The Impact of Fault Models on Software Robustness Evaluations

Stefan Winter
TU Darmstadt
sw@cs.tu-darmstadt.de

Constantin Sârbu
TU Darmstadt
cs@cs.tu-darmstadt.de

Neeraj Suri
TU Darmstadt
suri@cs.tu-darmstadt.de

Brendan Murphy
Microsoft Research
bmurphy@microsoft.com

ABSTRACT

Following the design and in-lab testing of software, the evaluation of its resilience to actual operational perturbations in the field is a key validation need. Software-implemented fault injection (SWIFI) is a widely used approach for evaluating the robustness of software components. Recent research [24, 18] indicates that the selection of the applied fault model has considerable influence on the results of SWIFI-based evaluations, thereby raising the question how to select appropriate fault models (i.e. that provide justified robustness evidence).

This paper proposes several metrics for comparatively evaluating fault models's abilities to reveal robustness vulnerabilities. It demonstrates their application in the context of OS device drivers by investigating the influence (and relative utility) of four commonly used fault models, i.e. *bit flips* (in function parameters and in binaries), *data type dependent parameter corruptions*, and *parameter fuzzing*. We assess the efficiency of these models at detecting robustness vulnerabilities during the SWIFI evaluation of a real embedded operating system kernel and discuss application guidelines for our metrics alongside.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault tolerance

General Terms

Measurement, Performance, Reliability

Keywords

Robustness Testing, Fault Injection, Fault Models

1. INTRODUCTION

Under a constant feature-driven market pressure and due to their ever increasing complexity, many software applications are often released without being sufficiently tested. Even if a software component[1] is considered to be sufficiently tested for one application scenario or operational environment, it may be insufficiently tested for reuse in another one. Especially commercial-off-the-shelf (COTS) commodity software components pose a problem in this respect. They are subject to (re)use in a variety of application scenarios that may well be unforeseen by their developers. This lack of knowledge on the intended application scenario makes it difficult for developers to estimate the sufficiency of their verification and validation efforts.

In contrast, the users of COTS components are aware of the application scenarios in which they are planning to use them. However, their ability to sufficiently test COTS components is frequently also limited, as they usually do not have access to the components's source code or other information available to the components's developers. Furthermore, the same component may be applied in multiple different operational environments within the same application: the same COTS operating system can for instance be applied for both the client and the server in a distributed client-server application. Since the operational conditions for these different applications of the same system differ, the user of this component needs to test the component for both of them differently. Thus, failures frequently result when the operational environment of the deployed system differs from the pre-release lab configurations used in the testing phase of the software development process.

To cope with this deployment variability aspect, SWIFI is a popular technique for evaluating the robustness of commodity commercial software components with respect to unexpected operational conditions. In order to assess whether a software component is "sufficiently" robust for release, it is exposed to operational perturbations in a controlled manner while its reactions are closely monitored.

The problem with such approaches is that they are well suited for demonstrating the *presence* of vulnerabilities, but not the *absence* thereof. In order to show the absence of vulnerabilities using SWIFI, it would be necessary to expose the component under evaluation (CUE) to every possible perturbation. While this is theoretically possible if the component's input space is finite, it is generally considered impracticable as the problem is equivalent to that of exhaustive testing [11].

SWIFI-based robustness evaluations therefore require selecting a processable number of perturbations for injection. This raises the question *how to select the injected perturba-*

ICSE '11, May 21–28, 2011, Honolulu, Hawaii, USA

[1]We adopt Szyperski's notion of software components: units of composition with contractually specified interfaces and explicit context dependencies only [30].

tions to maximize the robustness evidence derivable from the experimental evaluation. A common solution to this problem is provided by application-specific *operational profiles*, i.e. probability distributions for stimuli that software is being exposed to during operation. Their application to the perturbation selection problem is based on the argument that a total absence of vulnerabilities is a too strong condition, since a vulnerability does not necessarily result in a robustness violation. It does so only if the component is exposed to a perturbation which exploits this vulnerability. This reduces the problem to proving the absence of all vulnerabilities that can be exploited during operation. For this purpose operational profiles of the component are derived from its intended operational context, including (but not limited to) *typical* classes of perturbations, i.e. classes of perturbations that are expected to be frequently encountered during operation.

However, this approach has two major drawbacks. First, the robustness evidence derived from such an evaluation is only valid for the considered operational context of the evaluated component; this implies that a component needs to be re-evaluated for every intended context, thereby impeding its reusability. Second, a vulnerability, whose exploitation is highly unlikely in a given application scenario but whose exploitation consequences are of disastrous impact, would be ignored by operational profiles. However, critical system failures often result from highly unlikely and hence unexpected conditions that are by definition not covered by operational profiles [10, 31, 21].

