, the driver(s) with the highest Driver Error Diffusion should be the first targets. Details on specific error paths can now be used (i.e., Service Error Permeability) to guide the composition and placement of detection and recovery mechanisms.

5.3.3 Use of Measures

The previous section presented six different measures. A natural question an evaluator might have is therefore which of these to use for a specific project.

Three key uses for error propagation analysis were identified in Section 5.3. When the goal is to identify robustness bottlenecks, Service Error Exposure and Service Error Diffusion can be used to guide the search for specific services, and more information can be then be gained by studying individual Service Error Permeability values for each considered services. Driver with potential for spreading errors can be identified using Driver Error Diffusion.

Information used for debugging (exposing flaws) can be gained by looking at the specific injection cases identified by Service Error Exposure and Service Error Diffusion. These also help in locating error detection and recovery mechanisms in the system by identifying prominent propagation paths.

Ultimately it is the level of detailed required that guides the use of the proposed measures. As all presented measures are based on the same raw data, no significant overhead is attached to the calculation of each measure.

For the experimental setup used all measures are predefined as SQL scripts, which are loaded and executed on the data stored in the database. All scripts execute for at most a few seconds on the used workstation. The use of a database means that the only operation required when new experiments have been conducted is to load them into the database, a task for which a dedicated application has been implemented greatly simplifying the task.

5.4 Discussion

This section provides a general discussion on some of the concepts presented in this chapter. A more detailed discussion regarding implementation details and interpretation of the measures is found in Chapter 6.

Failure Classes

When investigating error propagation in a system not all errors can be treated equally. Some errors lead to severe failures and some to mere annoyances. What constitutes the "severity" of a failure is system dependent and a subjective property. Different users may consider different failures as worst. For instance, Durães et. al. [Durães and Madeira, 2003] define a set of *generic failure modes*, and depending on the user of the evaluation define severity scales as subsets of the generic failure modes accordingly. From a feedback point of view the worst failure is the loss of data without any warning, whereas from an availability point of view a completely unresponsive system is the worst.

For general purpose systems, such differentiating views become problematic. As an example, many users are frustrated when their desktop PC crashes due to failure for some driver they did not know existed on their system. However, that the system "crashes" may be an explicit decision by the OS to avoid inconsistencies of data or even loss of data. When no knowledge of the cause or remedy for an error exist, the only sane thing to do might be to take the system down and hope that the error has disappeared when the system is restarted. Had the system been written for a dedicated purpose, correct diagnosis and recovery might have been possible, and the crash avoided. A crash can also be desirable if the OS is to implement "fail silent" behavior, i.e., when the system fails it does so by stopping to respond and without any other side effects [Powell et al., 1988].

Since this thesis is concerned with robustness of OS's we have opted to use a generic severity scale for the failure modes defined. It is important to note that even though we do consider the scale as containing progressively more

severe failures, this only reflects a generic severity ranking. Context input is needed to refine the severity scale used for a specific system, in order to define useful and comparable failure classes. Additionally, more fine-grained failure modes can be defined when knowledge regarding a specific system is known. For instance, applications running on the system might be of different criticality and failure modes reflecting this may be desirable.

Interpretation & Evaluation

The usefulness of analysis using failure classes is that resources can be guided to the more severe failure classes, thus using them more efficiently. This applies to both fault prevention and removal efforts, such as improvement of the engineering process or different kinds of verification efforts, as well as for fault tolerance approaches where error detection and recovery is enhanced by addition of software mechanisms.

A typical process is to start with the most severe failure class, and then progressively approach the less severe classes as time and money permits. This helps to ensure that efforts are spent where the pay-off is the greatest and may also be used as a stop criteria for robustness enhancement.

Another important practical aspect is the impact different failure classes have. **Class 3** failures force the whole system to halt, i.e., one could argue that the error propagated to all services on the OS-Application level. In such cases, the services on the OS-Application layer do not impact the relative comparison of drivers, i.e., the Driver Error Diffusion. When comparing drivers using Driver Error Diffusion for **Class 3** failures one can therefore simplify Equations 5.1 and 5.2 to only consider the probability that an error propagates at all (since we know it propagates to all services). The consideration of each OS-Application level service would only give a linear scaling of the Driver Error Diffusion, not affecting the relative order across drivers. Chapter 6 shows how such simplifications can be made for a real system.

Imports vs. Exports

In this chapter we make a distinction between the imported and the exported services of a driver. This distinction may not be useful in all contexts, and the services can then simply be "bundled" together to form one set of services. This would simplify Equations 5.3 - 5.6 by using only one term $PS^i_{x.y}$ defined as follows:

$$PS^i_{x.y} = Pr\left(\text{error in } s_i | \text{ error in use of } s_{x.y}\right) \tag{5.7}$$

where $s_{x.y}$ is a service from the combined set ($\mathcal{O}$) of all imported and exported services for driver D_x.

Error Distribution & Operational Profile

An important aspect when interpreting the values for the measures presented is the lack of explicit dependence on error input distribution. In any practical setting such a distribution is very important. From a robustness point of view, the error distribution may be of less importance since the goal is not to estimate the reliability of the system. Equations 5.1 and 5.2 are conditioned on the presence of an error and can be combined with an error distribution when available.

Another important aspect is the implicit dependence on the operational profile of the system. The operational profile includes the usage profile of the applications running on the system which implicitly gives rise to a driver usage profile. Depending on how applications are used, different services provided by the OS are used and the usage profile of drivers differ. For a certain profile some services may not be used at all, whereas in others they are frequently used. This influences the values of the measures, since only services actually used are included.

The operational profile of a system may not be known at the time of the evaluation, or may change with time. This means that the robustness profile of the system may change as well. It is therefore important to try to use a profile closely matching the expected one when doing experimental estimations of the measures.

As a last point it is important to note that we do not try to test 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.

5.5 Related Work

Error propagation analysis and failure mode analysis are two intertwined concepts. Since both are well established techniques there is a plethora of literature making use of them. This section reviews some of the more important research contributions within both areas.

Error propagation studies how the effects of errors percolate through the system and affect other components than the source component of the fault [Lee and Iyer, 1993]. Voas et. al. presents EPA (Extended Propagation Analysis) [Voas et al., 1996, 1997; Voas, 1997a] which identifies code locations

which might violate the safety requirements of the system if faults occur in these locations. The authors introduce the term *failure tolerance* to mean that the system is tolerant to failures of 3^{rd} party software. In [Voas et al., 1997] the authors further speculate that a similar technique would be most useful for an OS setting, since system software consists of a multitude of interacting components.

For dependable system designs, error propagation is a phenomena that is to be avoided, since it allows the failure of one subsystem to affect the behavior of other subsystems and lead to a failure of the entire system. However, error propagation is a useful property in software testing, as it helps to reveal state corruption due to faults by propagating such faults to the interfaces of the system [Voas and Miller, 1994a, 1995; Voas et al., 1997]. Therefore, for components with high error propagation probability, testing is more likely to reveal faults in the code, if they are present. From this point of view, robustness evaluation identifies hot-spots in the system where errors can lead to severe consequences. Once these hot-spots have been identified and treated (either by ensuring that faults are not present or by adding error detection/recovery capabilities) the likelihood of the system propagating errors is lowered. In [Voas and Miller, 1994b] the propagation information is used to place such assertions at the most effective locations in the code, such that testing becomes effective.

Michael and Jones show that data state errors in software propagate uniformly, i.e., either all data state errors for a specific location propagate to the outputs of a program, or none of them do [Michael and Jones, 1997]. From a theoretical viewpoint this is surprising, but to a practitioner it might not be. When only a small subset of the value domain for a variable can be considered "correct", then most changes to this variable will be erroneous and may trigger propagation of faults. This is especially true for values which are assumed to be correct by the developer, and are therefore not checked for validity in the code.

Hiller developed an extensive propagation profiling framework for embedded control systems [Hiller et al., 2004; Hiller, 2002]. Based on a component model error propagation metrics similar to the ones developed in this thesis are presented. Whereas the focus there was on data level errors in control software, we focus on a single (although complex) component of computer-based systems, the OS.

5.6 Summary of Research Contributions

This chapter introduced the concept of error propagation in OS's, using failure mode analysis. A series of OS level propagation measures were defined that help an evaluator find system bottlenecks and to guide further efforts to components that are more likely to spread errors or more likely to be the sink for propagating errors. Related research projects in the area of failure mode analysis and error propagation, especially related to OS's were reviewed. Table 5.3 summarizes the measures introduced in this chapter.

The following research contributions are presented in this chapter:

- Error propagation in the context of OS's is defined in a generic and system independent manner. The fundamental propagation measure Service Error Permeabilityis used to measure the propagation across services in the OS-Driver interface with services in the OS-Application layer.
- The Service Error Exposure measure is introduced to measure the influence propagating errors have on specific OS services. It can be used to compare services in a relative manner.
- Service Error Diffusion can be used to rank *services* in the OS-Driver interface on their ability to spread errors.
- The Driver Error Diffusion measure similarly allows relative comparison across drivers on their ability to spread errors.

Table 5.3: Summary of the error propagation measures introduced.

Symbol	Equation	Description
$PDS^{i}_{x.y}$	5.1	The *Service Error Permeability* for driver services is the probability that an error in a driver service $ds_{x.y}$ will propagate to an OS-Application service s_i.
$POS^{i}_{x.y}$	5.2	The *Service Error Permeability* for an OS-Driver services is the probability that an error in an OS service used by driver D_x will driver service $ds_{x.y}$ will propagate an OS-Application service s_i.
$PS^{i}_{x.y}$	5.7	The *combined* Service Error Permeability makes no distinction between imported and exported functions.
E^i	5.3	The *Service Error Exposure* is used to compare OS-Application services on their susceptibility to propagating errors.
E^i_x	5.4	The *driver specific* Service Error Exposure is used to compare OS-Application services on their susceptibility to propagating errors. It differs from E^i in that it considers each driver individually.
$SE_{x.y}$	5.5	*Service Error Diffusion* is used to compare OS-Driver services on in their ability to spread errors in the system.
D^x	5.6	*Driver Error Diffusion* is used to identify and compare drivers on their ability to spread errors in the system.

Chapter 6

Error Model Evaluation - What to Inject

Which error model should be used for OS robustness evaluation? What are the trade-offs to make across error models?

The choice of error model for robustness evaluation of OS's influences both the results and the time required to perform the evaluation. This chapter investigates the effectiveness of three error models: bit-flips, data-type errors and Fuzzing errors. It builds on the measures introduced in Chapter 5 and uses these to evaluate the three error models. It helps answering **RQ2** - quantifiable measures - and **RQ4** - what to inject.

An extensive series of fault injection experiments show that the bit-flip model allows for more detailed results, however, at a higher cost in terms of the number of injections required. Fuzzing is found to be cheap to implement but is less precise compared to bit-flips. A novel 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/execution costs.

Furthermore, this chapter shows how the error propagation measures introduced in Chapter 5 can be used in the context of a real system.

6.1 Introduction

When performing a robustness evaluation of an OS several factors influence the choice of the error model used. In many cases the representativeness of the used error model compared to the errors found in a deployed system is of most importance. To be able to estimate the behavior of the OS in an operational setting, the injected errors need to as closely as possible match real errors. However, for COTS components, such as OS's, this is difficult, since a) the type and distribution of real errors may not be known, b) the operational setting might not yet be known, or c) the operational composition of the system may not be known.

This thesis focuses on the robustness aspect of the OS, taking a specific composition of the system into consideration. Robustness of the system is evaluated with a *typical* composition, to identify system vulnerabilities in the form of error propagation paths. This is for instance done when different platforms (OS and hardware) are evaluated on a prototype stage, or when platform robustness is evaluated as part of quality assurance of an entire system. In this type of setting, the evaluator typically lacks access to the source code of system components, or lack the ability (or permissions) to modify it.

Given the context of robustness evaluation (errors are external to the OS) and lack of source code, we target the interface between device drivers and the OS. This interface is typically defined as a set of functions that are called to perform services. The target for injection is the parameters to such functions that carry erroneous information from a driver to the OS.

A key question becomes which error model to choose for the evaluation, i.e., how are erroneous states of the system modeled by injecting errors. This chapter evaluates three contemporary error models based on their effectiveness in finding system vulnerabilities, their ease of implementation and the time required for performing the experimentation. A detailed discussion on each of the criteria investigated is provided.

The error models are evaluated using the three drivers previously introduced: `cerfio_serial` (serial port), `91C111` (Ethernet card) and `atadisk` (CompactFlash).

6.2 Considered Error Models

The considered error models for this study were introduced and described in Section 3.2. This section first briefly discusses the three models. Table 6.1 shows an overview of the three models, showing the number of services

in the OS-Driver interface and the total number of injections performed for each of error model. The number of used services differs across the models, with the serial driver using the most. One can also see that the **BF** model, as expected, incurs the highest number of injections and the **DT** model the fewest.

Table 6.1: Overview of the target drivers.

Driver	#Services	# Injection cases		
		BF	**DT**	**FZ**
cerfio_serial	60	2653	397	1395
91C111	54	1850	283	990
atadisk	47	1486	267	899

6.2.1 Data Type Error Model

The data type (**DT**) error model modifies the value of a parameter based on its data type, and is presented in detail in Section 3.2.1. It has been shown in previous studies that this type of injection is very scalable in terms of the number of data types used in API's, such as POSIX [DeVale et al., 1999]. The total number of data types targeted for the experiments reported here was 22. Given that the average number of services targeted across all three drivers was 54, with typically more than one parameter, this is a fairly low number.

6.2.2 Bit-Flip Error Model

For the bit-flip (**BF**) error model each targeted parameter is considered as a 32 bit value. 32 injections cases are defined, flipping the bits from 0 (least significant bit) to bit 31 (most significant bit) one after the other.

The flipping is achieved by casting the value to an integer and then using the xor-function. This approach is also detailed for instance in [Voas and Charron, 1996]. The new value is then used in the call to the real function.

The **BF** model does not necessarily need to be adapted to the data type used. However, it is beneficial to do so for some specific reasons:

- Reducing the number of bits used for data types using fewer bits reduces the total number of injections required. For instance, the data type `char` uses only 8 bits, whereas the type `int` uses 32. Since injecting in the remaining 24 bits of a `char` does not reflect a software error for this type, the number of bits targeted can be restricted to 8.

- Many of the parameters used in the interface are pointers to a value of some other data type (or `void`). By tracking such relations the pointer target can be used for injection, more closely simulating software errors than targeting the reference pointer values alone.
- A common feature in the driver interface is to use pointers to structures (`struct`'s). Without details on their members, **BF** errors cannot directly target them. Data type tracking facilitates this.

6.2.3 Fuzzing Error Model

The fuzzing (**FZ**) error model uses a pseudo-random generator to generate random values to inject. The targeted service invocation is intercepted and a new random value is chosen to replace the existing value. We use the standard C-runtime function `rand()` to generate the random values. Each target computer (board) stores the generated random value in persistent storage and uses this value as seed to the random generator for the next injection. This way we avoid generating the same value for each injection (which would have been the case had the same seed been used).

For each service targeted fifteen injections with different random values are performed. This number was selected to give a reasonable execution time of the experiments and yet produce useful results. Section 6.5.2 further discusses the number of injections for the **FZ** model.

6.3 Error Propagation

This section details our experimental estimation of the error propagation measures defined in Chapter 5. It is demonstrated how the analytical expressions in the previous chapter can be adapted to assessment with fault injection. A series of simplifications are presented and results and interpretations from large scale fault injection experiments are presented. To shorten the discussion, focus in this section will be put solely on the bit-flip model, with Section 6.4 devoted to the comparison across the three models.

6.3.1 Failure Class Distribution

Table 6.2 shows the failure class distribution for the three drivers using the **BF** model. The atadisk driver has the highest ratio of **Class 3** failures, even though for all three drivers the ratio stays below 4.5% of the injected errors. The cerfio_serial driver has considerably more **Class 2** failures than the other two drivers. This is due to a number of injections for this driver leading to

Table 6.2: The failure class distribution for the **BF** model.

Driver	NF	[%]	C1	[%]	C2	[%]	C3	[%]
cerfio_serial	2060	77.65	38	1.43	481	18.13	74	**2.79**
91C111	1320	71.35	416	22.49	50	2.70	64	**3.46**
atadisk	1117	75.17	300	20.19	3	0.20	66	**4.44**

hangs in OS services used by applications, i.e., services hang unexpectedly. The serial driver, being inherently of blocking nature, is the only driver to show such behavior to a significant extent. The other two drivers have higher **Class 1** ratios instead and all three drivers have roughly the same amount of **Class NF** failures, above 70%. The high number of **Class NF** failures suggest that there is potentially room for reducing the number of injections further, beyond what is already done through the pre-profiling stage.

6.3.2 Estimating Service Error Permeability

Service Error Permeability is the conditional probability that an error appearing in a OS-Driver interface service will propagate to a service in the OS-Application interface, given that one appears (see Equations 5.1 and 5.2). A distinction is made between errors appearing in services provided by drivers (exports) and those provided by the OS itself (imports). A simplification is also made in Equation 5.7 where no such distinction is made.

Service Error Permeability is estimated by the use of fault injection as the ratio between the number of injections performed resulting in a failure to the total number of injections for a given service. Service Error Permeability is calculated the same way for both imported and exported services. We denote, for a driver D_x, the number of injected errors in a service[1] $os_{x.y}$ with $N_{x.y}$ and the number of failures for service s_i with n_i. The *estimated* Service Error Permeability is then calculated as follows:

$$\widehat{SP}^{i}_{x.y} = \frac{n_i}{N_{x.y}} \tag{6.1}$$

Typically one studies each failure class in isolation. In this case n_i is the number of injections resulting in failure of the specific class under study. $\widehat{SP}^{i}_{x.y}$ is used as an estimate of both $PDS^{i}_{x.y}$ and $POS^{i}_{x.y}$ and as such correspond Equation 5.7.

Service Error Permeability can be used to study the relation between tuples of OS-Driver services and OS-Application services. The number of

[1]The calculation for exported services ($ds_{x.y}$) is analogous to that for imported.

services in the OS-Driver interface can be seen in Table 6.1. For each of the services, Service Error Permeability is defined in relation to each service studied at the OS-Application layer. Since Service Error Permeability is a probability, the values will be in the range $[0 \ldots 1]$. It is important to note that a value of 0.0 must not be interpreted as an proof that no errors will propagate along this path. It is only an indication that the likelihood is low, given the error model used. Similarly, a value of 1.0 only indicates that errors are likely to propagate, but is again dependent on the used error model.

As previously mentioned, propagation results are best interpreted by studying the individual failure classes separately. For Service Error Permeability **Class 3** failures are not relevant, since a **Class 3** failure will have the same effect on all application level services, since it renders the entire system irresponsive, either through a hang or a crash. Similarly, error propagation is hard to track for most **Class 2** failures, since the effect needs to be pinpointed to the specific service being the victim of the failure. This would require tracking not only negative reports for each service, i.e., when errors propagate, but also positive reports, i.e., each service call needs to be logged. This would put a tremendous pressure on the tracking mechanism to safely store or forward information on each call. Consequently we have not considered Service Error Permeability for **Class 2** failures, and it thus remains as a future extension to our work.

Our investigation using Service Error Permeability focuses on **Class 1** failures. Many such propagation paths exist (although with many having an estimated propagation permeability of 0.0). Therefore, Tables 6.3, 6.4 and 6.5 for brevity presents only the prominent error propagation paths identified, for each of the three drivers using the **BF** model.

Table 6.3 shows all propagation paths for cerfio_serial. "String compare" is not a specific OS service per se, but an added consistency check for the received echo strings sent by the test application. Similarly, "Serial Echo Error" is the echo server check performed on the host side. Driver services with the prefix COM are exported driver services.