Paper Contributions. Considering these drawbacks, we investigate an alternate approach based on the idea that the absence of vulnerabilities can be approximated by maximizing vulnerability detection and removal. Instead of selecting *typical*, application-specific perturbation classes for robustness testing, we give preference to those perturbation classes that make our evaluations *most efficient* in terms of detected vulnerabilities and the required effort for their detection.

Consequently, this paper makes the following contributions to the current state of the art:

- a method to compare the efficiency of different perturbation classes (termed *fault models*) for SWIFI robustness evaluations;
- a set of fault model efficiency metrics for this purpose;
- a pragmatic set of guidelines for the application of these metrics in SWIFI-based robustness evaluations.

As a case study, we perform SWIFI experimentation on an embedded operating system (OS) kernel (Windows CE 4.2) and demonstrate the effectiveness of our approach, comparing four commonly applied fault models.

The paper is organized as follows: Section 2 introduces basic terminology and related work. Section 3 introduces the metrics developed for cross-model efficiency evaluations, whose application is demonstrated in an experimental evaluation of our approach presented in Section 4 along with a demonstration and discussion of their utility in Section 5.

2. BACKGROUND AND RELATED WORK

2.1 Robustness Notion and System Model

In this paper we adopt the robustness notion from [12], according to which robustness is *"the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions."*

Invalid inputs and stressful environmental conditions are termed *perturbations*. If a perturbation causes a software component to enter an erroneous state, the component possesses a *vulnerability* which is activated by the perturbation. Perturbations are equivalent to what is called *external faults* in [3], whereas robustness vulnerabilities are termed *internal faults*. The application of (external) fault injection is hence a sound method for robustness evaluations.

A software CUE is expected to interact with other software components via explicit interfaces only. Being part of a composition, components are expected to provide *services* that are relevant for the implementation of the composition's *functionality*. *Interfaces* are sets of services and therefore constitute the means by which each component's functionality can be accessed.

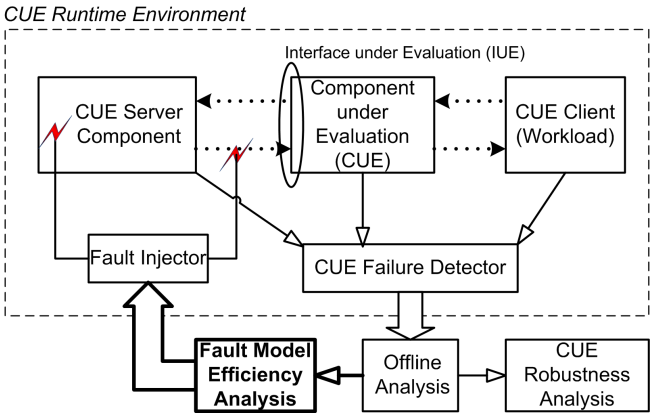

Figure 1: Contribution of our work in the context of SWIFI-based robustness evaluations

Figure 1 displays the considered robustness evaluation methodology. We conduct robustness evaluations of a CUE by injecting perturbations into its runtime environment during controlled executions. Perturbations are introduced into CUE *Servers*, i.e. components offering services to the CUE and on whose reliable provision of service the CUE depends. In order to trigger interactions of the CUE with the CUE Servers, CUE *Clients* use services provided by the CUE, i.e. create a *workload* for the CUE.

Since we consider direct interactions across the software components, any perturbation affecting a component must be mediated by its interfaces. We therefore consider direct fault injections into data passed to the CUE via the *interface under evaluation* (IUE) as well as injections into the binaries of CUE Server components interacting with the CUE via the IUE. We do not consider injections into the CUE itself, since the robustness of a software component is defined by its fault tolerance with respect to external faults.

A CUE *failure detector* monitors the CUE, its clients, and its servers for symptoms of predefined CUE *failure modes*[2] of interest. The results of injection experiments are reported to a component external to the CUE's runtime environment for offline analysis from which robustness properties of the CUE are derived according to a set of robustness metrics.

For our discussion of SWIFI-based robustness evaluations, the following terms are used. An *injection run* refers to

[2]Failure modes describe *how* a component can possibly fail, e.g. by becoming unresponsive.