Several observations can be made regarding the propagation paths reported in Table 6.3. First, the table shows that both imported and exported functions can lead to **Class 1** failures. Second, some propagation paths are distinctly more prominent than others, having Service Error Permeability values of up to 1.0, i.e., each injected error lead to a **Class 1** failure reported by the application testing the serial port functionality. Third, a "clustering" effect can be seen, where many OS-Driver services have multiple propagation paths with the same Service Error Permeability value. This indicates that injecting a transient error in these services will corrupt the "state" of the system and cause subsequent service invocations to fail as well. This

Table 6.3: **Class 1** error propagation paths for cerfio_serial, based on Service Error Permeability (SEP), for the **BF** model.

#	OS-Driver	OS-Application	SEP
1	InterruptDisable	CreateFile	1.000
2	InterruptDisable	GetCommState	1.000
3	InterruptDisable	WriteFile	1.000
4	InterruptDisable	SetCommTimeouts	1.000
5	InterruptDisable	ReadFile	1.000
6	InterruptDisable	GetCommTimeouts	1.000
7	InterruptDisable	String compare	1.000
8	InterruptDisable	CloseHandle	1.000
9	memcpy	SetCommState	0.042
10	COM_Open	WriteFile	0.031
11	COM_Open	SetCommState	0.031
12	COM_Open	ReadFile	0.031
13	COM_Open	SetCommTimeouts	0.031
14	COM_Open	String compare	0.031
15	COM_Open	GetCommState	0.031
16	COM_Read	String compare	0.016
17	EventModify	String compare	0.016
18	EventModify	Serial Echo Error	0.016
19	COM_IOControl	String compare	0.010
20	COM_IOControl	GetCommState	0.010
21	COM_IOControl	Serial Echo Error	0.010

is no big surprise considering the type of services involved. For instance, COM_Open, which when failing to open the serial port will cause subsequent service requests by the application fail as well.

Table 6.4 shows the top thirty **Class 1** error propagation paths for the Ethernet driver. One service has a Service Error Permeability value of 1.0, with several other having high permeability values. The Ndis prefix indicates that these services are provided by the Ndis library, a system library provided to support and simplify network card drivers. Similarly, the OS-Application services are also network related, as expected since the test application most affected is using networking services heavily. The most permeable path is surprisingly enough for NKDbgPrintfW, a function which prints debug information used by developers. This suggests that even functions that are not "expected" by developers to cause propagating errors must be

Table 6.4: **Class 1** error propagation paths for 91C111, based on Service Error Permeability (SEP), for the **BF** model.

#	OS-Driver	OS-Application	SEP
1	NKDbgPrintfW	WSACleanup	1.0000
2	NKDbgPrintfW	connect	1.0000
3	NKDbgPrintfW	shutdown	1.0000
4	NKDbgPrintfW	closesocket	1.0000
5	NdisOpenConfiguration	WSACleanup	0.9375
6	NdisOpenConfiguration	connect	0.9375
7	NdisOpenConfiguration	shutdown	0.9375
8	NdisOpenConfiguration	closesocket	0.9375
9	NdisCloseConfiguration	WSACleanup	0.8750
10	NdisCloseConfiguration	connect	0.8750
11	NdisCloseConfiguration	shutdown	0.8750
12	NdisCloseConfiguration	closesocket	0.8750
13	NdisInitializeWrapper	connect	0.5625
14	NdisInitializeWrapper	WSACleanup	0.5625
15	NdisInitializeWrapper	shutdown	0.5625
16	NdisInitializeWrapper	closesocket	0.5625
17	NdisMRegisterMiniport	closesocket	0.4844
18	NdisMRegisterMiniport	shutdown	0.4844
19	NdisMRegisterMiniport	WSACleanup	0.4844
20	NdisMRegisterMiniport	connect	0.4844
21	NdisMRegisterAdapterShutdownHandler	WSACleanup	0.4688
22	NdisMRegisterAdapterShutdownHandler	connect	0.4688
23	NdisMRegisterAdapterShutdownHandler	closesocket	0.4688
24	NdisMRegisterAdapterShutdownHandler	shutdown	0.4688
25	QueryPerformanceCounter	closesocket	0.4688
26	QueryPerformanceCounter	shutdown	0.4688
27	QueryPerformanceCounter	connect	0.4688
28	QueryPerformanceCounter	WSACleanup	0.4688
29	NdisReadConfiguration	shutdown	0.4393
30	NdisReadConfiguration	closesocket	0.4393

used with care.

The clustering of services is again shown clearly, i.e., several application level services show the same Service Error Permeability values. This indicates that they fail together; when one service fail, several other services

fail too. 91C111 had in total 71 propagation paths with a Service Error Permeability value above 0.0.

Table 6.5: Selection of **Class 1** error propagation paths for atadisk, based on Service Error Permeability (SEP), for the **BF** model.

#	OS-Driver	OS-Application	SEP
1	DSK_Init	GetFileTime	1.0000
2	DSK_Init	GetFileInformationByHandle	1.0000
3	DSK_Init	CloseHandle	1.0000
4	READ_PORT_USHORT	CloseHandle	1.0000
5	READ_PORT_USHORT	GetFileInformationByHandle	1.0000
6	READ_PORT_USHORT	CreateFile	1.0000
7	DetectATADisk	CreateFile	1.0000
8	DetectATADisk	WriteFile	1.0000
9	DetectATADisk	CloseHandle	1.0000
10	DetectATADisk	SetEndOfFile	1.0000

Table 6.5 for the CompactFlash driver shows a similar trend to the previous two tables. For this driver, two exported services show up, DSK_Init and DetectATADisk. The application level services in the list are related to file operations, as expected for this driver. In total atadisk has 176 registered propagation paths.

It is important to note that in Tables 6.3, 6.4 and 6.5 a higher value for Service Error Permeability indicates higher likelihood of propagating errors resulting in **Class 1** failures. A lower value may therefore be an indication of proneness to higher severity failures, or to a higher degree of fault tolerance. Service Error Permeability must therefore be used in conjunction with other measures, such as Service Error Exposure and Driver Diffusion.

6.3.3 Service Error Exposure

Service Error Exposure considers all driver-level services' contribution to the failure seen for a specific OS-Application service. Therefore, we analogously to the Service Error Permeability only consider **Class 1** failures also for Service Error Exposure.

The number of services used by each test application was purposely kept low, as a consequence of keeping the test applications small and simple. Tables 6.6, 6.7 and 6.8 show the driver specific Service Error Exposure calculated for each service for injections in cerfio_serial, 91C111 and atadisk, respectively.

The driver specific Service Error Exposure is calculated using Equation 5.4, which is based on the Service Error Permeability values (partially) presented in the previous section. Since it is a sum of probabilities, they are not limited, and no specific interpretation can be made on a single value. Their use lies in the relative comparison across many services. A higher value indicates that a services is more *exposed* to propagating errors from the drivers considered.

Table 6.6: Service Error Exposure values for the cerfio_serial, using the **BF** model.

#	Service	Service Error Exposure
1	String compare	1.0730
2	GetCommState	1.0417
3	SetCommTimeouts	1.0313
4	WriteFile	1.0313
5	ReadFile	1.0313
6	CreateFile	1.0000
7	CloseHandle	1.0000
8	GetCommTimeouts	1.0000
9	SetCommState	0.0729
10	Serial Echo Error	0.0260

All three tables show the same clustering effect observed for failures, indicated by observing the same Service Error Exposure value. That this is the case is not surprising. Considering for instance CreateFile and CloseHandle in Table 6.8 it is understandable that if CreateFile returns with error, then its handle will be invalid. A subsequent try to close it will also return an error, and consequently the error propagates to both services.

Another useful piece of information can be seen in Table 6.6 which shows "String compare" to have the highest Service Error Exposure value. Since this is a check performed on the received data and no returned error code for an OS service this indicates that in some cases erroneous data can be received without any other service indicating an error. This corresponds to a silent error, suggesting that data integrity checks might be needed for critical data. Similarly, "Serial echo error" signals that the data sent to the Host Computer in some cases is corrupted, suggesting that similar data integrity checks may be needed at the receiving side as well.

In Table 6.7 services 5-15 are from the test application for atadisk, indicating that in some cases injecting errors in the interface for 91C111 can cause **Class 1** failures also for applications not using the faulty driver.

Table 6.7: Service Error Exposure values for 91C111 using the **BF** model.

#	Service	Service Error Exposure
1	connect	6.5020
2	shutdown	6.5020
3	WSACleanup	6.5020
4	closesocket	6.4083
5	CreateFile	0.1250
6	CloseHandle	0.1250
7	ReadFile	0.0938
8	sizeof	0.0625
9	GetFileSize	0.0625
10	GetFileTime	0.0313
11	GetFileInformationByHandle	0.0313
12	SetEndOfFile	0.0313
13	WriteFile	0.0313
14	SetFilePointer	0.0313
15	DeleteFile	0.0313
16	getaddrinfo	0.0208

Table 6.8: Service Error Exposure values for atadisk using the **BF** model.

#	Service	Service Error Exposure
1	CloseHandle	30.1790
2	CreateFile	30.1790
3	ReadFile	22.6436
4	sizeof	15.0957
5	GetFileSize	15.0832
6	WriteFile	7.54787
7	SetFilePointer	7.54787
8	DeleteFile	7.54787
9	SetEndOfFile	7.54787
10	GetFileTime	7.53537
11	GetFileInformationByHandle	7.53537

6.3.4 Service Error Diffusion

When considering which OS-Driver *services* are more likely to spread errors in the system one can use the Service Error Diffusion measures, which considers one service's proneness to spread errors. Since we are considering driver

services specifically we do not have the failure class restrictions that apply to application level measures. We will therefore concentrate on the most severe failure class, **Class 3** failures.

Service Error Diffusion is defined in Equation 5.5 as a sum over all application level services. Since all services are affected uniformly, this translates into a simple scaling of the effects with the number of services used. A simplified expression can therefore be applied, where the application level services are not accounted for individually. This simplified version is shown in Equation 6.2, and is defined for driver D_x and service $s_{x.y}$ (either: $os_{x.y}$ or $ds_{x.y}$):

$$\overline{SE}_{x.y} = \frac{n_{x.y}}{N_{x.y}} \tag{6.2}$$

where $n_{x.y}$ is the number of **Class 3** failures and $N_{x.y}$ the number of injections performed for service $s_{x.y}$, as above.

Table 6.9: **Class 3** Service Error Diffusion values for cerfio_serial using the **BF** model.

#	Service	Service Error Diffusion
1	memset	0.3125
2	memcpy	0.2083
3	MmMapIoSpace	0.1528
4	LoadLibraryW	0.0909
5	FreeLibrary	0.0625
6	DisableThreadLibraryCalls	0.0625
7	LocalAlloc	0.0313
8	SetProcPermissions	0.0313
9	CreateThread	0.0234
10	HalTranslateBusAddress	0.0236

Tables 6.9, 6.10 and 6.11 present the non-zero valued services for the three drivers. One service stands out among the data, wcslen, for atadisk, which has a Service Error Diffusion value of 1.0, which means that all injected errors for this service resulted in a **Class 3** failure. This makes this service a top candidate for further robustness enhancement. Comparing the number of services in these tables with the number of services used in the OS-Driver interface for the three drivers (Table 6.1) one can observe that a small number of services give rise to all **Class 3** failures, for all three drivers.

Furthermore, it can be seen that some services cause severe failures for all three drivers, such as memset and memcpy. These are low-level system

Table 6.10: **Class 3** Service Error Diffusion values for 91C111 using the **BF** model.

#	Service	Service Error Diffusion
1	memset	0.2708
2	NdisAllocateMemory	0.1250
3	DisableThreadLibraryCalls	0.1250
4	QueryPerformanceCounter	0.0938
5	LoadLibraryW	0.0909
6	NdisMSynchronizeWithInterrupt	0.0781
7	FreeLibrary	0.0625
8	VirtualCopy	0.0313
9	NdisMSetAttributesEx	0.0188
10	RegOpenKeyExW	0.0093
11	NdisInitializeWrapper	0.0078
12	NdisMRegisterInterrupt	0.0077
13	KernelIoControl	0.0063

Table 6.11: **Class 3** Service Error Diffusion values for atadisk using the **BF** model.

#	Service	Service Error Diffusion
1	wcslen	1.0000
2	wcscpy	0.2727
3	memcpy	0.2708
4	memset	0.1875
5	DisableThreadLibraryCalls	0.0625
6	LocalAlloc	0.0313
7	MapPtrToProcess	0.0313

functions present in many drivers and the Service Error Diffusion is comparable across all three drivers indicating that for these drivers propagation is independent from the driver itself. If generic error detection and recovery mechanisms could be defined for these two services 84 **Class 3** failures could be removed across all three drivers for the **BF** model. This corresponds to 41% of the **Class 3** failures reported for the experiments.

6.3.5 Driver Error Diffusion

While Service Error Diffusion is used to identify individual services that are more likely to spread errors present at the OS-Driver interface this may become hard to overview and services may be spread across multiple drivers making any specific driver level improvement efforts costly. In this case one may want to focus on the *driver* that is more prone to spreading errors, rather than individual services. To this end we use Driver Error Diffusion.

Driver Error Diffusion can similarly to Service Error Diffusion be simplified for **Class 3** failures. Equation 6.3 presents the estimated Driver Error Diffusion not considering application level services. Driver Error Diffusion thus transforms to a sum of Service Error Diffusion values as follows:

$$\overline{D}^{x} = \sum_{\forall y} \overline{SE}_{x.y} = \sum_{\forall y} \frac{n_{x.y}}{N_{x.y}} \tag{6.3}$$

Table 6.12 shows the Driver Error Diffusion values for all three drivers using the **BF** model. The values presented are calculated with the simplified expression in Equation 6.3.

Table 6.12: Driver Error Diffusion for all three drivers considering **Class 3** failures.

Driver	Diffusion
cerfio_serial	1.00
91C111	0.93
atadisk	1.86

From Table 6.12 one can see that atadisk is clearly more prone to diffusing errors in the system. 91C111 and cerfio_serial are very close, with cerfio_serial having slightly higher value. Considering these values an evaluator might consider devoting extra resources to ensuring that atadisk does not contain errors.

6.4 Comparing Error Models

There are many criteria one could have for selecting the error model to use. Depending on the goal of the evaluation, criteria such as execution time or number of found failures may not be equally important. Therefore, we compare the three models on a wide range of criteria. First, a discussion on the uses and implications of each evaluation criteria is presented, followed

by a presentation and interpretation of the results. Table 6.13 shows an overview of the results for the three models.

The following criteria are considered when comparing the error models:

- **Number of failures found:** The absolute number of failures is important as a higher number may give more insight into how the system can fail and consequently give better feedback to developers of the system to improve it.
- **Number of injections and execution time:** The number of injections influences the time required to perform the evaluation. The relationship is not linear, since the execution time also depends on the outcome, but more injections generally means longer execution time.
- **Injection efficiency:** The efficiency of the injections is measured as the number of failures per injection. This measure helps making a trade-off between the two previous criteria.
- **Coverage:** The term coverage is here used to compare different models ability to pinpoint certain services as potentially vulnerable. Since no information on the *real vulnerabilities* exist, the comparison is based on a best-effort strategy, where the relative coverage across models is compared.
- **Implementation complexity:** The complexity of the implementation is a measure of the effort required to implement the error model. Since no lab experiments with real developers have been conducted, the comparison remains subjective. However, there are clear and distinct differences in the implementation effort needed for the studied models.

Table 6.13: The number of experiments is shown for each driver, error model and failure class.

Driver	Error Model	No Failure	%	Class 1	%	Class 2	%	Class 3	%
cerfio_serial	**BF**	2060	77.65%	38	1.43%	481	18.13%	74	**2.79%**
	DT	264	66.50%	65	16.37%	53	13.35%	15	**3.78%**
	FZ	908	65.09%	20	1.43%	401	28.74%	66	**4.73%**
91C111	**BF**	1320	71.35%	416	22.49%	50	2.70%	64	**3.46%**
	DT	203	71.73%	66	23.32%	1	0.35%	13	**4.59%**
	FZ	574	57.98%	389	39.29%	0	0.00%	27	**2.73%**
atadisk	**BF**	1117	75.17%	300	20.19%	3	0.20%	66	**4.44%**
	DT	172	64.42%	90	33.71%	1	0.37%	4	**1.50%**
	FZ	565	62.85%	310	34.48%	4	0.44%	6	**0.67%**

6.4.1 Number of Failures

The absolute number of failures that an error model triggers is important from a feedback perspective. The more cases of triggering vulnerabilities shown, the easier it will be to identify the vulnerability and possibly remove it.

Table 6.13 shows the number of failures found for each of the four failure classes, error model and driver. From the table it can clearly be seen that the **BF** model, having the most injections, also incur the most **Class 3** failures. The number of failures for the **BF** model is comparable across all three drivers. For the other two models there are differences in the number of **Class 3** failures, indicating that there are differences between the drivers in their ability to spread errors.

Table 6.13 further substantiate the fact that cerfio_serial is more prone to **Class 2** failures than the other two drivers. That this behavior is driver related, and not dependent on the error model is further supported by the fact that the percentage of injections leading to **Class 2** failure is distinctly higher for all error models for cerfio_serial, compared to the other two drivers. However, it is important to note that since the approach is experimental, all results are dependent on the specific setup used. In this case all test applications are written in a straight-forward manner, without any explicit fault-tolerance mechanisms. Such mechanisms will most probably change the results of the evaluation, and the results presented indeed suggest that such mechanisms are needed.

Furthermore, many injections do not result in any observable error propagation (58-78%) within the time used for each injection, i.e., no observable deviation from the expected behavior was observed. This is in line with multiple previous studies, e.g., [Durães and Madeira, 2003], [Gu et al., 2003] and [Jarboui et al., 2002a]. Experiments in the **Class NF** category are either masked by the system, for instance parameters not used in this context 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. It is important to note that all errors injected were in fact activated, since the pre-profiling eliminates services not used prior to injection (Section 4.6). The high number of **Class NF** experiments indicates that there is room for improving the selection of injection cases beyond the pre-profiling already carried out.

6.4.2 Execution Time

The number of injections performed and the time required for executing the experiments are related. An increase in the number of injections will mean increased execution time. However, the outcome of the experiments influence the execution time. An injection that does not lead to error propagation can be considerably faster than one that leads to a system hang, requiring first that the hang, is detected and then a reboot of the system.

Table 6.14 reports the execution times for the injections performed. The times reported include only the actual execution time, not implementation, setup and off-line processing times.

Table 6.14: Experiment execution times.

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

From the table it can clearly be seen that the **BF** model, having the most injections, also has the longest execution time. The time required for **BF** is roughly twice as much as for **FZ** and seven to eight times as much as **DT**. As noted above the outcome of the experiments influence the execution time, and this might differ across drivers. In our setup cerfio_serial and atadisk both take longer time when failing, which also influences the execution time.

A factor influencing the *effective* experiment time is the degree of operator involvement. The operator is required to specify the experiment to run (which driver, error model etc). The setup time is the same for all models. Additionally, some injections force the system into a state where it cannot automatically reboot, requiring a manual reboot by the operator. Consequently, without external reboot mechanisms the experiment is delayed until the operator is notified and can perform the reboot, which can prolong the execution time substantially. This additional delay is not part of the execution time in Table 6.14 since no assumption is made on the presence of the

operator. The issue of manual reboots is further discussed in Section 6.6.