Table 1: Fault models for robustness evaluations in the literature

Framework/Authors	Fault Location	Fault Type	Fault Latency	Injection Trigger
MAFALDA [2]	CUE Server IUE	SEU MBU DT	Transient Permanent	1^{st} occurrence
Albinet et al. [1]	IUE	DT	Transient	1^{st} occurrence
Kalakech et al. [19]	IUE	SEU DT	Transient	1^{st} occurrence
Xception [6]	CUE Server IUE	SEU MBU DT FZ	Transient Intermittent Permanent	1^{st} occurrence n^{th} occurrence Timer X-call
G-SWFIT [8]	CUE Server	Coding mistakes	Permanent	1^{st} occurrence
Medonça & Neves [23]	CUE Server	Coding mistakes	Permanent	1^{st} occurrence
Johansson [15]	IUE	SEU DT FZ	Transient Intermittent Permanent	1^{st} occurrence n^{th} occurrence Timer X-call Call Block

CUE	component under evaluation
IUE	interface under evaluation
SEU	single event upset
MBU	multiple bit upset
DT	data type dependent corruption
FZ	fuzzing

the injection of a specific fault, the execution of the workload, and the observation (and logging) of the injection's effects. An *injection campaign* is a collection of injection runs, pertaining to a specific fault model, workload, targeted CUE Server component, and IUE. A set of injection campaigns targeting the same CUE is called an *evaluation* of that CUE, possibly including multiple fault models, workloads, and IUEs (thus, also multiple CUE Server components).

The approach presented in this paper uses experimental data of injection campaigns performed for robustness evaluations to evaluate the applied fault model's efficiency. This information can be used to introduce a feedback loop as highlighted in Figure 1, providing guidance on the selection of perturbations for subsequent campaigns in an evaluation. The required information is defined by a set of metrics introduced in Section 3.

2.2 Comparative Fault Model Evaluations

We adopt the fault model notion from [16, 18, 17], where fault models[3] for SWIFI-based robustness evaluations of operating systems (OSs) were defined by three basic attributes: the *fault location* (where to inject), the *fault type* (what to inject), and the *fault timing* (when to inject), where the latter was further qualified as *injection trigger* and *fault latency*.

Table 1 lists a number of existing SWIFI frameworks that have been applied for software robustness evaluations similar to those we consider in this paper, along with their reported fault model support. Fault models are usually discussed only implicitly in the literature and some SWIFI frameworks provide mechanisms for flexibly extending the set of supported fault models, e.g. by user defined injection triggers or fault types. Thus, Table 1 is likely incomplete, but already indicates potentially large numbers of applicable fault models offered to evaluators, e.g. more than 90 possible attribute combinations for Xception [6].

The spectrum of **fault types** in the related literature are single bit flips (single-event upset, SEU), multiple simultaneous bit flips (multiple bit upset, MBU), data type dependent corruptions of data fields or parameters (DT), and substitution by random bit pattern (fuzzing, FZ). The **fault latency** addresses the duration of an injection in terms of repeated fault activation. Transient faults are activated exactly once, intermittent faults are activated a finite number of times, and permanent faults are activated every time. **Injection triggers** determine when a fault is injected, respectively when it can be activated for the first time. It can be injected the first time its injection location is referenced, after N references, after a predefined amount of time has passed, after some predefined exception or function call occurs (X-call), or after a predefined sequence of function calls (Call Block).

Although a multitude of fault models for software robustness evaluations are discussed in the literature, we are aware of only two attempts to compare models. Moraes et al. [24] compared the effects of fault injections at different locations, i.e. at the IUE and inside of components providing services to the CUE. They concluded that interface injections, which are generally less costly in terms of implementation complexity and execution time, are not a valid substitution for the emulation of programming mistakes within CUE Servers. The comparison covered only this single aspect, while other differences, e.g. in terms of applied fault types, were not taken into consideration.

Johansson et al. [18] compared the effects of three different fault models for interface injections on robustness evaluations of Windows CE .NET. However, the applied metrics were only suited for this specific evaluation and restricted to a specific CUE failure mode, i.e. system crashes. It is therefore not applicable when different failure modes, e.g. incorrect computation results, are considered most critical for a given application. Furthermore, as the applied SWIFI framework did not support injections into CUE Servers, the comparison of models with differing fault locations as the one provided in [24] was impossible.

[3] also termed as *error models* in the referenced papers

We reuse the SWIFI framework of [15, 18] and extend it to perform comparisons, as initiated in [24]. In the following section we extend the work presented in [18] and develop generalized and formally accurate definitions the informally specified metrics so that they can be used for (a) any robustness evaluation that aligns with the generic method described in Section 2.1 and (b) quantitative comparisons, i.e. we give quantitative re-definitions providing ratio scale measures for all considered metrics.