6.4.3 Injection Efficiency

The absolute number of failures each error model gives rise to also needs to be put in contrast to the number of injections performed, to give an indication of the efficiency of the model. Figure 6.1 graphically shows the data in Table 6.13. It can be seen that the overall trend is similar across all three drivers. This would clearly favor models incurring fewer injections, especially **DT**, but also the **FZ** injections

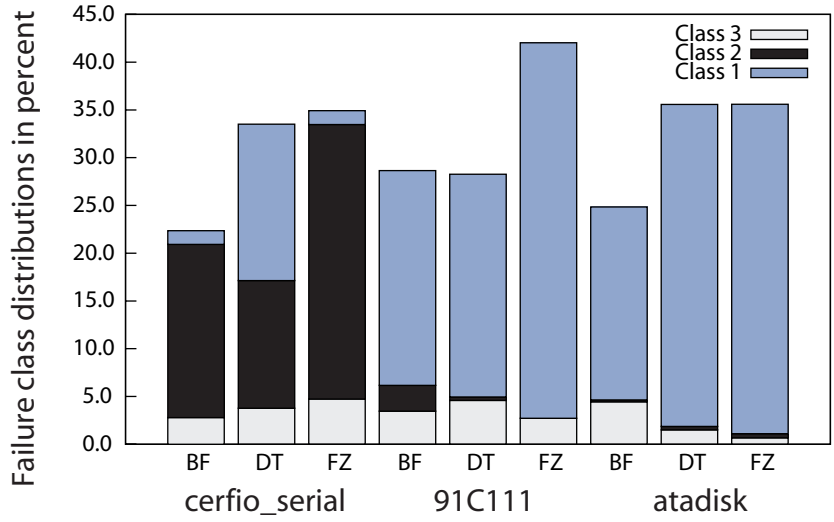

Figure 6.1: Injection efficiency.

The other criteria used are quantitative in nature, where a relative scale of "goodness" can be defined and used to rank the models. One aspect that can not be compared quantitatively is a model's ability to assess the "true" propagation patterns of the system. An *efficient* model may be one giving a bigger "bang for the buck", at least if finding failure triggering vulnerabilities, but may still be misleading. A separate concern is therefore to inspect the differences in propagation results for the three models studied, represented by the Driver Error Diffusion values for all three models and drivers in Table 6.15.

Table 6.15 shows that there are indeed differences across the results of the three models. **DT** and **FZ** identify the serial driver to be the most

Table 6.15: Driver Diffusion for **Class 3** failures.

Driver	**BF**	**DT**	**FZ**
cerfio_serial	1.00	1.50	1.93
91C111	0.93	0.98	0.59
atadisk	1.86	0.63	0.19

vulnerable driver, whereas **BF** pin-points atadisk to be the most vulnerable, with the serial and Ethernet drivers having very similar values. It can also be observed that the results for atadisk is clearly more spread than for the other two drivers, with 91C111 being fairly consistent across all three models. This indicates that the services for 91C111 giving rise to **Class 3** failures suffer from "uniform" vulnerabilities, i.e., any small change in the data supplied will trigger failures. On the contrary, services used by atadisk have vulnerabilities triggered only for more specific values, in this case triggered by bit-flips.

A straightforward view of the results is presented in Table 6.13, where the number of experiments in each failure class is detailed, both in actual numbers and as percentages of all injections.

The first observation is that for all drivers and error models the percentage of injections ending up as **Class 3** failures is below 5%, indicating that the OS is capable of handling many perturbations and avoiding a catastrophic failure.

Table 6.16: **Class 3** Service Error Diffusion values for cerfio_serial using the **DT** error model.

#	Service	Service Error Diffusion
1	MmMapIoSpace	0.5000
2	LocalAlloc	0.4000
3	LoadLibraryW	0.2500
4	SetProcPermissions	0.2000
5	memcpy	0.0909
6	CreateThread	0.0625

When comparing the error models, clear differences can be identified. For instance, where for Driver Error Diffusion **DT** previously identified cerfio_serial as the most diffusive driver, 91C111 has a higher ratio of **Class 3** failures for this error model. Only considering the ratio may in this case be misleading, as cerfio_serial in this case has more services with high Service Error Diffusion compared to 91C111 for **DT**, as seen from Tables 6.16 and

Table 6.17: **Class 3** Service Error Diffusion values for 91C111 using the **DT** error model.

#	Service	Service Error Diffusion
1	LoadLibraryW	0.2500
2	memcpy	0.1818
3	NdisAllocateMemory	0.1764
4	RegOpenKeyExW	0.1500
5	memset	0.1333
6	NdisMRegisterInterrupt	0.0555
7	NdisMSetAttributesEx	0.0322

6.17. Similarly, where Driver Error Diffusion with the **BF** model indicates atadisk to be by far the most diffusive driver, the ratios of **Class 3** failures show them to be relatively close. This is an effect of diffusion being a "sum of probabilities". Diffusion shows that atadisk has more services (especially wsclen with 1.0) with a high permeability than 91C111 (Tables 6.10 and 6.11).

6.4.4 Coverage: Identifying Services

Table 6.18 depicts services incurring **Class 3** failures and the number of failures for each service/error model. **BF** outperforms the other error models, both in terms of the number of identified vulnerable services, and the total number of **Class 3** failures (more clearly visible in Table 6.13). **BF** identifies 22 individual services, **DT** 12 and **FZ** 11 services.

Considering which services the different models uniquely identifies, i.e., services which only one model identifies, again **BF** outperforms **DT** and **FZ**. **BF** identifies seven services and **FZ** two, which are not identified by any other error model. **DT** identifies no such unique services. From the number of failures identified, **FZ** identifies several services with only one case, suggesting that the random nature of he **FZ** error model has a higher probability of finding unique service vulnerabilities. Whereas the more systematic injections performed for **BF** typically reveal more than one failure.

6.4.5 Implementation Complexity

The implementation cost, measured as the time required for the implementation of an error model is naturally subjective. The amount of programming experience, knowledge of the area and the the availability of tools and docu-

Table 6.18: Services identified by **Class 3** failures.

#	Service	**BF**	**DT**	**FZ**
1	CreateThread	3	1	0
2	DDKReg_GetWindowInfo	0	0	1
3	DisableThreadLibraryCalls	8	0	1
4	FreeLibrary	4	0	1
5	HalTranslateBusAddress	3	0	5
6	KernelIOControl	1	0	0
7	LoadLibraryW	2	2	0
8	LocalAlloc	2	4	0
9	MapPtrToProcess	2	1	1
10	memcpy	46	3	34
11	memset	74	3	18
12	MmMapIoSpace	11	9	27
13	NdisAllocateMemory	12	3	0
14	NdisInitializeWrapper	1	0	0
15	NdisMRegisterInterrupt	1	1	0
16	NdisMSetAttributesEx	3	1	1
17	NdisMSynchronizeWithInterrupt	5	0	0
18	QueryPerformanceCounter	3	0	0
19	RegOpenKeyExW	1	3	0
20	SetProcPermissions	1	1	9
21	VirtualAlloc	0	0	1
22	VirtualCopy	4	0	0
23	wcscpy	6	0	0
24	wcslen	11	0	0

mentation all influence the required time. However, some observations during the course of implementing the injection framework suggests that there are differences across the error models.

Whereas Table 6.14 shows that **BF** and **FZ** are clearly more expensive in terms of execution time compared to **DT**, a major drawback with the **DT** error model is the cost for implementation. The difference lies in that for every function in the service interface the data type of each parameter needs to be tracked, such that the right injector can be chosen. **BF** and **FZ** on the other hand do not have this requirement, making their implementation costs considerably cheaper. Both use simple injection technologies, making the two models comparable in terms of implementation costs. Additionally, the time

required for defining the injection cases for each data type is considerable higher for the **DT** model.

The cost for the **DT** model could potentially be reduced by use of automatic parsing tools and/or reflection-capable programming languages. The implementation cost is also a one-time cost for each driver which might be acceptable if the experiments are to be repeated in a regression testing fashion. Furthermore, the cost might be acceptable in comparison to other verification efforts used. Further research on the true costs for such error models is indeed warranted.

It is also important to note that even though the **BF** and **FZ** model do not require data type tracking they can benefit from it. By knowing the data type used, the number of bits targeted can be limited for data types not using all bits anyway (such as 8-bit integers). This technique has been applied in the experiments presented in this thesis.

6.5 Composite Error Model

Two major findings can be extracted from the previously presented results, namely: a) that **BF** identifies the most **Class 3** failures, both in terms of absolute number and in the number of individual services identified, and b) **FZ**, even though not triggering as many failures as **BF** identifies, identifies **Class 3** failures for additional services, beyond those identified by **BF** and **DT** combined.

Even though these results must be interpreted in the context of our case study they show different error models, although being injected on the same level and therefore being comparable, have different properties. It would be desirable to combine the models (**BF** and **FZ**) into a combined model, drawing on the strengths of both models. We do this by combining the two models into a so called *composite* model (**CO**).

For the composite model we will focus on the **Class 3** failure class as it in most cases is the critical class for robustness evaluation. The main hurdle for use of the **BF** model was the comparatively high number of injections. Thus we first focus on reducing the number of injections required by studying the impact each bit injection has and selecting only a subset of the bits for injections. Furthermore the number of injections required for the **FZ** model is studied to find a reasonable injection set. The following subsections will detail these studies.

6.5.1 Distinguishing Control vs Data

As mentioned, the number of required injections for **BF** increases the required execution time dramatically compared to the other two models. The high number of cases for each parameter is due to the fact that one injection is made for each bit in the parameter value, thus typically 32 injections per parameter. For a parameter of type `int` holding an integer value this uniform injection may represent a valid selection of error values. However, in many cases, especially for device drivers written in C, an integer value may not actually be used to represent all 32-bit integer values. Instead only a small subset of the values are used, and consequently only a small subset of the bits. It is therefore conceivable that not all 32 bits need to be targeted.

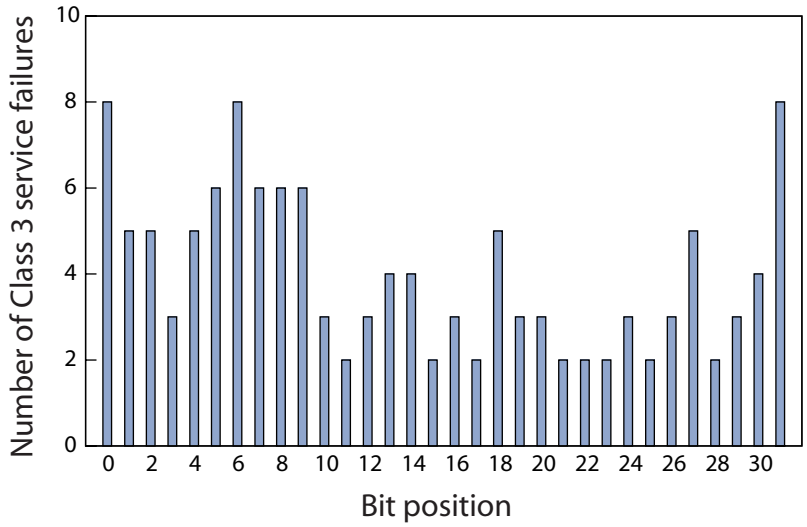

Figure 6.2: The number of services identified by **Class 3** failures by the **BF** model.

Figure 6.2 shows for each of the targeted bits how many services where identified having **Class 3** failures when bit-flips were injected in that bit. It can from the figure clearly be seen that there is no uniform distribution across the bits. The lower order bits, bits 0-9, identify more services than the other bits, with the exception of the most significant bit (31) which typically has special significance, such as being the sign bit for signed data types.

Figure 6.2 show the number of specific services identified with vulnerabilities for each service, but not whether the services identified by bit 0 are the same as those for bit 1. For this we use Figure 6.3 which shows the cu-

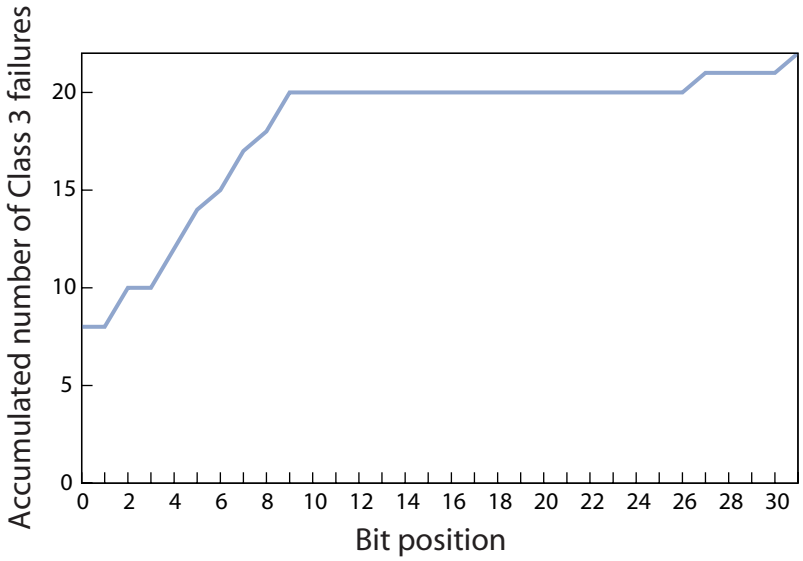

Figure 6.3: Moving from bit 0 and upwards the number of services increases until bit 10.

mulative number of identified services. Reading the figure from left to right it shows how many services are identified by first injecting in bit 0, followed by addition of injections in bit 1, then 2 and so on. It shows that the set of vulnerable services increases in size up till bit 9 where twenty different services were identified. Another service is found at bit 27 and the last one at bit 31, which as previously mentioned often have special significance. Closer inspection reveals that the service first found at bit 27 is also identified by bit 31.

The observations made regarding the impact of individual bits suggests that the subset of bits for which bit-flip injections should be made can be reduced to only include bits 0-9 and 31, i.e., in total 11 bits compared to the original 32 bits considered. This translates into a reduction of injections by 49.8% in total compared to the full set used previously. Some fault injection tools support such specification of injections, like Xception [Carreira et al., 1998].

When studying the parameters used for services identified by **BF**, but not by **FZ**, shows a clear trend: many of these parameters are *control values*, like pointers to data, handles to files, modules, functions etc. Such parameters are intuitively more sensitive to *small* value changes, i.e., changes caused by flipping lower order bits. As an example consider a pointer to some data stored in an allocated memory area. Large changes to the pointer value

(changes of higher order bits) are more likely to cause the erroneous value to lie outside the allocated memory area than a smaller change. Memory access errors though being severe can be detected by the system (in some cases), however, small changes may be harder to detect and may cause failures that are harder to prevent. Similarly, the **FZ** model, using random values, is more likely to choose values that are well outside the expected data range. This is reflected in the difference observed between the two models. On the other hand, **FZ**'s random nature means it can find vulnerabilities not found by more "structured" approaches, reflected in the fact that **FZ** identifies several additional service failures on top of those found by **BF** alone.

6.5.2 The Number of Injections for Fuzzing

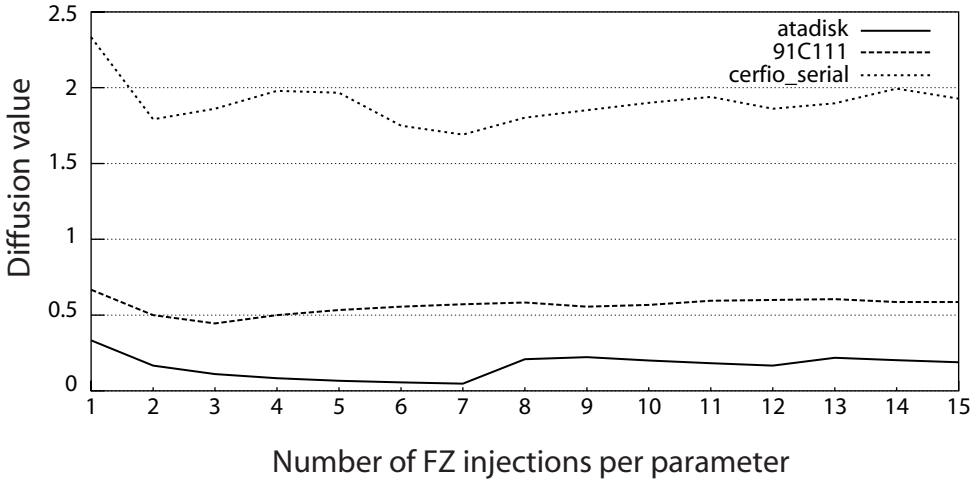

Figure 6.4: Stability of Diffusion for the **FZ** model with respect to the number of injections.

Since the **FZ** model, in contrast to **BF** and **DT**, requires the evaluator to set the number of injections to be performed, this becomes an important question for judging the usefulness of the **FZ** error model. Previously, we have already shown how the fifteen injections performed provide comparable and useful results. One question remaining is whether fifteen injections is sufficient for assessing error propagation. Figure 6.4 shows how the Driver Error Diffusion values stabilize as the number of injections is increased. The values stabilize after roughly ten injections, but there are some differences across the drivers suggesting that stabilization may also be driver dependent.

However, for these three drivers the curves remain clearly separated for any number of injections shown.

6.5.3 Composite Model & Effectiveness

The results from the previous section clearly show the need for using multiple error models. When resources are plentiful it is therefore recommendable to use multiple error models to get comprehensive coverage. To decrease the cost of evaluation (in implementation and experimentation time/effort) this may not be desirable. In this section we therefore propose and evaluate a *composite* model (**CO**). The new **CO** model combines *the **BF** model using least significant bits (together with the most significant one) alongside a series of **FZ** experiments*. Section 6.5.2 indicates that even as few as ten **FZ** injections give stable Driver Error Diffusion values. This number was chosen to decrease the overall number of injections, but it is reasonable that more **FZ** injections will increase the probability of finding "rare" cases.

Table 6.19: Diffusion values for the three drivers

Driver	Diffusion
cerfio_serial	1.58
91C111	0.91
atadisk	1.01

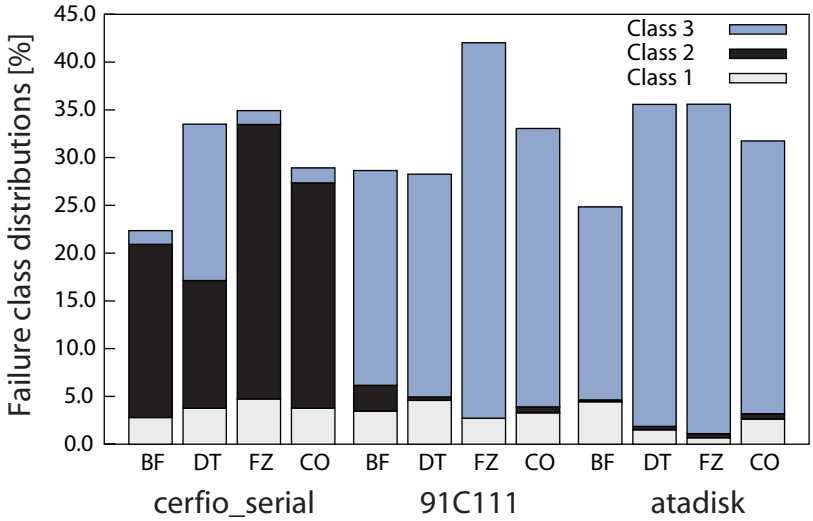

Figure 6.5: Failure class distribution for **CO** compared to **BF**, **DT** and **FZ**.

The **CO** model is evaluated by considering the **BF** injections in bit 0-9 and bit 31, together with the first ten **FZ** injections. **CO** identifies all services having **Class 3** failures but one (VirtualAlloc) compared to the full set of **BF** + **FZ** injections. An overview of the results is shown in Figure 6.5. The figure shows that the results achieved with this subset of injection is comparable with the results for the other models alone, making its ability to assess propagation effects on par with the other models. This is further substantiated by the Driver Error Diffusion values for the **CO** model presented in Table 6.19.