3. FAULT MODEL EFFICIENCY METRICS

We propose four metrics for capturing the efficiency of fault models by applying them to injection campaigns that utilize the respective models. Two of the metrics are *benefit-oriented*, i.e. the measures are preferably maximized, and the other two are *cost-oriented*, i.e. the measures are preferably minimized.

The proposed metrics are intended to lay the foundation of objective fault model comparisons. They provide campaign-granularity measures that can either be used for post-evaluation comparisons (in order to guide future evaluations of similar systems) or in-evaluation comparisons (in order to design campaigns based on evaluations of previous campaigns within the same evaluation).

We would like to emphasize that the proposed metrics are defined in a way that they reflect each considered failure mode of the CUE. Having separate measurements for different failure modes enables evaluators to weigh the obtained results according to the (application-specific) severity associated with the respective failure mode. Measurements for all possible failure modes can be performed in one single evaluation. If a different operational context is considered for the same CUE, only a modification of these weights is required instead of a re-evaluation.

3.1 Metric 1: IUE Coverage

The *IUE Coverage* of a fault model is the extent to which it enables the identification of vulnerable services provided by the CUE. A model's coverage cannot directly be used for comparisons across different CUEs, because no relative measure can be established due to the unknown total number of vulnerable services in the given evaluation context. It may, however, be used for the comparison of different fault models applied in robustness evaluations of the same CUE. *Unique coverage* denotes the number of vulnerable services detected by only one model among all compared fault models.

For the definition of IUE coverage, a precise notion of *service vulnerability* is required. A service of the CUE is termed as *vulnerable* with respect to a certain failure mode if there is a known injection run targeting this service that results in a respective CUE failure.

DEFINITION 1. *For a given injection campaign c, let $V_{\mathrm{fm}}(c)$ denote the set of identified vulnerable services with respect to failure mode* fm *and let $S_t^{\mathrm{IUE}}(c)$ denote the set of all services in the IUE that are targeted during campaign c with* $\mathrm{IUE}(c)$ *denoting the IUE targeted in c. The* IUE coverage $\mathrm{cov}_{\mathrm{fm}}^{\mathrm{IUE}}(c)$ *of this campaign with respect to* fm *is then defined by the cardinalities of V_{fm} and $S_t^{\mathrm{IUE}}(c)$ as*

$$\mathrm{cov}_{\mathrm{fm}}^{\mathrm{IUE}}(c) = \frac{|V_{\mathrm{fm}}(c)|}{|S_t^{\mathrm{IUE}}(c)|} \tag{1}$$

For a set of injection campaigns $C = \{c_1, c_2, \ldots, c_n\}$ that only differ in terms of the applied fault model, the number of services uniquely covered *during some injection campaign $c_i \in C$ is given by*

$$\overline{\mathrm{cov}}_{\mathrm{fm}}^{\mathrm{IUE}}(c_i) = \frac{|V_{\mathrm{fm}}(c_i) \setminus \bigcup_{\substack{c_j \in C \\ j \neq i}} V_{\mathrm{fm}}(c_j)|}{|S_t^{\mathrm{IUE}}(c_i)|} \tag{2}$$

We have defined coverage to reflect the targeted IUE as we have observed significant differences depending on the applied fault model and the IUE. This information is lost when coverage is defined globally for all services that the CUE uses for interaction. Consider for instance the case where a CUE has two interfaces with unequal numbers of services and that each interface is exclusively targeted by some specific fault model. Then each model may cover all of the services belonging to the interface it targets. Overall, the model targeting the "larger" interface would yield a better coverage (and a better unique coverage) and might consequently be given preference over the model targeting the "smaller" interface. However, it would be transparent to the evaluator that this seemingly preferable model fails to cover any service in the other (potentially highly critical) interface.

If an injected fault triggers multiple vulnerabilities, the effect of one fault may be masked by the effect of another fault and may therefore remain undetected. This can be interpreted as a weakness of the applied fault model that fails to trigger the vulnerabilities independently, which is reflected by the definition of coverage. The advantage of triggering multiple vulnerabilities in a single run vanishes with the ability to detect them. As coverage denotes the ability of a fault model to trigger present robustness vulnerabilities, it is a benefit-oriented metric and thus preferably maximized.

3.2 Metric 2: Injection Efficiency

Injection efficiency is defined by the number of failures observed per injection. This metric puts the number of observed failures (which is considered a measure for the experiments's effectiveness) in relation to the number of injections necessary to provoke this number of failures.

If several distinct failure modes are considered in a robustness evaluation, the obtained injection efficiencies have to be weighted according to (application-specific) robustness requirements of the CUE's intended application. Since the presented metric is intended to abstract as far as possible from concrete application requirements, a separate injection efficiency measure is considered for every failure mode. With respect to possible error masking effects in cases where multiple vulnerabilities are triggered during a single injection run, the inutility argument from the coverage metric introduction applies equally and is consistently reflected by the metric definition. As injection efficiency displays the ability of a fault model to trigger vulnerabilities leading to failures of a particular failure mode, it is a benefit-oriented metric and thus preferably maximized.

DEFINITION 2. *If for a given injection campaign c the number of injection runs is denoted by $r(c)$ and for each failure mode* fm *the number of observed failures is denoted by $f_{\mathrm{fm}}(c)$, then the injection efficiency* $\mathrm{ie}_{\mathrm{fm}}(c)$ *is given by*

$$\mathrm{ie}_{\mathrm{fm}}(c) = \frac{f_{\mathrm{fm}}(c)}{r(c)} \tag{3}$$

3.3 Metric 3: Average Execution Time

The average *execution time* of an injection run shows how efficiently a robustness evaluation can be performed with a

specific model. Since an absolute measure (for a whole evaluation) or even a per-injection measure would ignore the fact that the outcome of an injection influences the execution time dramatically (e.g. system crashes usually require a restart), the metric must also take the outcome of each injection run into consideration.

Hence, the execution time for the (in terms of correlated processing overhead) lowest common failure mode among evaluations with different fault models is a promising indicator. However "System Crash" or "Hang" failure modes should be excluded, as the timeouts usually applied for their detection can strongly influence the results. A potentially considered "No Failure" mode might as well be excluded because system failures eventually necessitate further processing of the results, for instance to provide more detailed (potentially fault type specific) logging and reporting functionality. However, as there may be scenarios when an inclusion of this mode is indispensable for provision of meaningful results[4], its assessment is highly recommended even if its contribution in a specific comparison may be of minor significance. The average execution time per injection run is a cost-oriented metric and, hence, preferably minimized.

DEFINITION 3. *If for a given injection campaign c the cumulated execution time of all injection runs that resulted in failure mode* fm *is denoted by* $\mathrm{cet}_{\mathrm{fm}}(c)$ *and the number of observed failures of mode* fm *is given by* $f_{\mathrm{fm}}(c)$*, the execution time* $\mathrm{et}_{\mathrm{fm}}(c)$ *for each failure mode is defined as*

$$\mathrm{et}_{\mathrm{fm}}(c) = \frac{\mathrm{cet}_{\mathrm{fm}}(c)}{f_{\mathrm{fm}}(c)} \tag{4}$$

3.4 Metric 4: Implementation Complexity

The *implementation complexity* of a fault model indicates the required effort to set up a fault injection mechanism for the respective model. This may be measured *a posteriori* by the amount of implementation time (e.g. in so-called *man hours* of uninterrupted work), *source lines of code* (SLOC), or any other software complexity measure as long as it is measured uniformly for all model implementations.

Any implementation complexity metric suitable for fault model comparison should mainly rely on comparisons of implementation efforts inherent in the model's requirements or properties. If, for example, the injection mechanism of one model requires the modification of one service invocation, while another requires setting up a table and monitoring a number of invocations before the actual injection can take place, the implementation complexity is clearly higher than in the first case, since additional logic and data structures are required. Notably, these attributes are related to properties of the applied fault model and not a purely syntactical complexity measure of the actual injector's implementation.

We have considered several approaches for estimating the implementation effort *a priori* on the basis of functionality-related properties, e.g. *IFPUG function points* [14] and *COSMIC full function points* [13], but none of them seems to be sufficiently accurate without experience in terms of either expert knowledge or statistical data on previous projects.

We therefore decided to use *Delivered Source Instructions* (DSI) to measure implementation complexity a posteriori. Being a SLOC metric, DSI possesses the disadvantages of purely syntactical metrics discussed above. However, as conceptually preferable function point metrics cannot be expected to provide more accurate measures without a considerable amount of expertise, DSI is chosen as a more easily applicable alternative. Restrictions only apply to measurement comparisons among different developers, implementation languages, or IDEs. If these are the same for all compared models, syntactical metrics enable valid cross-model comparisons. In our targeted scenario, the application of different fault models with one given injection framework maintained by one (or only few) evaluators is being considered.