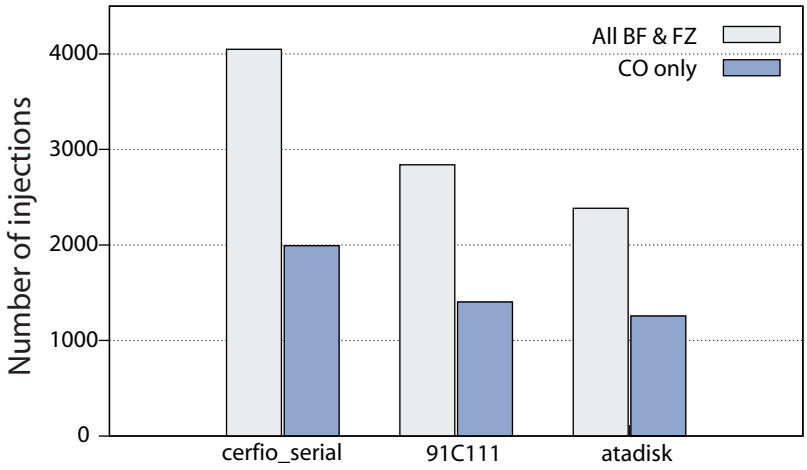

Figure 6.6: The number of injections for the composite model compared to bit-flips and fuzzing together.

The number of injections required is compared in Figure 6.6. The figure clearly shows that the number of injections is well below half of the experiments required for the full set of **BF** and **FZ**, which translates into considerable save in experimentation time.

6.6 Discussion

This section discusses some important aspects of the work and the results provided for the case study.

Imports vs. Exports

In the system model presented in Chapter 3 a distinction is made between imports and exports. The data presented in this chapter has not made any

distinction between imported and exported services. The reason is simple, in no case were an injection in an exported service able to trigger a **Class 3** failure in the system.

Table 6.20: A comparison between the results for imported and exported services.

Driver	Interface	**Class 3**	**Class 2**	**Class 1**
cerfio_serial	export	0	31	3
	import	74	450	35
91C111	export	0	0	0
	import	64	50	416
atadisk	export	0	0	43
	import	66	3	257

There could be many reasons for this effect. First of all the number of services in the exported interface is lower than the imported (typically around 10 compared to 30-40), which makes it reasonable that most failures will be found in the larger set. Secondly, the exported interface is a standard interface, used by many drivers. It is therefore likely that the OS takes special care to validate misuses of these services and that major flaws have already been detected during testing. Thirdly these services very closely matches OS-Application services, which suggests that the amount of additional work done by the OS is small for these services. Consequently, the effects errors can have will be mostly on the applications themselves, as **Class 2** and **Class 1** failures.

Experimental Techniques

As with any experimental evaluation technique it is important to consider the limitations of the chosen approach. Uncertainties are introduced at multiple levels and they need to be identified and understood to properly interpret the results.

First of all the error model used and evaluated here are *generic*. They are not based on any specific system scenario, but rather represent the subset of data level errors occurring at the OS-Driver interface. If system-specific faults are to be considered more specific error models need to be included as well or instead of the generic ones presented here. Furthermore, even for the subset of conceivable errors appearing at this interface only a small fraction is actually used. The results provided in this chapter supports our belief

that these are representative for a wider selection of errors since even though there are differences between the models, they overall show a similar pattern.

Secondly, the results presented are influenced by external factors such as the selected workload and the composition of the system (the selected OS components). To minimize the variability of the results and to minimize unpredictable influences we use a targeted generic workload and minimize the number of system components (see section 3.3). For a specific system a workload closely resembling the expected one should be used as well, and the system should be composed such that it resembles the final system as close as possible.

Finally, the experimental procedures themselves may be a source of influence on the final results, both in terms of what and how the outcomes are observed, and any undesirable influence caused by the added software used for the experimentation. We have followed common practise in the selection of observation points, namely from a user perspective. Furthermore, we have minimized the number of components required for the execution of the experiments and made efforts to minimize any potential impact they might have. However, as the experiments have not yet been repeated in a similar environment we cannot be 100% certain that no such influence exists.

Vulnerability vs. Bug

An interesting question arises when studying the results presented is whether the presence of **Class 2** and **Class 3** failures is an indication of "bugs" in the system. The answer is both yes and no, since a vulnerability discovered by experimental fault injection may, or may not be present in a deployed system[2]. A common case, especially for device drivers, is that documentation states that certain rules should be obeyed when using specific services. These rules may not be enforced, for performance reasons, e.g., the cost of checking each parameter value for a driver may be too high. Instead a "gentlemen's agreement" is used, where the OS assumes that services are not misused. If a driver does misuse such a service it may not be considered a bug in the system in the traditional sense, but surely is a robustness vulnerability. Such vulnerabilities have recently attracted more attention in research, since they constitutes threats to the system's security where an attacker can render the system non-responsive and thereby threaten the availability of the system.

As previously mentioned we focus on robustness and therefore consider all discovered failures of the system vulnerabilities.

[2]Hence the use of the term *vulnerability* instead of *bug*.

Random vs. Structured Injection

There is an ongoing debate in the testing community whether random testing is an appropriate testing technique in general, or one should aim for more classical techniques, such as equivalence calls or boundary value testing Hamlet [2006]. Our choice of models reflect this conflict, where **DT** and **BF** represent more structured approaches, whereas **FZ** introduces randomness. The results presented also supports many researchers view on random testing, namely that it has many weaknesses, but may in some cases be preferred, because no alternative is definitely better.

The advantage of structured approaches is that they can draw from existing knowledge when selecting injection cases, this is for instance very clear in the case of **DT**. **DT** is on the other hand limited to the ability of the evaluator to select appropriate injection cases, an inherently very difficult task. **BF** makes this task simpler, by defining injections based on the representation of the injection target (the parameter value), but is still limited to the specific modifications done by flipping the bits. **FZ** imposes no such restrictions, simply choosing randomly selected values.

The results clearly show that **BF** finds more vulnerabilities, in more services, and that **DT** is clearly more efficient (requires fewer injections) than **FZ**. However, **FZ** is able to identify services beyond the set identified by **BF** and **DT**, also with a limited number of injections.

Overall, the results favor using multiple error models, and the composite model shows that using the two models requiring the least implementation effort can give very promising results.

Operator Involvement

The degree to which the operator (the person setting up the experiments and supervising them) is involved in the process affects the *effective* time required to perform the experiments. First of all the operator is required to configure the system and specify which experiments to perform. For the framework used in this thesis this time is the same for each error model. The second task is to supervise the experiments and when needed manually restart boards that have hung. For the experiments presented here, it happens in many cases that the system crashes without being able to automatically restart itself. In these cases the operator (in this case the author self) has to manually force a cold restart of the target board. Table 6.21 presents data on the amount of manual reboots required.

When the system is unable to restart automatically experiments are delayed until the operator notices the problem and takes action. The Host

Table 6.21: The percentage of **Class 3** failures that required the boards to be manually rebooted by the evaluator.

Driver	Error Model	Manual reboots [%]
cerfio_serial	**BF**	8.1
	DT	46.7
	FZ	15.2
91C111	**BF**	54.7
	DT	23.1
	FZ	37.0
atadisk	**BF**	9.1
	DT	25.0
	FZ	16.7

Computer is equipped with a watchdog timer that notifies the operator if no log messages have been received within the last four minutes, well beyond the execution time of an experiment that is automatically self rebooted (which is also triggered by a watchdog timeout as described in Section 4.5.3).

Since only eleven services overall have failures requiring the operator to manually reboot the machines, the number of such reboots for a given driver and error model depends on how these services are used, giving rise to the differences reported in Table 6.21.

A further development of the injection framework would be to implement the hardware required to automatically reboot the target board when the host machine watchdog is triggered.

As described previously, a generic time penalty is assigned every manually rebooted experiment. The times reported in Table 6.14 are therefore not considering the operator time.

The results in Table 6.21 indicates the efficiency achievable with watchdog timers monitoring system processes. For many of the injected errors a system level monitoring watchdog, which restarts failing processes could increase the availability of the system. This requires that "micro-rebooting" of the targeted components is possible. Such strategies have for instance been deployed in [Candea et al., 2004; Herder et al., 2007].

Extraction of Bit Profiles

Section 6.5.1 uses the bit profiles to find which bits find the most service failures. A very practical question is of course if the found profile is generally applicable to many systems or if the results are specific to the setup used for

these experiments. To get a clear picture of this more drivers and systems would have to be profiled, and relations to specific drivers and service types further evaluated.

6.7 Related Work

There have been several efforts made to compare error models and to find representative errors to inject for specific systems and purposes. In this section we review some of the most relevant efforts that relate to the work in this thesis. We have therefore limited the selection to those that consider software faults, especially with focus on OS's and robustness evaluations. A longer treatment of related work is found in Chapter 2.

Albinet et al. have also studied errors in device driver by injecting errors in the OS-Driver interface [Albinet et al., 2004]. The error model used is **DT** using the terminology in this thesis, but with a lower number of injection cases compared to ours. Injection on a Linux-based system shows a higher ratio of kernel hangs than observed in this chapter. This may be due to differences between the two systems, or the drivers tested.

Arlat et al. study the dependability of microkernel-based system using fault injection. The MAFALDA tool is used to inject faults and perform failure mode analysis. The error model consists of both injections in parameter values to microkernel services and injections in both code and data segments of a component. For both locations bit-flips are used to simulate both software and hardware faults. The type of injections is partially similar to ours (service parameters) but encompass only the **BF** model. The results for parameter injection show a very low ratio of kernel hangs and crashes, possibly suggesting that microkernel architectures are better at handling these types of errors than monolithic systems.

Jarboui et al. compare **BF** and **DT** error models for the Linux kernel [Jarboui et al., 2002b,a]. Firstly they also find a distinct difference in the number of injections required for **BF** compared to **DT**. Similarly to our results the number of severe outcomes are small and both models show similar behavior in terms of failure mode distributions. The also compare these results with errors injected inside the kernel code (which we have not done) and observe a higher ratio of severe outcomes suggesting that there is a certain level of parameter validation performed for kernel services present in the system.

The fuzzing model was first used on for utility programs for UNIX systems [Miller et al., 1990] and has later been applied also for protocols and application interfaces [Howard and Lipner, 2006; Fre]. Oehlert makes a dis-

tinction between intelligent and unintelligent fuzzing [Oehlert, 2005]. The former more closely resembles our **DT** model, where knowledge about the format used is assumed. The latter does not require any prior knowledge and may therefore better explore unexpected inputs, which was also shown in the results presented in this chapter.

To the best of our knowledge this thesis represents a first effort to use fuzzing at the OS-Driver interface and the first time fuzzing is quantitatively compared to other error models in a comparable setting.

6.8 Summary of Research Contributions

This chapter presents a comparative study of three different error models: bit-flips (**BF**), data-type (**DT**) and fuzzing (**FZ**). The models are compared using data from a representative case study conducted on Windows CE .Net. Furthermore, the robustness measures introduced in Chapter 5 are used to compare the three models on their abilities to trigger error propagation in the system. Overall the following key observations are made, which can be used as input and recommendations for future robustness evaluations:

- The measures derived in Chapter 5 are shown to be useful for studying failure and propagation characteristics of the OS, identifying services and drivers with potential robustness vulnerabilities.

- The **BF** model finds more vulnerabilities than the other models. It also identifies more services in the OS-Driver interface having vulnerabilities making it the preferred choice for robustness evaluations.

- All three models are well suited to study error propagation characteristics using especially the Driver Error Diffusion measures. Some differences across the models are observed, relating to the use of control values in the interfaces.

- The **DT** model uses the fewest injections, followed by the **FZ** model and the **BF** model. The use of profiling can reduce the number of injections for **BF** and a careful study of the number of injections for **FZ** shows that one can perform experiments with relatively few injections.

- In terms of implementation costs the **DT** model is the most costly. **BF** and **FZ** are comparable but the time required for any implementation depends on many factors, including skills, experiences and availability of tools and documentation.

- For identifying service vulnerabilities the **BF** model is the preferred choice. However, the random nature of **FZ** allows for finding other vulnerabilities that the other two models do not find.
- A new *composite error model* is defined as a composition of bit-flips and fuzzing injections. A subset of the available bits for a parameter are targeted after a profiling step revealing which bits have a higher impact on the system.

Chapter 7

Error Timing Models - When to Inject

When - in the time domain - should errors be injected?

Services used in driver/OS interactions are typically invoked multiple times during the lifetime of the driver. Therefore, when injecting errors in services which are called multiple times the time at which an error is injected will obviously affect the outcome of the experiment. Consequently, controlling the time at which the error is injected is a crucial part of the robustness evaluation. Multiple tools have been developed which allow for control of the time of injection. Most tools allow injection based on user-defined events or according to some time distribution. However, surprisingly little research has been spent on strategies for selecting injection times, beyond time distributions.

This chapter is devoted to a novel timing model used for robustness evaluation of OS's to errors in device drivers. It helps answering **RQ5** - when to inject - by defining a usage profile of a driver, which can be used to control and select the time at which the errors are injected. Extensive experimentation shows that controlling the time of injection is indeed important, and furthermore that its effectiveness depends on the usage profile of the driver.

7.1 Introduction

When discussing timing issues for fault injection two aspects of the errors injected are relevant: the time at which it is injected and the duration it stays active. This chapter focuses on the former property of an error.

For the duration of injected errors we focus on software related errors. The system is assumed to function properly when no errors are injected. The duration of injected errors is transient. Transient errors appear and then disappear shortly thereafter. This model reflects Heisenbugs, i.e., those software faults which due to external conditions do not deterministically reoccur every time the system is used. Such faults are hard to find with traditional testing techniques and may therefore occur even in well tested systems. Injections are performed in the OS-Driver interface, thus limiting the potential injection instances to when services in this interface are used. The transient model translates into the error being injected once for the targeted service and then disappearing before the second call to the same service.

Two main generic strategies exist to trigger the injection of an error: *event-triggered* and *time-triggered*. In the former approach specific events are used to trigger the injection, and in the latter approach time is used to trigger injection. Event-driven injection typically allows for a more fine-grained control of the individual injections, but requires the triggering events to be defined. Time-triggered injection relies on a larger number of injections, distributed over time, and consequently requires more injections.

This chapter presents an approach extending the event-triggered approach presented in Chapter 6, where errors are injected in the first call to a service, a so called *first-occurrence* approach. First-occurrence only targets the first call to a service, disregarding any subsequent calls. The usage profile of the driver is used to build a usage model of the driver, and the service calls to be targeted can be selected to cover a wider spectrum of system states.

The rest of the chapter will be structured as follows: First a discussion on the two alternative timing models provides the foundation and background needed for the rest of the chapter. The driver usage profile model is presented and discussed, followed by a description of the evaluation criteria used for the experimental evaluation of the approach. The description of the the implementation and the results are then presented and discussed. The conclusions made and a summary of the research contributions follows in the last section.

7.2 Timing Models

The time at which an error is injected is also referred to as the triggering mechanism for the error. The need for controlling and monitoring the triggering event has previously been identified as important, but inherently difficult [Whittaker, 2003]. Several of the fault injection tools surveyed in Section 2.3 allow for controlling the triggering of errors, at least to some extent. On an abstract level an error is always triggered by an *event*.

For practical purposes one makes a distinction between events triggered by special events taking place in the system, and those triggered by time alone, giving rise to the two classes of timing models for fault injection, event-triggered and time-triggered.

7.2.1 Event-Trigger

In the event-triggered case, the most common approach for software is *location-based* injection. This strategy is based on the premise that since the system's vulnerable states cannot generally be postulated a-priori, the events triggering injection are based on reaching certain locations in the code of a module. The simplest of such strategies is to inject the first time a certain location is reached (*first-occurrence* strategy).

A location-based approach is relevant especially for code injection, where errors (or faults) are injected directly into the source code (or executable binary) to mimic software faults [Durães and Madeira, 2002; Ng and Chen, 2001], or into the instruction stream of the CPU [Gu et al., 2003]. This comes from the fact that the software faults mimicked have specific locations in the code.

A variation of the first-occurrence approach is to use an n-occurrence trigger, i.e., the error is injected after n calls to a service, or after reaching a location for the n^{th} time. This approach is a generalization of the first-occurrence approach, but requires the user to set the value of n, which is far from trivial. The approach is implemented for instance in the FERRARI tool [Kanawati et al., 1995].

In [Tsai et al., 1999] CPU registers and memory are targeted for fault injection using bit-flips. To maximize the activation rate of faults, and their impact, the timing of the injections are controlled by the workload in the system. Offline analysis detects paths through the workload for a given set of inputs and faults are injected along the paths. Alternatively, the activation rate of the CPU is used to perform injections at peak-usage times.

The advantage of the event-triggered approach is that it can be tailored to evaluate specific locations in a system, or as specific events take place.

It can therefore speed up the evaluation process by reducing the number of injected errors and focusing on only the relevant events in the system. The disadvantage is that often these events need to be defined by the user. Selecting them is a difficult process, possibly requiring deep understanding of the system, its components and their interaction.

7.2.2 Time-Trigger

When using a time-triggered approach a timeout is defined, after which the error is injected. Typically a large number of injections are performed, and their injection times follow some specific distribution (e.g., uniform, normal, exponential etc) [Kao and Iyer, 1994; Han et al., 1995; Rodriguez et al., 2002]. In this case, often the location is also randomly selected across a set of predefined locations. This approach is common when simulating physical faults (radiation, EMI etc.) which are inherently "random" in nature [Karlsson et al., 1994].

Alternatively, the triggering event is defined as a combination of time and location, such that after the timeout has elapsed the error is injected at a predefined location, possibly using first-occurrence, or after the n^{th} occurrence of a call. FERRARI, for instance, allows for specifying a time distribution (such as uniform) after which the fault is injected [Kanawati et al., 1992, 1995].

Many fault injection tools support both event and time-triggered injection [Carreira et al., 1998; Kanawati et al., 1995], but still leaves the burden of choosing events and/or distributions to the user.

The time-triggered strategy does away with the burden of selecting triggering events, but on the other hand instead relies on a large number of injections to get statistically significant results. In any case a distribution needs to be selected and justified.

7.3 Driver Usage Profile

As seen from Section 7.2 both models have advantages and disadvantages. Since effectiveness is one of the key goals of robustness evaluation and with our focus on software faults we have opted for an event-driven approach. The event-driver approach is more suitable for driver errors as it is easier to control and is more fine-grained. It is also more suitable for software faults.

As many services are called multiple times during the execution of the test applications the question arises which of these calls to target. Targeting each call will generally not be possible due to the large number of calls. Thus

a selection of a subset is required. The simplest, and most straight-forward approach is to use the first-occurrence approach. This is indeed the approach used in the previous chapter and in several other projects. However, when injecting in interfaces between components (such as drivers), the injected error may stem from several distinct locations in the component, i.e., may have several distinct fault origins. Thus, injection on first-occurrence will *only target a subset* of those potential fault locations, namely those corresponding to the first call. Subsequent calls to a service will not be targeted, even though the driver (and indeed the whole system) may be in a different state which may be of interest to evaluate.

We have developed a methodology to select the relevant service invocation based on the observation that a service request from an application translates into one, or more calls into the driver by the OS. In any practical context only a small subset of the possible sequences of calls that can be made are actually observed. As an example, it does not make sense for an application to read from a file before it has been opened, although there is nothing stopping it from trying. Since we base our evaluation on a system which is functional, it can be expected that such behavior is not present when the workload used is executed. Thus, the operational behavior of the driver can be broken down into a series of calls to the driver, where certain sub-sequences are more frequent than others. Such subsequences represent common sequences of calls made from applications, thereby defining the "operations" performed by the driver, such as "creating a file", or "setting connection parameters" etc. Such an elementary sequence of calls is called a *call block*. In our model, the call blocks are used to trigger the injection of errors in the OS-Driver interface. Thereby giving a more fine-grained control over the time of injection, compared to first-occurrence and time-triggered injection.

The driver usage profile is defined as an ordered list of calls to services provided by the driver (the $ds_{x.y}$ services in Figure 3.1). This definition is slightly different from the traditional definition of an *operational profile* as defined for instance by Musa [Musa, 1993]. The operational profile is typically defined as the frequency distribution across a component's functions. Our usage profile additionally considers the order in which the functions are called. The list of calls made is termed the *call string* of the driver. The call string is further divided into a set of *call blocks*, which are disjoint subsequences of the call string.