Another a posteriori measure used to compensate the lack of a direct functional measure is McCabe's *cyclomatic complexity* [22]. While DSI measures implementation complexity as the amount of delivered code lines, cyclomatic complexity takes a more algorithmic and less implementation specific approach, considering the control flow of the algorithm rather than the size of its implementation.

Both implementation complexity measures are cumulated for all source code related to the fault model implementation.

DEFINITION 4. *The implementation complexity of a fault model is given by the DSI sum* $\mathrm{ic}_{\mathrm{DSI}}$ *accumulating the DSIs of all related source code files and the sum of the cyclomatic complexities of all associated functions* $\mathrm{ic}_{\mathrm{cyc}}$*.*

In accordance with Boehm's definition in [5], the number of DSIs equals the number of non-comment source lines.

DSI fulfills COCOMO's [4] requirements on code counting and the obtained measures can thus directly be applied for COCOMO estimations by any software project manager using estimation variable values that match the situation of his or her particular development team best.

Our definition of implementation complexity does not include efforts for instrumenting software components or for setting up the CUE's operational environment for the evaluation. The reason is that most modern fault injection frameworks perform instrumentations for fault injection online to enable more flexible injection triggers. In order to inject a transient fault into a CUE Server component after a certain time interval, for instance, the CUE Server needs to be modified during operation in the course of the evaluation. Such instrumentations are therefore captured by the execution time metric. Furthermore, we do not consider efforts related to installing already implemented fault models, as we are not aware of any fault injection framework that enables an extension by existing fault model implementations from other frameworks. We also do not consider efforts for the initial setup of the injection framework, because these depend more on the actual framework than on a specific fault model.

We are aware of the limitations of this combined metric to adequately measure implementation complexity, but nonetheless suggest its application since all contending metrics that we are aware of have similar limitations. Both DSI and cyclomatic complexity provide ratio scale measures. Implementation complexity is a cost-oriented metric and, thus, preferably minimized.

4. METRIC APPLICATION

We have validated the applicability of the proposed metrics, applying four different fault models in a robustness evaluation of the Windows CE 4.2 kernel. In the following we present the experiment setup and the results we obtained using the previously introduced metrics.

[4]e.g. if there are only two distinct failure modes or in case of comparing fault models evaluated among experimental setups with different CUE failure modes/detectors

4.1 Experiment Setup

Figure 2 provides an overview on the experiment setup. The CUE is the OS kernel of Windows CE .NET 4.2. The CUE Server components targeted for injection are device drivers, used by the CUE to access the system's hardware. We chose the OS's driver interfaces for evaluation, as drivers constitute a major cause for OS outages [7, 28, 9].

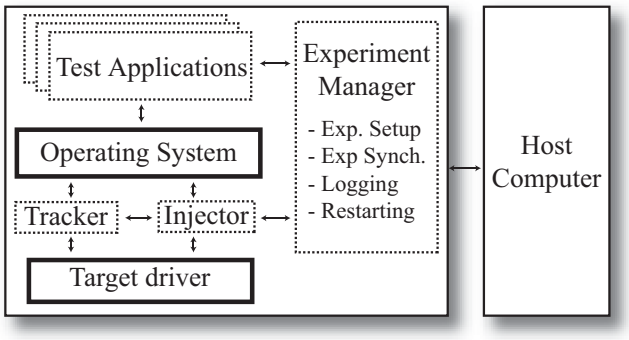

Figure 2: Setup for experimental evaluations

The targeted drivers are a serial port driver, an Ethernet driver, and a CompactFlash card driver. Each of these drivers provides an interface (the driver's *exported* interface, EXP) to the OS kernel and in turn uses a number of interfaces provided by the kernel, i.e. the Device Driver Kit (DDK) [26], kernel core functionality (CORE) [25], and the Network Driver Interface Specification (NDIS) [27].

Test applications constitute the CUE Clients intended to trigger the execution of targeted services. We use test applications specifically designed to trigger executions of the targeted drivers.

The fault injectors are implemented as software wrappers located between the CUE and the targeted CUE Servers, intercepting mutual service invocations. Faults are injected either by corrupting parameters of intercepted service invocations or by modifying drivers's binary images.

We evaluate the performance of four different fault models for a robustness evaluation of the CUE using the SWIFI tool from [15]. Three models have been implemented for extensive experimentation by our group [18, 17]. The fourth model was additionally implemented to enable comparisons of models with differing fault locations, such as in [24]. The applied fault types are representative for a large class of fault models (cf. Table 1 in Section 2) spanning:

Parameter Bit Flips (BF) A parameter of an intercepted service call in the IUE is altered by changing the value of one or more bits in its binary value. This model is intended to simulate *single event upsets* (SEUs) in hardware components that propagate through the CUE Servers to the IUE. Injections of these faults are frequently performed at the IUE for efficiency considerations.

Data Type Dependent Parameter Corruption (DT) A parameter of an intercepted service call in the IUE is changed to another value of the same data type. The main reason for taking the range of a data type into account when altering a value is the considered type of faults. If a data-type-related fault results from a *programming mistake* in a software component, then at least obvious violations of a programming language's type concept can be detected at compile time and removed prior to deployment. Thus, a large fraction of the faults that remain in a deployed component can be expected to result in erroneous run time values that are of the same or a similar data type as the correct value.

Parameter Fuzzing (FZ) Parameters of intercepted service calls in the IUE are replaced by random values, uniformly selected across all possible values for the system architecture's word size. As opposed to the BF and DT fault models, FZ is inspired by random testing rather than systematic test case derivation.

Single Event Upsets in Binaries (SEU) Faults are injected by randomly flipping single bits in CUE Server binaries. The fault type is derived from SEU effects that physical perturbations have on computer hardware, i.e. bit flips in memory and the processing circuitry. The SEU model differs from the BF model only in terms of the injection location.

We restrict our comparison to fault models with equal fault timing properties in order to single out the effects of differing fault types and locations more clearly. We chose to inject transient faults occurring on the first call to a targeted service in order to minimize the runtime of our comparison.

The *experiment manager* component controls the execution of experiments. While the interceptors (i.e. tracker and injector in Figure 2) are monitored for experiment progress, the CUE, its Servers, and Clients are monitored for failures during an injection run. Evaluation-relevant events are logged by sending them to a logging server on the host computer. After the completion of an injection run, the target manager restarts the system in order to run the next injection on a clean system image that does not carry any possibly induced dormant faults or errors. This is necessary to keep individual injection runs as independent as possible and, thus, their results individually reproducible.

We consider four different CUE failure modes that have been discussed frequently in the literature (cf. [20]).

No Failure (NF) No effect is detected. An injected fault may be either not activated or masked by the OS.

Application Error (AE) The injected fault is activated and the error propagates to the test applications, but no service specification is violated. This is for instance the case when the OS detects an erroneous state and returns an exception to the test application. Incorrect results returned by an OS service that nonetheless satisfy its specification in terms of robustness (i.e. that are wrong but still within the valid data range for the result) also fall in this category. This failure mode does not cover incorrect results that violate the specified data ranges of services.

Application Hang (AH) The injected fault is activated and the error propagates to the application interface. The specification of the fault triggering OS service is violated. The detectable effect is either an incorrect result that violates the specified data range of the service, abnormal test application termination, or lacking or wrong error codes.

System Crash (SC) The injected fault is activated and causes the OS to hang or crash, meaning that it stops to provide services to any client. Although the system may manage to reboot autonomously in some cases, external monitoring and control is generally required for SC failure detection and the system is usually recovered by a manual reset.

4.2 Experimental Results

As the performed injection campaigns only differ in terms of the applied fault models and the injection target, these two properties are used to identify particular campaigns in the following discussion.

Table 2: Coverages for failure mode Application Error (AE) [%]

Campaign	$\text{cov}_{\text{AE}}^{\text{DDK}}$	$\overline{\text{cov}}_{\text{AE}}^{\text{DDK}}$	$\text{cov}_{\text{AE}}^{\text{CORE}}$	$\overline{\text{cov}}_{\text{AE}}^{\text{CORE}}$	$\text{cov}_{\text{AE}}^{\text{NDIS}}$	$\overline{\text{cov}}_{\text{AE}}^{\text{NDIS}}$	$\text{cov}_{\text{AE}}^{\text{EXP}}$	$\overline{\text{cov}}_{\text{AE}}^{\text{EXP}}$
c_{BF}	71.43	0.0	31.03	0.0	57.14	0.0	30.77	0.0
c_{DT}	71.43	0.0	27.59	1.15	9.92	0.0	0.0	0.0
c_{FZ}	28.57	14.29	40.23	16.09	64.29	7.14	0.0	0.0
c_{SEU}	0.0	0.0	0.0	0.0	0.0	0.0	92.31	46.15

Table 3: Coverages for failure mode Application Hang (AH) [%]