7.3.1 Call String

Figure 7.1 illustrates the driver usage profile as a time-service diagram. The calls made to drivers services are illustrated as rectangles in the figure ($a-d$).

The call string is formed by assigning tokens to each service from a predefined alphabet and then for each call adding one token to the list. The call string for the example in Figure7.1 is thus *ababcdabdab*. As can be seen, some sequences of calls are repeating, forming call blocks as subsequences of the call string (α, β and γ). Note that sequential execution of the driver is assumed.

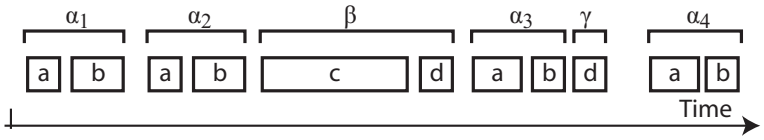

Figure 7.1: Example of called services.

7.3.2 Call Blocks

The example in Figure 7.1 shows that services a and b are called multiple times during the execution. Each call to service a is followed by a call to b; a and b thus form a *call block*, which is repeated during the execution of the driver. The sequences cd and d are not repeating, and cannot be added to any other call block. These sequences form the non-repeating call blocks. The call string is thus split into call blocks as indicated in Figure 7.1. Using a conventional regular expression syntax the the sequence can be represented compactly as **(ab){2}cd(ab)(d)(ab)**, where **(ab){2}** means that the sequence ab is repeated twice.

Currently the assignment of call blocks is performed through a combination of identification of repeating blocks and a priori knowledge regarding the functionality of the driver. As call string grow in size automated techniques will be required to handle the large number of tokens. Section 7.6 discusses the use of special data structures to automate call block identification.

In figure 7.2 a similar call block structure is illustrated, and additionally shows the services called within each call block. When targeting service s_i using the first-occurrence approach injections are performed only the first call to that service. Subsequent calls to s_i, for instance in α_2 or α_3, are not targeted. Using the call block strategy multiple calls to s_i can be targeted.

7.3.3 Operational Phases

In general, a driver's lifetime can be split into three disjoint phases: initialization, working and clean up phases, as seen in Figure 7.3. In the initialization phase the driver sets up required data structures and registers its presence with the OS. Thereafter follows the working phase, where the driver performs

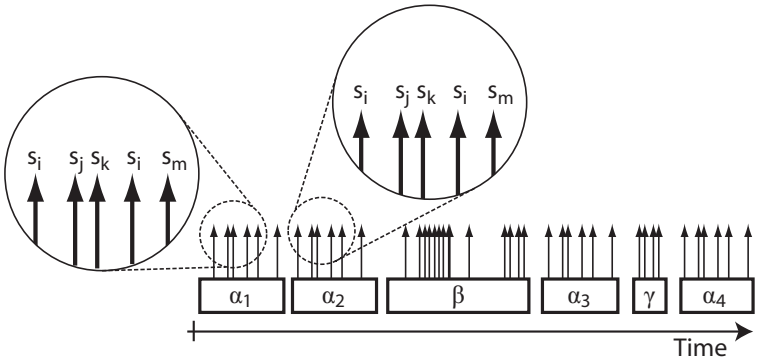

Figure 7.2: Example of a driver calling OS services in different call blocks.

work on behalf of applications or the OS itself. Finally, the clean up phase unregisters the driver with the OS and releases any resources held.

The operational phases become relevant when discussing selection of call blocks for injection. Intuitively it can be expected that failures in the initialization and clean up phases are more severe than in the working phase, as in these phases the driver interacts with many OS services which may affect the state not only of the driver but the whole system. The working phase on the other hand, is where drivers spend the most time (may therefore have been more extensively tested) and perturbations may be expected and therefore considered by developers.

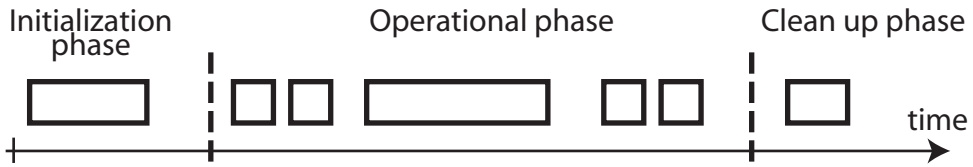

Figure 7.3: The operational phases of a driver.

In this work we are mostly focusing on the driver's operational phase. However, with application level knowledge, similar phases can be defined also for the workload used. Looking at the workload used for the case study (Section 4.5.4), the driver specific test applications can be decomposed into two rounds of initialization, working and clean up phases, as illustrated in Figure 7.4. Note that this is specific to these test applications and requires access to and knowledge of the applications.

Each call block is targeted for injection, i.e., each operation performed by the driver. For call blocks that are repeated multiple times a filtering may take place, to reduce the number of injections. Preferably at least one

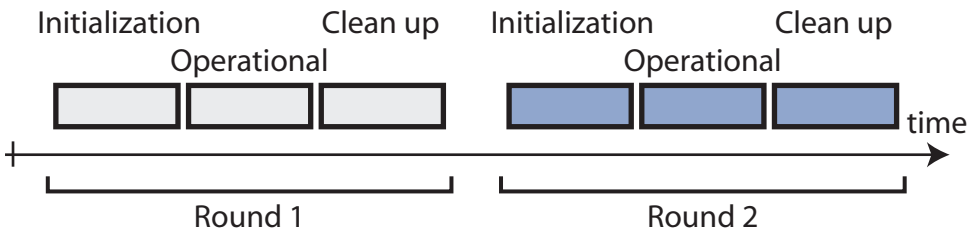

Figure 7.4: The operational phases of the workload.

call block per operational phase should in this case be targeted. For each call block, the first occurrence approach can be used on each service called in that call block. Note that typically not all services are called in each call block, giving rise to some call blocks requiring many injections and some few.

7.4 Experimental Setup

To evaluate the usefulness of the proposed approach an implementation has been made for the Windows CE .Net. The serial port driver and the Ethernet driver were selected and call blocks were derived for both drivers.

The proposed approach will be compared to a traditional first-occurrence approach. The main criteria used for the comparison will be the number of injections required, the failure class distribution observed and the number of severe vulnerabilities observed for each of the two approaches.

This section first presents the two drivers, the injection strategy and details the selected call blocks identified for each of the drivers.

7.4.1 Targeted Drivers

Two drivers are selected for this case study, the serial port driver (cerfio_serial) and Ethernet driver (91C111). The two drivers are well suitable for evaluation as they a) represent functionality found in all modern OS's, and b) represent different functionalities, giving rise to a different usage profile of the OS.

The difference in OS usage profile can be illustrated by studying the frequencies at which OS services are called by the drivers for some workload, in our case the test applications described in Section 4.5.4. Figure 7.5 and 7.6 show the difference in profile for the two drivers. The x-axes show the services called by the two drivers and the y-axes show the number of calls

made to each service. The two figures clearly show that the cerfio_serial calls a higher number of services and more frequently than 91C111.

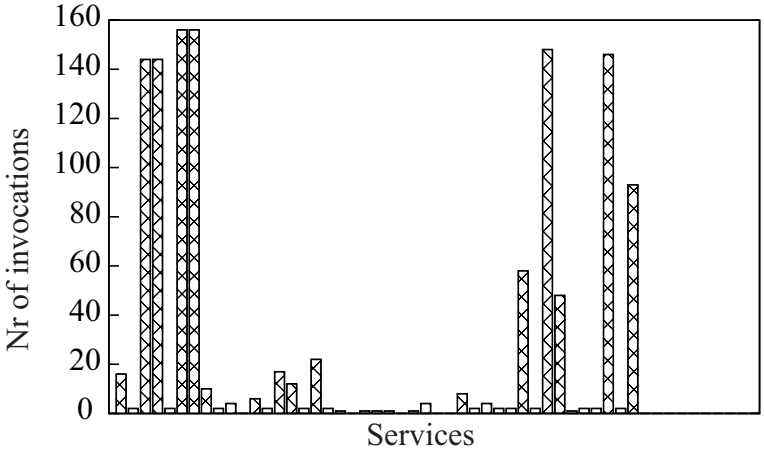

Figure 7.5: Call profile for cerfio_serial

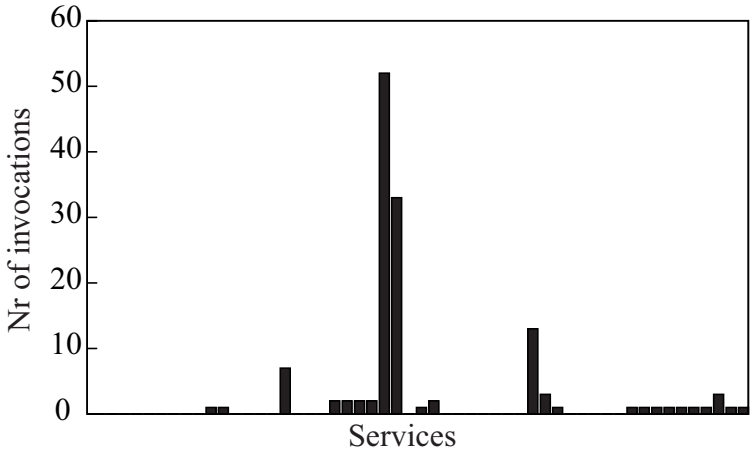

Figure 7.6: Call profile of 91C111

cerfio_serial uses 41 services. On average a service is invoked 30.5 times for the given workload with a standard deviation of 53.5 and median of 2. This reflects the fact that there are some services that are used frequently (for reading/writing, synchronization etc.) and some only once or twice (like configuration of the device). For 91C111 the average number of invocations is 5.4 with a median of 1 and standard deviation of 11.7.

Both figures show that many services are called multiple times, indicating that first-occurrence may not find all vulnerabilities. The difference between

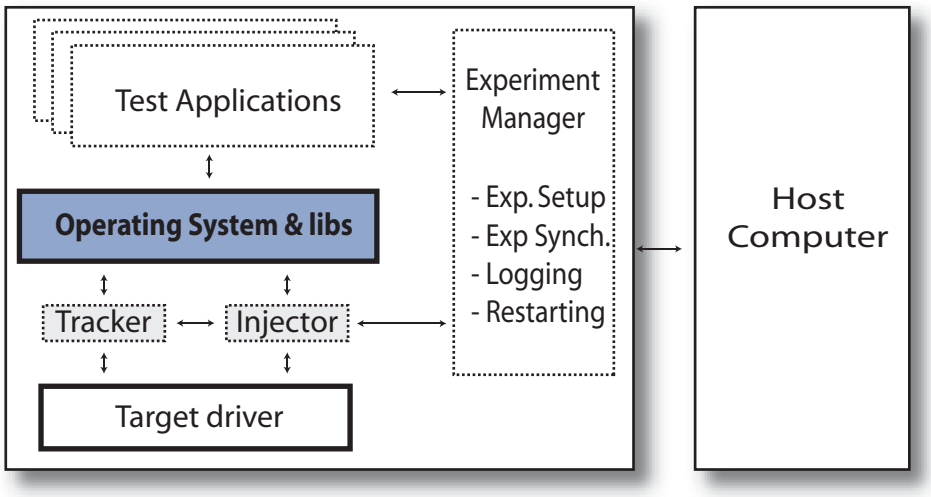

Figure 7.7: The experimental setup.

the two drivers also suggests that as more services are called frequently for cerfio_serial it is to be expected that the call block approach will be more effective for this driver.

7.4.2 Error Model

Building on the results from Chapter 6 the bit-flip (**BF**) model was chosen to evaluate the two approaches. The model was chosen as it is the most vulnerability-revealing model of the ones evaluated in Chapter 6 and will therefore better explore the potential of both approaches. Focus will be put on the most severe class of failures and experiments are therefore focused on the import interface of the drivers, which was the only one to experience **Class 3** failures in Chapter 6.

7.4.3 Injection

The experimental setup used for the injection differs slightly from the one used in the previous chapter in that a new Interceptor module (*tracker*) has been introduced between the OS and the targeted driver. The experimental setup is shown in Figure 7.7. For the call block approach the injector is configured to inject errors when the tracker signals that the targeted call block is reached. Apart from these changes the setup remains the same as previously described.

Before injections can be made, a profiling execution of the driver is performed. During this error-free execution the tracker module records all calls being made to the driver, i.e., it records the call string for the driver. For the two drivers and the workload used for these experiments the call strings were both deterministic, i.e., every time the workload was executed without injecting errors the same call string was generated. This is a simplification and a result of choosing a simple and deterministic workload. Using a deterministic workload reduced the triggering of injections in a specific call block to counting calls made to OS services. Section 7.6 further discusses this issue.

7.4.4 Call Strings and Call Blocks

This section reports on the call strings and call blocks identified for the two targeted drivers.

Serial Port Driver

The workload for cerfio_serial first writes a string of characters to the serial port which are read by the host computer connected to it. The host computer echoes the same string back and they are read one by one by the application. This process is then repeated once more. The workload generates calls to the driver, forming the call string shown in Figure 7.8, represented as a regular expression. In total the call string for the serial port driver contains 152 tokens.

The first call performed is an initialization call to the driver, `DllMain`. Such an entry point exists for each driver. After this follows a series of calls to perform the services requested.

In Figure 7.8 the tokens assigned are listed in Table 7.1, which shows the Stream interface entry points provided by cerfio_serial. Additionally, the **DllMain** function called when the Dll is loaded is assigned the token **D**. More detailed information on the Stream interface and device drivers for Windows CE .Net can be found in [Boling, 2003].

The calls to **DllMain** and to **COM_Init** make up the initialization phase of the driver. The working phase (which of course is workload dependent) consists of a series of calls to **COM_Open**, **COM_Read**, **COM_Write**, **COM_Close** and **COM_IOControl**. The the clean up phase of the workload finishes with the call to **COM_Close**. The pattern is then repeated once more.

The call string in Figure 7.8 is split into call blocks, as illustrated in Table 7.2 and Figure 7.9. Five call blocks are identified ($\delta, \alpha, \beta, \gamma$ and ω), some of which are repeating. For the remainder of the presentation in this chapter

Table 7.1: Stream interface for serial driver.

Number	Name	Purpose
0	COM_Init	Initializing the driver.
1	COM_Deinit	Depricated
2	COM_Open	Open a connection to the device or a file.
3	COM_Close	Close a previously opened connection or file.
4	COM_Read	Read from an open connection or file.
5	COM_Write	Write to an open connection file.
6	COM_Seek	Move within the file. Usually do not work on connection-oriented devices.
7	COM_IOControl	Send control commands to the device. Are typically device specific.
8	COM_PowerDown	Tell the device to move to a power saving state.
9	COM_PowerUp	Tell device to come back from power saving states.

D02775(747){23}732775(747){23}73

Figure 7.8: The serial driver call string.

we term the targeted call blocks $\delta, \alpha, \beta_1, \gamma_1, \omega_1, \beta_2, \gamma_2$ and ω_2 as shown along the x-axis in Figure 7.9. Note that the x-axis indicates time as sequences of call blocks, not physical time, i.e., the length of a box does not represent the execution time of the service.

Table 7.2: Call blocks for cerfio_serial.

Call block	Tokens	Occurrences
δ	pre-load	1
α	0	1
β	2775	2
γ	747	46
ω	73	2

For the call blocks that reoccur once (β and ω) we target each instance of the call block. Call block γ repeats all in all 46 times, and targeting each of them would clearly be very time consuming. Therefore, we target one instance of γ in each repeating sequence. For the first sequence we target

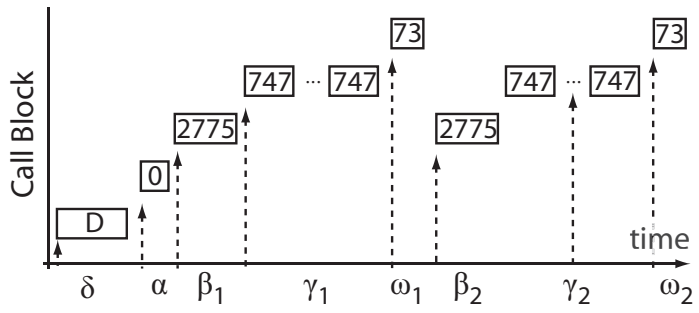

Figure 7.9: The call blocks for the serial port.

the first instance and for the second we have chosen one instance arbitrarily (the sixth).

Ethernet Driver

The network card driver workload works in a similar fashion as cerfio_serial workload. A message is sent over the network and is echoed back by the host computer. However, as the NDIS wrapper is used instead for the Stream interface for the Ethernet driver a slightly different tracker mechanism is used (the driver exports Stream functions as well though as part of being a proper driver).

Table 7.3: NDIS callback functions for the passthrough wrapper.

Number	Name
1	MiniportInitialize
3	MiniportSend
3	MiniportSendPackets
4	MiniportQueryInformation
5	MiniportProcessSetPowerOid
6	MiniportSetInformation
7	MiniportReturnPacket
8	MiniportTransferData
9	MiniportHalt
10	MiniportReset
11	MiniportCancelSendPackets
12	MiniportDevicePnPEvent
13	MiniportAdapterShutdown

The NDIS architecture is a layered one, with the upper layer being proto-

col layers (TCP/IP etc) and the lower layer being the device driver (termed miniport driver). The layered model, with its defined interfaces, makes it possible to introduce new filtering layers in-between existing layers (which is what some firewalls and anti-virus software do). A passthrough wrapper is implemented, that is added on top of the miniport driver targeted (91C111.dll). The passthrough wrapper, as the name suggests, does not alter the data in any way, simply passes it through to the miniport driver. The purpose is only to track the calls made to the driver. The callback functions exported are summarized in Table 7.3. Less than ten functions were exercised for our workload, which allowed us to still use single numerical tokens for the call string. Note that this is no real limitation to the approach, since any alphabet could have been used.

D1(4){9}1(4){15}666(4){10}666444336663444(3){9}

Figure 7.10: The Ethernet driver call string.

The call string for the Ethernet driver is shown in Figure 7.10. Again, the first call is to **DllMain**. Additionally the driver has a specific setup export (**DriverEntry**) which together forms the first entry in the call string (**D**).

Table 7.4: Call blocks for the Ethernet driver.

Call block	Tokens	Occurrences
δ	DllMain + DriverEntry	1
α	1 4444 44444	2
β	444 444	1
γ	666 444	2
μ	444 4444	1
ω	33 666 63644 333333333	1

As for cerfio_serial a manual inspection gave rise to the call blocks illustrated in Figure 7.11. As the Ethernet driver gave rise to significantly fewer call blocks we target all call blocks using any services. For some call blocks no OS services were used, and consequently no injections were performed. Similarly, the last call block (ω) gives rise to no calls made to the OS. Therefore, it was not further split into call blocks.

7.5 Result of Evaluation

Fault injection experiments were carried out for both drivers. Injections were performed using both the first-occurrence approach and the proposed call-

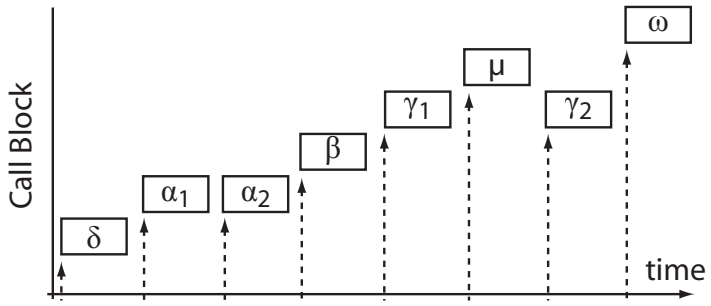

Figure 7.11: The network driver call blocks.

block approach. An overview of the results is shown in Table 7.5 and a more detailed view is shown in Table 7.6. Note the use of the the previously introduced names for identifying the call blocks. The results are graphically illustrated in Figures 7.12 and 7.13. A discussion and further interpretation of these results is found in the Section 7.6.

Table 7.5: Comparing the first-occurrence and call block approaches on the number of injections and the number of experiment outcomes in the most severe failure class (**Class 3**).

Trigger	Serial driver		Network driver	
	#Injections	#Class 3	#Injections	#Class 3
First occurrence	2428	**10**	1818	12
Call blocks	8408	**13**	2356	12

7.5.1 Serial Port Driver

The distribution of outcomes across failure classes is shown in Figure 7.12. For comparison purposes the outcome of the first-occurrence injections are shown as well. Additionally the number of injections for each call block is shown as opaque, black bars. For illustrative purposes the **Class NF** is not shown, but all data is found in Table 7.6.

Call blocks γ_1 and γ_2 exhibit the lowest number of **Class 3** failures. These call blocks correspond to the working phase of the driver, were it is only sending and receiving data. Since the work done in this phase corresponds to the main part of a driver's lifetime, it is reasonable to expect it to be sufficiently well specified and understood for the OS to be implemented tolerating many

fluctuation in device/driver behavior. A small difference in **Class 2** behavior can also be observed, with γ_2 having a slightly higher ratio.

Compared to the working phase, the initialization phase of the driver shows a higher ratio of **Class 3** failures (δ and α). Similarly, the initialization phase of the test application (β) shows a high ratio of **Class 3** failures. The same holds for the clean up phase of the test application (ω). Whereas ω_1 and ω_2 show close to identical distributions, β_2 shows more Class 1 failures than β_1.

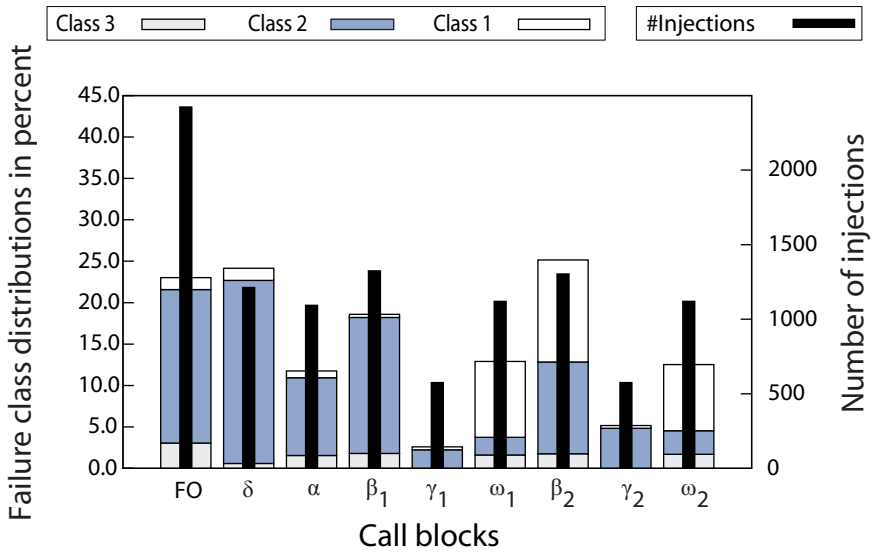

Figure 7.12: Failure class distribution and number of injections for the first-occurrence approach and for each call block of cerfio_serial.

From Figure 7.12 it can also be seen that the first initialization call block δ is less prone to **Class 3** failures than the second call block in the initialization phase, α. When `DllMain` is called (i.e., in δ) driver developers are discouraged (in the documentation) to include any time consuming or complex operations, restricting to only initialization of synchronization objects and other lightweight operations. This minimizes calls made to the OS in this critical phase of the system. Consequently we observe fewer **Class 3** failure. However, as the these operations may now fail, with the whole driver being unable to make progress, we see a rise in **Class 2** failures instead compared to α.

Comparing with the first-occurrence injections it can clearly be seen that the first-occurrence approach gives very similar results as the call blocks in

the initialization phase. This behavior is of course expected, since it injects in the first call to each service used by the driver.

Not only the distribution across failure classes is of interest, but also the number services found to have severe (**Class 3**) failures. Table 7.7 presents the result for cerfio_serial. From the 41 services being targeted, 13 services caused **Class 3** failures. Compared with the first-occurrence approach this is an increase of three services, i.e., three services not previously found to have **Class 3** failures were identified. This indicates that *choosing the triggering event for injection for the serial port driver has a significant impact on the results obtained* and that *first-occurrence is not sufficient for a comprehensive evaluation.*

Table 7.6: Injection results for the serial (cerfio_serial.dll) and the network card driver (91C111.dll).

Driver Name	Call Blocks	#Injections	NF		C1		C2		C3	
			#	%	#	%	#	%	#	%
cerfio_serial.dll	FO	2428	1889	76.98%	35	1.44%	450	18.53%	74	3.05%
	δ	1217	923	75.84%	18	1.48%	269	22.10%	7	0.58%
	α	1098	969	88.25%	9	0.82%	103	9.38%	17	1.55%
	β_1	1356	1082	79.79%	5	0.37%	218	16.08%	51	3.76%
	γ_1	580	565	97.41%	2	0.34%	13	2.24%	0	0.00%
	ω_1	1124	979	87.10%	103	9.16%	24	2.14%	18	1.60%
	β_2	1328	979	73.72%	161	12.12%	145	10.92%	43	3.24%
	γ_2	580	550	94.83%	2	0.34%	28	4.83%	0	0.00%
	ω_2	1125	984	87.47%	90	8.00%	32	2.84%	19	1.69%
91C111.dll	FO	1818	1288	70.85%	416	22.88%	50	2.75%	64	3.52%
	δ	1756	1159	66.00%	494	28.13%	32	1.82%	71	4.04%
	α_1	96	96	100.00%	0	0.00%	0	0.00%	0	0.00%
	α_2	171	101	59.06%	65	38.01%	1	0.58%	4	2.34%
	β	171	99	57.89%	65	38.01%	4	2.34%	3	1.75%
	γ_1	171	102	59.65%	64	37.43%	2	1.17%	3	1.75%
	μ	0	-	-	-	-	-	-	-	-
	γ_2	0	-	-	-	-	-	-	-	-
	ω	0	-	-	-	-	-	-	-	-

Table 7.7: The services having Class 3 failures for cerfio_serial. Services not identified by first-occurrence are marked with **X**.

Service/Call block	FO	δ	α	β_1	γ_1	ω_1	β_2	γ_2	ω_2
CreateThread	x			x			x		
DisableThreadLibraryCalls	x	x							
EventModify						**X**			**X**
FreeLibrary	x	x							
HalTranslateBusAddress	x		x						
InitializeCriticalSection		**X**							
InterlockedDecrement									**X**
LoadLibraryW	x	x							
LocalAlloc	x	x							
memcpy	x			x			x		
memset	x			x			x		
MmMapIoSpace	x								
SetProcPermissions	x			x			x		
TransBusAddrToStatic			**X**						

7.5.2 Ethernet driver

Table 7.6 and Figure 7.13 show that call block δ shows a very similar distribution as the first-occurrence approach. For call blocks μ, γ_2 and ω the driver does not perform any calls to the OS, and consequently no injections were performed and these call blocks are not shown in Figure 7.13.

The call blocks α_2, β and γ_1 show a very similar behavior, due to the fact that only one services is used for all three call blocks. Therefore the same amount of injections are performed. No injections in call block α_1 show any **Class 3** failures. Overall no new services were found to have **Class 3** failures (see Table 7.5).

7.6 Discussion

This section interprets and discusses the results of the case study. Focus is mostly on the most severe class of failures, **Class 3**.

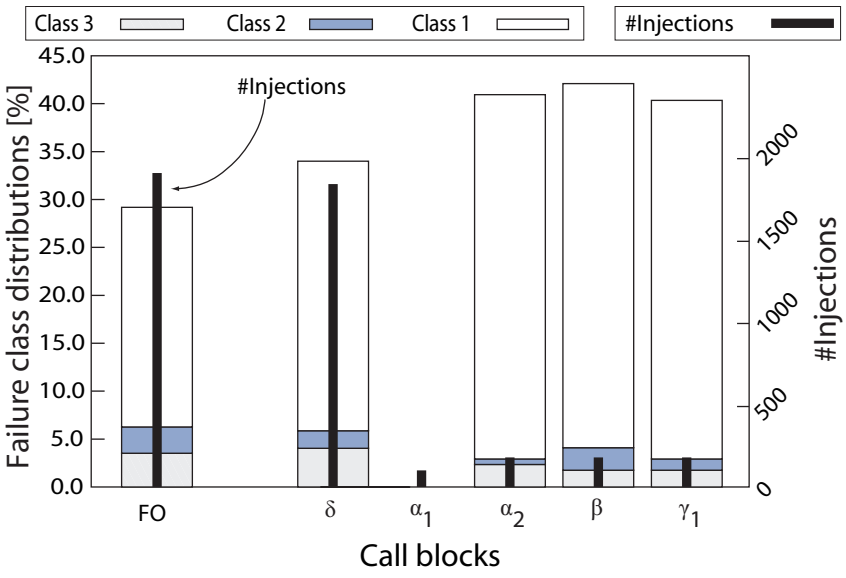

Figure 7.13: Failure class distribution and number of injections for each call block of 91C111. Call blocks without injections are excluded.

7.6.1 Difference in Driver Types

The results for the two drivers show a significant difference across them. For the serial driver a good number of new services were found to experience **Class 3** failures. The Ethernet driver on the other hand did not get any additional service vulnerabilities with the new call block approach.

Section 7.4.1 shows that the serial driver uses more services, and more frequently than the Ethernet driver, suggesting that it would be more susceptible to call block injections, and this is indeed also the case. For the Ethernet driver no new **Class 3** failures where identified and the δ call block shows a very similar behavior to the first-occurrence injections, further substantiating the intuition.

Further analysis on the calls made for each call block is presented in Figures 7.14 and 7.15. Figure 7.14 shows the call profile for cerfio_serial, i.e., the number of calls made by the driver for each of the call blocks identified. It can clearly be observed that the calls made are spread throughout the lifetime of the driver, with call block β having the most calls.

Figure 7.15 shows the call profile for the Ethernet driver. It shows that the Ethernet driver is more active in the initialization phase of the driver (to the left in the figures) than in the working phase. Compared with cerfio_serial in Figure 7.14 the distribution of calls is clearly geared towards the initialization

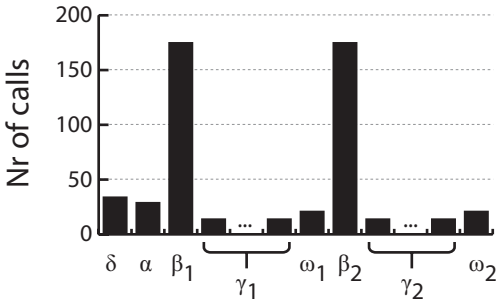

Figure 7.14: The call profile for the serial driver.

phase.

This explains why the call block approach finds more new vulnerabilities for the serial drivers than for the Ethernet driver. It confirms that *for drivers which perform few calls, and especially during the initialization phase, first-occurrence is to be preferred.*

The profiling of the drivers show that the effectiveness of the approach can to some degree be predicted and suggests that *a profiling of the targeted drivers should be conducted before triggering techniques are selected*, to minimize the time for implementation and number of injections required. Note that such profiling can be conducted *prior* to injection any errors! However, profiles, such as Figures 7.14 and 7.15 does require call blocks to be defined.

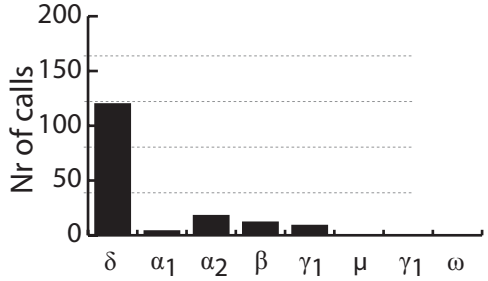

Figure 7.15: The call profile for the Ethernet driver.

7.6.2 Comparing with First Occurrence

The first-occurrence approach has several distinct advantages compared to the call block approach. First and foremost it uses fewer injections. It is also appropriate when doing code level injections (Table 7.5), where the location of the modeled fault coincides with the injected error.

The higher number of injections for the call block approach translates into longer time required for performing the evaluation. This can be somewhat lessened by performing pre-profiling to remove non-significant experiments. Unfortunately, the higher number of injections is inherently inevitable since the call block approach injects in multiple invocations of a service, whereas first-occurrence only injects in the first call to a service.

The additional costs give rise to a trade-off with the usefulness of the results. New service vulnerabilities can indeed be identified using the call block approach, as shown in Section 7.5. First-occurrence found ten **Class 3** services, whereas the call block injections found thirteen. This represents an increase of 30%. On the other hand, for the Ethernet driver no additional services are identified due to its call profile being geared towards the initial phase.

7.6.3 Identifying Call Blocks

The length of a call string varies depending on the workload used to generate it. Manual inspection was sufficient to identify call blocks for the experiments presented in this chapter. However, this will not be feasible for longer call strings. For longer call strings some level of automation is required.

The repeating nature of call blocks is what makes them identifiable in the call string. To identify a repeating sequence of tokens from a given alphabet in a string is a well studied problem. Examples of uses is to identify repeating sequences in DNA. Multiple data structures and algorithms have been devised for this purpose. Many examples of such problems can be found for instance in [Gusfield, 1997]. We have explored the use of *suffix trees*, a special tree data structure, which finds repeating sequences quickly. However, further research is needed to develop appropriate tools and techniques to fully take advantage of this data structure.

Figure 7.12 and 7.13 show that some call blocks are more useful in identifying **Class 3** failures. These call blocks typically belong to either the initialization or clean up phases of the driver and workload. By focusing mostly on the initialization (like first-occurrence) and clean up phases one could potentially reduce the number of injections required.

7.6.4 Workload

With operational phases and call blocks forming the basis for the usage profile it is important to identify representative workloads to be used to drive experiments. Representative of the system's expected workload once it becomes operational. In many cases no, or only partial information is available

regarding the expected operational profile. If not known, a common approach is to use a synthetic workload, exercising the system in a diverse way [Johansson, 2001]. The alternative to using a synthetic workload is to use real-world applications. However, many applications are not suitable to be used directly, due to required user inputs, non-determinism etc.

As call blocks represent higher level operations carried out on the driver it is important that the call string is stable across runs, i.e., that the same call string is generated every time the workload is executed. The workload used in this thesis is indeed stable and deterministic, and gives rise to the same call string for each run. This is an important property for any workload used in a comparative purpose and is a well established approach in the benchmarking community, like the standard tests in SPEC [SPE]. A deterministic workload is key for achieving reproducibility, an important property identified for dependability benchmarks [Johansson, 2001; Kanoun et al., 2005]. As one still strives after using real-world workloads, an approach is to modify applications by removing sources of non-determinism, such as user inputs, by using specific user scenarios or use cases.

Another source of non-determinism for device drivers is the fact that they can, in general, be accessed by several applications concurrently. Depending on the semantics of the device (or driver) this may be supported or not. A serial driver does for instance not generally accept concurrent accesses, whereas a network card driver typically does. For the experiments in this thesis we have deliberately focused on single-access application scenarios. This is a simplification and extending the approach to handle concurrency is a topic for future research. Concurrent accesses would give rise to interleaved tokens in the call string, belonging to different processes/tasks. Furthermore, the implementation of triggers will be more complex than the current one (based on simple counters), since online pattern recognition is required. Whether concurrent access should at all be considered for comparative (benchmarking) purposes can be discussed, as it gives rise to potential issues with reproducibility.

7.6.5 Error Duration

As previously described we have used a transient duration model for the injected errors, i.e., each error appears once and then disappears for subsequent invocations of the same services. Our injection framework does support intermittent (error disappears after n invocations) as well as permanent errors. However, none of these models have been evaluated yet.

7.6.6 Timing Errors

For this work we have considered the time of injection. This is not to be confused with timing errors, which change the timing behavior of the system by for instance delaying or dropping calls being issued. Since many devices require specific timing requirements to be met, this may be a relevant error model for evaluating device drivers and OS's. Experimental support for this error model is implemented in the fault injection framework developed, but no systematic evaluation has yet been carried out using this model.

7.7 Related Work

The call block strategy is one of several that uses a profile of the system to trigger injection. In [Tsai et al., 1999] stress-based injections are performed, where injection is synchronized with high workload activity in the system. Similarly application resource usage is analyzed to guide injection into resources actively used.

Tsai and Singh [2000] used a setup very similar to ours, but with the intent to test applications on Windows NT by corruption of parameter values to library calls. The first-occurrence strategy is used, and a comment is made that regarding injection in subsequent calls: "...preliminary results showed that such injections produced similar results." [Tsai and Singh, 2000, page 4]. We believe that this assertion does not generally hold. However, the results for the network driver shows that depending on the call profile of the targeted component, different behaviors are observed. This suggests that more research is needed to completely characterize for which components first-occurrence is most suitable and for which not.

In this work we do not consider distributed systems explicitly. For distributed systems the concept of *global state* is of key importance and one may want to inject errors at particular local and/or global states. Whereas the technique presented here is suitable for local states, it does not handle global states of distributed systems, where even detection of specific global events/states is difficult. Some work has been done within this specific area, for instance the Loki tool [Chandra et al., 2004].

7.8 Summary of Research Contributions

The time at which a fault is injected can have an impact on the robustness evaluation of a system. In the context of device drivers for OS's, this chapter has established that selection of the triggering events (controlling error

timing) does impact the evaluation results. Furthermore, it is shown that a profiling of the driver reveals its sensitivity to the timing of injections.

The following distinct contributions are put forward in this chapter:

- A novel timing model for device drivers is presented. The new model is based on the concept of a call block, a subsequence of calls to the driver corresponding to higher level operations.
- It is detailed how the proposed model can be used to identify injection triggers for device drivers used in fault injection.
- A large case study for two device drivers show that selecting the error timing impacts the robustness evaluation, especially for drivers which actively use OS services throughout their lifetime.

Chapter 8

Conclusion & Future Research

What have we learned, and how do we move forward?

This chapter concludes the thesis by summarizing its main contributions and discussing their relevance to the research community.

Additionally, this chapter aims to broaden the scope of the techniques used by surveying and discussing their usability in enhancing the dependability of OS's. Different areas of dependability enhancements are discussed, covering both fault-removal and fault-tolerance techniques.

The work in this thesis forms the basis for many interesting new research directions. The thesis is therefore concluded by scoping out multiple future directions of research. This includes both refinements and extensions as well as new exiting problems warranting further research.

8.1 Contributions

This section recaps the research questions posed in Chapter 1 and discusses the individual contributions made and their relevance.

8.1.1 Category 1: Conceptual

Research Question 1: *How do errors in device drivers propagate in an OS? What is a good model for identification of such propagation paths?*

Errors in device drivers have been shown to cause many failures in OS's and to propagate to applications running on the OS. Our model of an OS allows us to capture both errors causing severe failures in the system and those data errors propagating to applications through the OS.

The results presented in Chapter 6 clearly show that there is a significant difference across services regarding error propagation. Some services are identified to be robust, i.e, severe system states cannot be provoked through it. Other services can lead to severe consequences, including a complete system crash. Furthermore, it is shown that the context in which the service is used, e.g., the driver using it, has a significant impact on its damage potential.

Research Question 2: *What are quantifiable measures of robustness profiling of OS's?*

Chapter 5 presents a framework with which error propagation across the OS can be estimated. The measures presented allow for identifying individual services more vulnerable to propagating errors, either as sources or sinks. As such they are useful for discriminating services based on susceptibility to propagating errors, which gives developers hints on which services are more likely to experience problems during runtime. Furthermore, they allow discrimination across drivers and applications, which can be used to prioritize across multiple contending drivers/applications or for verification resource planning by managers. Components with higher exposure or diffusion should be the first targets for improvements.

Overall, the measures defined provide evaluators with the tools to make informed decisions, on various levels.

8.1.2 Category 2: Experimental Validation

Research Question 3: ***Where to inject?*** *Where are errors representing faults in drivers best injected? What are the advantages and disadvantages of different locations?*

The use of standard interfaces is beneficial since it facilitates both injection of errors and observing their effects with minimal intrusion on the system studied. It gives clear and easily interpretable feedback to the evaluator on where errors propagate and which services and drivers are more likely to spread them if they are present.

Research Question 4: ***What to inject?*** *Which error model should be used for robustness evaluation? What are the trade-offs that can be made?*

Chapter 6 evaluates three contemporary error models, chosen based on their suitability for injection at the OS-Driver interface and their prior use for this purpose. The contributions here are two-fold. First, to the best of our knowledge this thesis is the first comprehensive comparison across multiple error models at the interface level. This study highlights the strengths and weaknesses of the models, highlighting differences across them. Secondly, our comparison evaluates the models on the number of identified failures, coverage of services, execution time, efficiency and implementation complexity. This allows selecting the most appropriate error model, based on trade-offs across the parameters.

For the case study performed bit-flips reveal the most severe (**Class 3**) failures and provoke failures in the highest number of services. Additional services having severe failures can be identified using the Fuzzing error model. The least number of injections were incurred by the data type error model.

Chapter 6 further shows how a new *composite* model can be defined, combining the bit-flip and Fuzzing error models to achieve a good trade-off between efficiency, coverage and number of injections performed.

Research Question 5: ***When to inject?*** *Which timing model should be used for injection?*

The timing model used controls the time at which errors are injected, i.e., the events triggering injections. Classically, errors are injected either on first-occurrence, i.e., the first time a service is called, or injected according to some predefined time distribution. Chapter 7 proposes a novel timing model based on the usage profile of the component interface, in this case the OS-Driver interface. By first profiling the operations performed on the driver injections can be concentrated on operations using the concept of *call blocks*, i.e., repeating sequences of calls.

The new timing models allows for more focused injections, giving more comprehensive results, without requiring deep knowledge of the services in the OS-Driver interface. Furthermore, profiling of drivers reveal that certain types of drivers are more sensitive in the initialization phase, whereas some are sensitive also in the working and clean up phase, suggesting that the

first-occurrence approach may be more suitable for the former case.

8.1.3 Injection Framework

For carrying out all the fault injection experiments required a flexible and scalable fault injection framework for Windows CE .Net has been implemented. The framework allows for easy and fast extension to new error models using a plugin model for error models. The flexible architecture makes it easy to implement new error models and to incorporate new drivers.

8.2 Applications of Robustness Evaluation

This section illustrates how the robustness evaluation framework presented in the previous chapters can be used to enhance the robustness of an OS. Such evaluations can serve primarily three purposes, a) as support in the testing of systems, b) as a source for developer feedback, helping developers build more robust code by highlighting potential robustness bottlenecks using robustness profiles, and c) as input to active robustness enhancing activities, such as addition of error detection and recovery modules. The following sections detail each of these potential uses of robustness evaluation.

8.2.1 Robustness Profiling

Robustness evaluation of platforms, such as OS's, is useful because it can provide useful feedback to the developers of applications built on top of such platforms. By providing so called robustness profiles, a developer is made aware of potential robustness vulnerabilities in the system and is better equipped to make decisions on which OS services to use and the consequences that might come from using them.

Robustness profiles give information on where potential robustness bottlenecks may exist in a system or component. The robustness profile can, for instance, consist of a subset of the measures presented in Chapter 5. The information gained can be used to raise awareness among developers on the consequences faults can have for specific services. When possible, developers can elect to use different services to achieve the same goal in a safer manner.

Robustness profiles can be used as input to testers, which can focus testing on those parts of the system using vulnerable services. Robustness profiles also can be used to focus code inspections and design reviews on those parts more likely to cause damage in the system.

8.2.2 Robustness Evaluation in Testing

Robustness evaluation can be considered a special branch of testing, where focus is put on the non-functional requirements of the system. Typically a robustness evaluation requires an acceptable level of functionality to be present in the system being evaluated, i.e., the used workload must successfully execute on the OS without any errors. This implies that functional testing of the system has been carried out.

There are three phases of testing where robustness evaluation focusing on device drivers may be of great assistant:

- **Acceptance Testing:** To verify that the OS and its drivers behave at a reasonable level
- **Integration Testing:** To verify that a driver can be integrated and used in the system
- **Regression Testing:** When major configuration changes have been performed the robustness of the system needs to be re-evaluated

Acceptance Testing

As part of the requirements for a specific software component requirements on robustness may be included. This may involve specifying which services may propagate errors and/or at which severity, for instance by specifying that no service may cause a crash of the system, no matter which values it is used with. As such the presented robustness evaluation framework may be used to validate or invalidate such properties.

Integration Testing

When integrating components with each other one needs to make sure that interaction across components works as expected. Furthermore, one may be interested in evaluating the consequences faults in one component have on the other component(s). When misbehaving components are able to cause severe failures they may either require additional focused verification efforts or may need to be equipped with error handling capabilities.

The technique presented in this thesis is well suited for testing the integration of new drivers in the OS as it works on the OS-Driver interface level, which is where the interaction takes place. It is important to note though that the error propagation profiling is not aimed at evaluating drivers specifically, but the OS-Driver interaction. Therefore, detected vulnerabilities for a new driver should not automatically be "blamed" solely on the driver. Similarly,

robustness profiling does not focus on the functionality of a specific driver, and is therefore a complement to functional testing, not a replacement.

Regression Testing

As components evolve as part of the development process their error propagation abilities may change, to the better or the worse. As part of regression testing campaigns error propagation can be evaluated such that new propagation paths are detected as soon as possible and may be treated appropriately.

The scalability and automation possibilities of interface-based fault injection makes it excellent for regression testing. Using the pre-profiling approach described the injections are adopted to any changes in workload on the system. Additional injections for new services are easily defined. New error models require minimal changes to the system before inclusion.

8.2.3 Robustness Enhancing Wrappers

In many cases modifications to system components exhibiting robustness vulnerabilities are not possible, or even desirable. This is for instance the case for pure black-box systems, where the lack of access to source code prohibits any modifications. Even with access to source code, legal reasons may prohibit modifying the code. Typical robustness enhancing modifications include addition of error checking and handling code, such as executable assertions [Voas and Miller, 1994b; Hiller, 2000]. For systems geared towards high performance it may not be viable to add time-consuming checks to the components involved, especially for general-purpose systems such as OS's.

As an alternative to modifying the involved components an attractive alternative is to add new components "wrapping" the original component [Fraser et al., 1999; Ghosh et al., 1999; Mitchem et al., 2000]. Such wrappers can be added where needed and thus be applied on a policy basis or where likely to be most effective [Hiller et al., 2002a]. Error propagation analysis can be used to identify prominent propagation paths, such as in [Hiller et al., 2002a].

The data collected from fault injection experiments can be used to design assertions, which can be implemented as wrappers [Voas, 1997b; Whisnant et al., 2004]. It can also be used to enhance the wrappers design by other means. Several research projects have looked into the use of wrappers for enhancing OS robustness and security. In [Arlat et al., 2002] the authors describe fault injection campaigns carried out on two microkernel-based OS's. Robustness enhancing wrappers are added to some functional components of the OS by formally defining predicates that must hold over the course of the

execution. This requires access to internal states of the OS which is implemented using reflection. It is noted that even when such access is possible, the definition of (correct) predicates is time-consuming and difficult. The required formal models of behavior may in practice make the approach extremely difficult to implement for general purpose COTS OS's. The authors propose to instead use operational consistency checks, such as acceptance or validity checks.

In [Fetzer and Xiao, 2002b,a] wrappers are used to track non-robust arguments to C libraries made by applications. Stateful wrappers keep track of memory allocations on the heap and stack and can verify that accesses are only made to allocated memory, presented first in [Fetzer and Xiao, 2001] (a similar technique is presented in [DeVale and Koopman, 2001] for exception hardening of I/O libraries). For memory not present on the heap or on the stack signal handlers are set up to track any access violations. Furthermore, data structures are validated using existing validation functions provided by the system and when such functions are not available, state information is kept similar to memory allocation to verify the correctness of arguments. If a violation is found, a *safe return code* is returned to the application.

Nooks is an add-on subsystem to an OS, protecting the OS from a vast majority of failures in device drivers [Swift et al., 2005]. Drivers are isolated (i.e., wrapped) within lightweight protection domains. All interaction with the kernel is tracked, to both isolate failures and for facilitating cleanup procedures. The protection is achieved by limiting a driver's write access to kernel memory and by kernel object tracking mechanisms. Nooks can protect against memory violations and kernel structure corruption. Fault injection was used to validate the approach and can be used to define specific parameter checks to improve failure isolation. In later work [Swift et al., 2006], the authors extended the recovery capabilities of the system by introducing *shadow drivers*. Shadow drivers temporarily takes over while drivers are reloaded and restarted, the system also handles state information transfer to the newly started driver, making recovery transparent to the user of the system. The *micro reboot* strategy is a promising recovery approach, as it avoids time consuming and possibly disruptive system reboots [Candea et al., 2004; Herder et al., 2007].

8.3 Outlook on the Future

This section reflects on the themes presented by discussing and speculating on future steps in research needed for them to further evolve.

8.3.1 Fault Injection Technology

The fault injection performed for this thesis is based on intercepting calls made in device drivers. Thus it is based on real calls made in the system, i.e., a workload is needed to generate the required calls. This is in contrast to approach where test harnesses are set up to simulate operational conditions, where each service can be tested in isolation. The benefit of the latter approach is that more injections can be performed per time unit, but on the other hand it requires operational conditions to be set up, which may be difficult to do, especially for low-level system software, such as device drivers. Since both approaches have merits a comprehensive evaluation of both on a larger project would give insights and guidance on where one is more useful than the other.

For the fault injection approach presented here to gain widespread acceptance and adoption it needs to be incorporated into a proper tool set. Such a tool set must minimize the semantical burden on the evaluator and automate the process of evaluation as much as possible, still allowing for user-driver extensibility and scalability. These areas of the presented approach need to be handled by the tool:

- Profiling of the targeted driver, including identification of all used services and automatic generation of injection wrapper.
- Provide a selection of error models that the evaluator can choose from, as well as a standard interface for adding custom error models.
- Automatically perform the injections and collect the required logs and store them in a database.
- Provide the evaluator with the mechanisms to automatically calculate relevant measures, including error propagation.
- Additionally, throughout the process data must be stored in open formats, e.g., XML, enabling integration with external tools and future enhancements.

8.3.2 Error Propagation

There are many uses for information on error propagation, some already discussed previously in this chapter. In this thesis a four-graded scale has been used to classify each experiment into different failure classes. Propagation is then typically studied on a failure class basis. The scale used is based on severity, without any specific system in mind. However, further refinement of

the scale may be useful for specific systems, specifically incorporating application level information, such as data corruption or other application specific failures of different severity. Developing guidelines for including application specific failures and still preserving comparative capabilities is an interesting challenge.

A related issue to the inclusion of application-specific information in failure classification is to further investigate the role of the workload selection on the outcome of the evaluation. It is well established that the used workload should as closely as possible resemble the real workload on the system, i.e., one uses the operational profile of the system [Musa, 1993]. However, the used workload can also have an impact on the failure revealing capabilities of the evaluation, where some workloads may be more likely to expose propagating errors than others. For instance, applications containing some level of error checking and correction may, transparently to the user, be able to handle (i.e., not reporting or showing effect of) many of the propagating errors. When such applications are used "as is" they may hide important robustness information from the evaluator. This suggests that the best option would be to use "error revealing" applications, which is what is done in this thesis. Due to these conflicting goals, the composition of an "efficient" workload becomes complicated, as it might not reflect the actual use of the system and therefore skew the results. More research is needed into identification of both useful and realistic workloads to be used in conjunction with fault injection experiments.

Knowing which services may affect your application can be found out by studying the Service Exposure measure. This way individual services can be identified and the designer of the application can verify if these services are used in the application in the first place, and if so, that they are used properly and that propagating errors are handled. However, this process is complex and can be time-consuming, especially since it needs to be redone when changes have been made to the application. A new research direction is therefore to define *Application Exposure* measures, capturing how applications are affected by propagating errors. Then, techniques for assessment of such effects need to be found, for instance using fault injection. This is a challenging task, especially for cases when no source code is available. Finally, Application Exposure and robustness profiles of the OS are composed into a system-level robustness profile, considering the specific applications running on the system.

8.3.3 Error Models

Chapter 6 evaluated the appropriateness of three error models and the results clearly favors simpler models, such as bit-flips and fuzzing , over semantically richer models such as the data type model. This is also in line with current advances in random testing [Hamlet, 2006; Pacheco et al., 2007]. However, the results presented here must be interpreted in light of the specific case study where they were found. Therefore, more research is needed in the area of software error models, both for simpler and more complex models. The composite model presented shows that, at least for specific systems/contexts, models may have to be combined to be most effective. These models, even though covering a wide spectrum of properties, are of course not complete. The set of models evaluated should therefore be enlarged, especially considering code level faults, such as mutations, to make a more comprehensive selection of models available for system evaluators.

Also on the data level, all three models can be extended. The data-level errors can for instance be extended with semantic knowledge of the functions tested in the interface. Fuzzing can be extended using advances in random testing [Hamlet, 2006]. Bit-flips can be further extended to incorporate multiple flips (extended from the SEU model) and further extend the work on selective bit injections.

Another important research direction is to validate the proposed robustness evaluation methodology as part of a structured development process. This would allow for a stronger connection with the "bug-revealing" capabilities of the chosen error models, an important aspect not focused on in this thesis.

8.3.4 Error Timing

The timing model presented was developed within the context of device drivers. However, we believe that it could be of much more general use in component-based testing. In systems were components are seen as black boxes and robustness evaluation is warranted, the selection of triggering events is as difficult as for device drivers. An extension of the work to such systems could potentially further validate its usefulness and make it more accessible to developers.

The drivers evaluated had no concurrent access patterns, simplifying the analysis. Further research is needed to handle concurrent access patterns.

For making the call block technique more approachable it should be based on automatic identification of call blocks from a given call string. Whether complete automation is possible is still an open question, but supporting

tools can be developed based on pattern recognition techniques, such as suffix trees. Initial prototype tools have been implemented and show some promise. More research is needed in development of algorithms and tools to handle larger data sets.

8.4 Practical Lessons Learned

During the process of working on the material for this thesis several (non-scientific) lessons were learned. This section aims to list some of these lessons, and the purpose is to share our experience with these systems. Some of these may be obvious to experienced researchers, but we still hope they can be useful for young researchers and developers.

Lesson 1: Store structured data

By moving to a structured storage of data (i.e., a database vs. simple text files) has an associated overhead, in terms of time, effort and skills required. Our experience is that this cost is small compared to the benefits gained. Having data in a database makes it easy to change analysis tools, to modify the analysis or extend it. It also simplifies accessing the data as tools already exist to work with databases. It is also easy to search the data for inconsistencies, arising due to software bugs or incomplete log files, something which otherwise may be hard when the amount of data increases rapidly.

Lesson 2: Use revision control

A key to any successful programming task is securing the code from accidental (or malicious) changes. This not only includes having a structured back-up system (that is also tested!), but also to use revision control for the source code. This simplifies the task, even when there is only one developer, for instance when working on multiple machines. Additionally, putting the results (the raw log files in our case) under revision control is also beneficial, as loosing such files may destroy many hours of experimentation.

Lesson 3: Don't trust the documentation!

In many cases documentation can be outdated, or incomplete. When the system doesn't behave the way the documentation states it should, it may be that the documentation is outdated and not that something is wrong. This is especially true for articles written before the release of a software (white papers). Make sure that you have the latest version of the documentation

and search the Internet for users experiencing similar problems. News groups and web forums are good places to get answers.

Lesson 4: Use the right tool for the job

Typically there are many tools (such as programming languages) that may be used to accomplish a given task. They differ in ease of use and feature set. For instance, many high-level programming languages (such as Java/.Net) allow for very rapid development using modern development environments and provide simple to use interfaces to for instance build intuitive user interfaces, interacting with databases etc. Choose the tool that is best suited for the problem, considering the time it requires to implement the solution and the possibilities of extending it for future needs. Choosing a tool based only on familiarity might not be the best decision in the long run.

A last word from the author

This thesis has focused on identifying vulnerabilities and weaknesses in software systems, rather than on increasing the dependability of such systems. As a last comment on my work I would therefore like to paraphrase a comment made by Jim Gray in 1990. Namely that it is after all possible to build truly fault tolerant systems (containing software) having a mean time between failures of several years or more using the right techniques and tools [Gray, 1990]. It is encouraging as a software engineer to know that such goals are indeed achievable.

Bibliography

BOINC: Berkeley Open Infrastructure for Network Computing. Project web site. URL http://boinc.berkeley.edu/. Accessed 2007-10-27.

The Ballista Project. Project web site. URL http://www.ballista.org. Accessed 2007-10-27.

The EU-IST Dependability Benchmarking project (DBench). Project web site. URL http://www.dbench.org. Accessed 2007-10-27.

The Embedded Microprocessor Benchmark Consortium (EEMBC). Consortium web site. URL http://www.eembc.org. Accessed 2007-10-27.

FreeBSD Kernel Stress Test Suite. Online collection of tests. URL http://people.freebsd.org/~pho/stress/index.html. Accessed: 2007-10-27.

IEEE Standard Glossary of Software Engineering. IEEE Standard 610.12-1990, December 1990.

Intrinsyc Software. Company web site. URL http://www.intrinsyc.com. Accessed 2007-10-27.

The LINPACK. Web page. URL http://www.netlib.org/linpack/. Accessed 2007-10-27.

PROTOS - Security Testing of Protocol Implementations. Project web site. URL http://www.ee.oulu.fi/research/ouspg/protos/. Accessed: 2007-10-27.

The Standard Performance Evaluation Corporation (SPEC). Organization web site. URL http://www.spec.org. Accessed 2007-10-27.

The Transaction Processing Performance Council (TPC). Organization web site. URL http://www.tpc.org. Accessed 2007-10-27.

Transact-SQL Reference for SQL Server 2005. Online reference document. URL `http://msdn2.microsoft.com/en-us/library/ms189826.aspx`. Accessed 2007-10-27.

Arnaud Albinet, Jean Arlat, and Jean-Charles Fabre. Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 807–816, 2004.

Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Leiane Martins, and David Powell. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, February 1990.

Jean Arlat, Alain Costes, Yves Crouzet, Jean-Claude Laprie, and David Powell. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. *IEEE Transaction on Computers*, 42(8):913–923, August 1993.

Jean Arlat, Jean-Charles Fabre, Manuel Rodriguez, and Frederic Salles. Dependability of COTS Microkernel-Based Systems. *IEEE Transactions on Computers*, 51(2):138–163, February 2002.

Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January-March 2004.

Thomas Ball and Sriram Rajamani. The slam project: Debugging system software via static analysis. In *Proceedings of Symposium on Principles of Programming Languages*, pages 1–3, 2002.

James H. Barton, Edward W. Czeck, Zary Z. Segall, and Daniel P. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Transactions on Computers*, 39(4):575–582, April 1990.

Douglas Boling. *Programming Microsoft Windows CE .Net*. Microsoft Press, third edition, 2003.

Aaron B. Brown and David A. Patterson. To Err is Human. In *Procedeedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY)*, 2001.

Aaron B. Brown, Leonard C. Chung, and David A. Patterson. Including the Human Factor in Dependability Benchmarks. In *Proceedings of the DSN Workshop on Dependability Benchmarking*, pages F9–14, 2002.

George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot A Technique for Cheap Recovery. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 31–44, 2004.

João Carreira, Henrique Madeira, and João Gabriel Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, February 1998.

João Viegas Carreira, Diamantino Costa, and João Gabriel Silva. Fault Injection Spot-Checks Computer System Dependability. *IEEE Spectrum*, 36(8):50–55, August 1999.

George J. Carrette. The web site for crashme. URL `http://people.delphiforums.com/gjc/crashme.html`. Accessed 2007-10-27.

Ramesh Chandra, Ryan M. Lefever, Kaustubh R. Joshi, Michel Cukier, and William H. Sanders. A Global-State-Triggered Fault Injector for Distributed System Evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605, 2004.

Shuo Chen, Jun Xu, Ravishankar K. Iyer, and Keith Whisnant. Evaluating the Security Threat of Firewall Data Corruption Caused by Instruction Transient Errors. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 495 – 504, 2002.

Ram Chillarege. *Handbook on Software Reliability Engineering*, chapter Orthogonal Defect Classification, pages 359–400. McGraw-Hill, 1996.

Ram Chillarege and Nicholas S. Bowen. Understanding Large System Failures - A Fault Injection Experiment. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 356–363, 1989.

Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal Defect Classification - A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, November 1992.

Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An Empirical Study of Operating System Errors. In *Proceedings of Symposium on Operating Systems Principles*, pages 73–88, 2001.

Jörgen Christmansson and Ram Chillarege. Generation of an Error Set that Emulates Software Faults Based on Field Data. In *International Symposium on Fault Tolerant Computing*, pages 304–313, 1996.

Jörgen Christmansson, Martin Hiller, and Martin Rimén. An Experimental Comparison of Fault and Error Injection. In *Proceedings of the International Conference on Software Reliability Engineering*, pages 378–396, 1998.

Jeffrey Clark and Dhiraj K. Pradhan. Fault Injection: A Method for Validating Computer-System Dependability. *IEEE Computer*, 28(6):47–56, June 1995.

Christian Constantinescu. Neutron SER Characterization of Microprocessors. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 754–759, 2005.

Christian Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4):14–19, 2003.

Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, third edition, February 2005. URL `http://www.oreilly.com/catalog/linuxdrive3/book/index.csp`.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, 1978.

John DeVale and Philip Koopman. Performance Evaluation of Exception Handling in I/O Libraries. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 519–524, July 2001.

John DeVale, Philip Koopman, and David Guttendorf. The Ballista Software Robustness Testing Service. In *Proceedings of the Testing Computer Software Conference*, 1999.

Christopher P. Dingman, Joe Marshall, and Daniel P. Siewiorek. Measuring Robustness of a Fault-Tolerant Aerospace System. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 522–526, 1995.

Wenliang Du and Aditya P. Mathur. Testing for Software Vulnerability Using Environment Perturbation. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 603–612, 2000.

João Durães and Henrique Madeira. Multidimensional Characterization of the Impact of Faulty Drivers on the Operating System Behavior. *IEICE Transactions*, E86-D(12):2563–2570, December 2003.

João Durães and Henrique Madeira. Emulation of Software Faults by Educated Mutations at Machine-Code Level. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 329–340, 2002.

João Durães and Henrique Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

Christof Fetzer and Zhen Xiao. A Flexible Generator Architecture for Improving Software Dependability. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 102–113, 2002a.

Christof Fetzer and Zhen Xiao. Detecting Heap Smashing Attacks Through Fault Containment Wrappers. In *Proceedings of IEEE Symposium on Reliable Distributed Systems*, pages 80–89, 2001.

Christof Fetzer and Zhen Xiao. An Automated Approach to Increasing the Robustness of C Libraries. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 155–164, June 2002b.

Justin E. Forrester and Barton P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the USENIX Windows Systems Symposium*, pages 59–68, 2000.

Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

Archana Ganapathi and David Patterson. Crash Data Collection: A Windows Case Study. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 280–285, 2005.

Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows XP Kernel Crash Analysis. In *Proceedings of Large Installation System Administration Conference*, 2006. URL `http://www.cs.berkeley.edu/~archanag/publications/lisa.pdf`.

Anup Ghosh, Matthew Schmid, and Shah. Testing the Robustness of Windows NT Software. In *Proceedings of International Symposium on Software Reliability Engineering*, pages 231 – 235, 1998.

Anup Ghosh, Matthew Schmid, and Frank Hill. Wrapping Windows NT Software for Robustness. In *Proceedings of International Symposium on Fault-Tolerant Computing*, pages 344 – 347, 1999.

Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated Whitebox Fuzz Testing. Microsoft Technical Report MSR-TR-2007-91, Microsoft Research, July 2007.

Jim Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, 1990. ISSN 0018-9529.

Jim Gray. Why Do Computers Stop and What Can We Do About It. Technical Report TR 85.5, Tandem, 1985.

Michael Grottke and Kishor S. Trivedi. Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. *Computer*, 40(2):107–109, 2007.

Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of Linux Kernel Behavior under Errors. In *Proceedings of the International Conference on Dependable Systems and networks*, pages 459 – 468, 2003.

Weining Gu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 887 – 896, 2004.

Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

Dick Hamlet. When Only Random Testing Will Do. In *Proceedings of the International Workshop on Random testing*, pages 1–9, 2006. ISBN 1-59593-457-X. doi: http://doi.acm.org/10.1145/1145735.1145737.

Seungjae Han, Kang G. Shin, and Harold A. Rosenberg. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems. In *Proceedings of the International Computer Performance and Dependability Symposium*, pages 204–213, 1995.

Jane Huffman Hayes and Jeff Offutt. Input validation analysis and testing. *Empirical Software Engineering*, 11(4):493–522, December 2006.

John L. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, 2000.

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 41–50, 2007.

Martin Hiller. Executable Assertions for Detecting Data Errors in Embedded Control Systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 24–33, 2000.

Martin Hiller. *A Software Profiling Methodology for Design and Assessment of Dependable Software*. Ph.D. Thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2002.

Martin Hiller, Arshad Jhumka, and Neeraj Suri. On the Placement of Software Mechanisms for Detection of Data Errors. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 135–144, 2002a.

Martin Hiller, Arshad Jhumka, and Neeraj Suri. PROPANE: An Environment for Examining the Propagation of Errors in Software. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 81–85, July 2002b.

Martin Hiller, Arshad Jhumka, and Neeraj Suri. EPIC: Profiling the Propagation and Effect of Data Errors in Software. *IEEE Transactions on Computers*, 53(5):512–530, May 2004.

Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, first edition, 2006.

Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4):75–82, April 1997.

Ravishankar Iyer and Paola Velardi. Hardware-Related Software Errors: Measurement and Analysis. *IEEE Transactions on Software Engineering*, SE-11(2):223–231, 1985.

Tahar Jarboui, Jean Arlat, Yves Crouzet, and Karama Kanoun. Experimental Analysis of the Errors Induced into Linux by Three Fault Injection Techniques. In *International Conference on Dependable Systems and Networks*, pages 331– 336, 2002a.

Tahar Jarboui, Jean Arlat, Yves Crouzet, Karama Kanoun, and Thomas Marteau. Analysis of the Effects of Real and Injected Software Faults. In

Proceedings of the Pacific Rim International Symposium on Dependable Computing, pages 51–58, 2002b.

Tahar Jarboui, Jean Arlat, Yves Crouzet, Karama Kanoun, and Thomas Marteau. Impact of Internal and External Software Faults on the Linux Kernel. *IEICE Transactions on Information and Systems*, E86-D(12): 2571–2578, 2003.

Steve Jobs. Keynote talk at Apple World Wide Developers Conference, 2006.

Andréas Johansson. Dependability Benchmarking. Master's thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2001.

Ali Kalakech, Tahar Jarboui, Jean Arlat, Yves Crouzet, and Karama Kanoun. Benchmarking Operating System Dependability: Windows 2000 as a Case Study. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pages 261–270, 2004a.

Ali Kalakech, Karama Kanoun, Yves Crouzet, and Jean Arlat. Benchmarking The Dependability of Windows NT4, 2000 and XP. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 681–686, 2004b.

Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A Tool for the Validation of System Dependability Properties. In *International Symposium on Fault-Tolerant Computing*, pages 336–344, 1992.

Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transactions on Computers*, 44(2):248–260, February 1995.

Cem Kaner, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software*. John Wiley & Sons, 1999.

Karama Kanoun, Jean Arlat, Diamantion J.G. Costa, Mario DalCin, Pedro Gil, Jean-Claude Laprie, Henrique Madeira, and Neeraj Suri. Dbench (Dependability Benchmarking). In *Workshop on The Euoropean Dependability Initiative*, pages D12–15, 2001.

Karama Kanoun, Yves Crouzet, Ali Kalakech, Ana-Elena Rugina, and Philippe Rumeau. Benchmarking the Dependability of Windows and Linux using PostMark^{TM}Workloads. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 11–20, 2005.

Wei-Lun Kao and Ravishankar K. Iyer. DEFINE: A Distributed Fault Injection and Monitoring Environment. In *Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 252–259, 1994.

Wei-Lun Kao, Ravishankar K. Iyer, and Dong Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, November 1993.

Johan Karlsson, Peter Liden, Peter Dahlgren, Rolf Johansson, and Ulf Gunneflo. Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms. *IEEE Micro*, 14(1):8–23, February 1994. ISSN 0272-1732.

Philip Koopman. Toward a scalable method for quantifying aspects of fault tolerance, software assurance, and computer security. In *Computer Security, Dependability and Assurance: From Needs to Solutions, 1998. Proceedings*, pages 103–131, 1999.

Philip Koopman and John DeVale. Comparing the Robustness of POSIX Operating Systems. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 72–79, 1999.

Philip Koopman and John DeVale. The Exception Handling Effectiveness of POSIX Operating System. *IEEE Transactions on Software Engineering*, 26(9):837 – 848, September 2000.

Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing Operating Systems Using Robustness Benchmarks. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 72–79, 1997.

Nathan P Kropp, Philip J Koopman, and Daniel P Siewiorek. Automated Robustness Testing of Off-the-Shelf Software Components. In *Proceedings of the International Symposium on Fault Tolerant Computing*, pages 230–239, 1998.

Sanjeev Kumar and Kai Li. Using Model Checking to Debug Device Firmware. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2002.

Jean-Claude Laprie, editor. *Dependability: Basic Concepts and Terminology.* Springer-Verlag, 1992.

Inhwan Lee and Ravishankar K. Iyer. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 20–29, 1993.

Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

Henrique Madeira, Diamantino Costa, and Marco Vieira. On the Emulation of Software Faults by Software Fault Injection. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 417–426, June 2000.

Henrique Madeira, Raphael R. Some, F. Moreira, Diamantino Costa, and David Rennels. Experimental Evaluation of a COTS System for Space Applications. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 325– 330, 2002.

Eric Marsden and Jean-Charles Fabre. Failure Mode Analysis of CORBA Service Implementations. In *Proceedings of Middleware*, volume 2218 of *Lecture Notes on Computer Science*, pages 216–231. Springer Verlag, 2001.

Manuel Mendonca and Nuno Neves. Robustness Testing of the Windows DDK. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 554–564, 2007.

Christoph C. Michael and Ryan C. Jones. On the Uniformity of Error Propagation in Software. In *Proceedings of the Annual Conference on Computer Assurance*, pages 68–76, 1997.

Visual Studio, Microsoft Portable Executable and Common Object File Format Specification. Microsoft, 8.0 edition, May 2006. URL `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`.

Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical Report 1268, Department of Computer Sciences, University of Wisconsin, 1995.

Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, December 1990. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/96267.96279.

Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An Empirical Study of the Robustness of MacOS Applications Using Random Testing. In *Proceedings of the International Workshop on Random Testing*, pages 46–54, 2006.

Terrence Mitchem, Raymond Lu, Richard O'Brien, and Kent Larson. Linux Kernel Loadable Wrappers. In *Proceedings of DARPA Information Survivability Conference & Exposition*, volume 2, pages 296–307, 2000.

R. Moraes, R. Barbosa, J. Duraes, N. Mendes, E. Martins, and H. Madeira. Injection of faults at component interfaces and inside the component code: are they equivalent? In *Proceedings of the Dependable Computing Conference*, pages 53–64, 2006.

L.J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990. ISSN 0098-5589.

MSDN. The Microsoft Developer Network (MSDN). Online reference documents. URL `http://www.msdn.microsoft.com`. Accessed 2007-10-27.

Arup Mukherjee and Daniel P. Siewiorek. Measuring software dependability by robustness benchmarking. *IEEE Transactions on Software Engineering*, 23(6):366–378, June 1997. ISSN 0098-5589.

Brendan Murphy. Automating Software Failure Reporting. *Queue*, 2(8): 42–48, 2004.

Brendan Murphy and Björn Levidow. Windows 2000 Dependability. In *Proceedings of the Workshop on Dependable Networks and OS*, 2000.

John Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pages 14–32, March 1993.

Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, 2 edition, 2004.

Nuno Ferreira Neves, João Antunes, Miguel Correia, Paulo Veríssimo, and Rui Neves. Using Attack Injection to Discover New Vulnerabilities. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 457–466, 2006.

Wee Teck Ng and Peter M. Chen. The design and Verification of the Rio File Cache. *IEEE Transactions on Computers*, 50(4):322–337, 2001. ISSN 0018-9340.

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

Walter Oney. *Programming the MS Windows Driver Model*. Microsoft Press, 2003.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering*, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.37.

Jiantao Pan, Philip Koopman, Daniel Siewiorek, Yennun Huang, Robert Gruber, and Mimi Ling Jiang. Robustness Testing and Hardening of CORBA ORB Implementations. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 141–150, 2001.

David Powell, G. Bonn, D. Seaton, Paulo Verissimo, and F. Waeselynck. The Delta-4 Approach to Dependability in Open Distributed Computing Systems. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 246–251, 1988.

Dhiraj K. Pradhan, editor. *Fault-Tolerant Computer System Design*. Prentice Hall, 1996.

Manuel Rodriguez, Arnaud Albinet, and Jean Arlat. MAFALDA-RT: A Tool for Dependability Assessment of Real-Time Systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 267–272, 2002.

Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. FIAT - Fault Injection Based Automated Testing Environment. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 102–107, 1988.

Charles P. Shelton, Philip Koopman, and Kobey DeVale. Robustness Testing of the Microsoft Win32 API. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2000.

Kang G. Shin. HARTS: a Distributed Real-Time Architecture. *IEEE Computer*, 24(5):25–35, May 1991.

Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the Effect of Technology Trends on

the Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

Daniel Siewiorek, John J. Hudak, Byung-Hoon Suh, and Zary Segal. Development of a Benchmark to Measure System Robustness. In *Proceedings of International Symposium on Fault-Tolerant Computing*, pages 88–97, 1993.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagnet. *Operating Systems Concepts*. John Wiley & Sons, seventh edition, December 2004.

Daniel Simpson. Windows XP Embedded with Service Pack 1 Reliability. Technical report, Microsoft Corporation, 2003. URL `http://msdn2.microsoft.com/en-us/library/ms838661.aspx`. Accessed 2007-10-27.

David T. Stott, Benjamin Floering, Daniel Burke, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proceedings of the International Symposium on Computer Performance and Dependability*, pages 91–100, 2000.

Mark Sullivan and Ram Chillarege. Software Defects and their Impact on System Availability - A Study of Field Failures in Operating Systems. In *International Symposium Fault-Tolerant Computing*, pages 2–9, 1991.

Mark Sullivan and Ram Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. In *International Symposium on Fault-Tolerant Computing*, pages 475–484, 1992.

Martin Süßkraut and Christof Fetzer. Robustness and Security Hardening of COTS Software Libraries. In *International Conference on Dependable Systems and Networks*, pages 61–71, 2007.

Martin Süßkraut and Christof Fetzer. Automatically Finding and Patching Bad Error Handling. In *Proceedings of the European Dependable Computing Conference*, pages 13–22, 2006.

Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Transactions on Computer Systems*, 23(1):77–110, 2005.

Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering Device Drivers. *ACM Transactions on Computer Systems*, 24(4):333–360, November 2006.

Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2 edition, 2001.

Timothy Tsai and Navjot Singh. Reliability Testing of Applications on Windows NT. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 427–436, 2000.

Timothy K. Tsai and Ravishankar K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *Proceedings of the Performance Tools/MMB, LNCS 977*, pages 26–40. Springer Verlag, 1995.

Timothy K. Tsai, Ravishankar K. Iyer, and Doug Jewitt. An Approach towards Benchmarking of Fault-Tolerant Commercial Systems. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 314–325, 1996.

Timothy K. Tsai, Mei-Chen Hsueh, Hong Zhao, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Stress-Based and Path-Based Fault Injection. *IEEE Transactions on Computers*, 48(11):1183–1201, November 1999.

Marco Vieira and Henrique Madeira. Portable Faultloads Based on Operator Faults for DBMS Dependability Benchmarking. In *Proceedings of the International Computer Software and Applications Conference*, pages 202–209, 2004.

Marco Vieira and Henrique Madeira. Recovery and Performance Balance of a COTS DBMS in the Presence of Operator Faults. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 615–624, 2002a.

Marco Vieira and Henrique Madeira. Definition of Faultloads Based on Operator Faults for DMBS Recovery Benchmarking. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pages 265–272, 2002b.

Jeffrey Voas. Certifying Software for High-Assurance Environments. *IEEE Software*, 31(6):48–54, July-August 1999.

Jeffrey Voas. Error Propagation Analysis for COTS Systems. *IEE Computing & Control Engineering Journal*, 8(6):269–272, December 1997a.

Jeffrey Voas. Building Software Recovery Assertions from a Fault Injection-based Propagation Analysis. In *Proceedings of the International Computer Software and Applications Conference*, pages 505–510, 1997b.

Jeffrey Voas and F. Charron. Tolerant Software Interfaces: Can COTS-based Systems be Trusted Without Them? In *Proceedings of the International Conference on Computer Safety, Reliability and Security*, 1996.

Jeffrey Voas and Keith W. Miller. Dynamic Testability Analysis for Assessing Fault Tolerance. *High Integrity Systems Journal*, 1(2):171–178, 1994a.

Jeffrey Voas and Keith W. Miller. Software Testability: the New Verification. *IEEE Software*, 12(3):17–28, 1995.

Jeffrey Voas and Keith W. Miller. Putting Assertions in Their Place. In *Proccedings of the International Symposium on Software Reliability Engineering*, pages 152–157, 1994b.

Jeffrey Voas, Gary McGraw, and Anup K. Gosh. Gluing Software Together: How Good is Your Glue? In *Proceedings of the Pacific Northwest Software Quality Conference*, oct 1996.

Jeffrey Voas, Frank Charron, Gary McGraw, Keith Miller, and Michael Friedman. Predicting How Badly "Good" Software Can Behave. *IEEE Software*, 14(4):73–83, July-August 1997.

Alan R. Weiss and Richard Clucas. The Standardization of Embedded benchmarking: The Pitfalls and the opportunities. In *Proceedings of the Embedded Systems Conference*, 1999.

Elaine Weyuker. Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, 15(5):54–59, Sep. – Oct. 1998.

Keith Whisnant, Ravishankar Iyer, Zbignew Kalbarczyk, Phillip H. Jones III, David Rennels, and Raphael Some. The Effects of an ARMOR-Based SIFT Environment on the Performance and Dependability of User Applications. *IEEE Transactions on Software Engineering*, 30(4):257–277, April 2004.

James A. Whittaker. *How to Break Software.* Addison-Wesley, 2003.

Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Networked Windows NT System Field Failure Data Analysis. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, pages 178–185, 1999.

Jun Xu, Shuo Chen, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. An Experimental Study of Security Vulnerabilities Caused by Errors. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 421–432, 2001.

Steven J. Zeil. Testing for Perturbations of Program Statements. *IEEE Transactions on Software Engineering*, SE-9(3):335–346, May 1983.

CV

Personal Data

Name: Andréas Johansson

Date of birth: March 24, 1977

Place of birth: Falkenberg, Sweden

School Education

1984-1993 Apelskolan, Ullared, Sweden

1993-1996 Falkenbergs Gymnasieskola, Falkenberg, Sweden

University Education

1997-2001 Master of Science in Computer Engineering, Chalmers, Göteborg, Sweden

2002-2008 Ph.D. in Computer Science, Technische Universität Darmstadt, Darmstadt, Germany