Campaign	$\text{cov}_{\text{AH}}^{\text{DDK}}$	$\overline{\text{cov}}_{\text{AH}}^{\text{DDK}}$	$\text{cov}_{\text{AH}}^{\text{CORE}}$	$\overline{\text{cov}}_{\text{AH}}^{\text{CORE}}$	$\text{cov}_{\text{AH}}^{\text{NDIS}}$	$\overline{\text{cov}}_{\text{AH}}^{\text{NDIS}}$	$\text{cov}_{\text{AH}}^{\text{EXP}}$	$\overline{\text{cov}}_{\text{AH}}^{\text{EXP}}$
c_{BF}	14.29	0.0	14.94	4.60	57.14	57.14	7.69	0.0
c_{DT}	28.57	0.0	13.79	4.60	0.0	0.0	15.38	0.0
c_{FZ}	57.14	42.86	25.29	13.79	0.0	0.0	30.77	7.69
c_{SEU}	0.0	0.0	0.0	0.0	0.0	0.0	53.38	23.08

Table 4: Coverages for failure mode System Crash (SC) [%]

Campaign	$\text{cov}_{\text{SC}}^{\text{DDK}}$	$\overline{\text{cov}}_{\text{SC}}^{\text{DDK}}$	$\text{cov}_{\text{SC}}^{\text{CORE}}$	$\overline{\text{cov}}_{\text{SC}}^{\text{CORE}}$	$\text{cov}_{\text{SC}}^{\text{NDIS}}$	$\overline{\text{cov}}_{\text{SC}}^{\text{NDIS}}$	$\text{cov}_{\text{SC}}^{\text{EXP}}$	$\overline{\text{cov}}_{\text{SC}}^{\text{EXP}}$
c_{BF}	14.29	0.0	18.39	5.75	35.71	14.29	0.0	0.0
c_{DT}	14.29	0.0	9.20	0.0	21.43	0.0	0.0	0.0
c_{FZ}	14.29	0.0	10.34	2.30	7.14	0.0	0.0	0.0
c_{SEU}	0.0	0.0	0.0	0.0	0.0	0.0	92.31	92.31

DDK	device driver kit
CORE	kernel core functions
NDIS	network driver interface
EXP	exported driver interface
BF	parameter bit flips
DT	data type dependent parameter corruption
FZ	parameter fuzzing
SEU	single-event upsets in binaries

Three of the proposed metrics (IUE coverage, injection efficiency, execution time) were directly derived from the same data that was collected for evaluating the robustness of the CUE. Additional tools were used to assess the implementation complexity of the implemented fault models. DSIs were counted using SLOCCount 2.26 [32]. McCabe's cyclomatic complexity was measured using SourceMonitor 2.5 [29]. We now overview the experimental results for the four targeted metric types, and discuss their implications in Section 5. The results were obtained by conducting more than 300 injections per model and driver.

IUE Coverage The obtained coverage values are grouped by detected failure modes in Tables 2, 3, and 4. For each IUE (DDK, CORE, NDIS, EXP), the coverage $\text{cov}_{\text{fm}}^{\text{IUE}}$ and unique coverage $\overline{\text{cov}}_{\text{fm}}^{\text{IUE}}$ are given in the tables according to Definition 1. The presented numbers reflect the fraction of all services in the respective IUE that were covered by each model. The first column of Table 2, for instance, provides a comparison of the four applied fault models with respect to the percentage of services in the DDK interface for which they have detected vulnerabilities that led to AE failures. The second column reveals that although BF and DT cover a larger fraction of DDK services, FZ covers 14.29% of all DDK services *uniquely*, i.e. these services are not covered by BF or DT. A combination of FZ and either DT or BF would thus yield a coverage of 85.72% of all services provided by the DDK interface. A coverage of 0.0 means that no service of the respective IUE was identified as vulnerable. A unique coverage of 0.0 means that every covered service was also identified by some other model. The presented data noticeably contains identical numbers, e.g. 14.29 occurs five times. The reason is that every targeted interface comprises less than 100 services. 14.29% is one out of 7, which is the number of services used by the drivers in the DDK and the NDIS interfaces.

Injection Campaign Efficiency Figure 3 illustrates the cumulated injection efficiencies for each model and every targeted driver. Regarding the SC failure mode, SEU outperforms all other models for each targeted driver. For AE and AH, no such clear statement can be made; the results also depend on the targeted driver. However, except for the AH efficiency with the Ethernet driver as targeted CUE Server, the highest efficiency values are either obtained for the Fuzzing model or the SEU model.

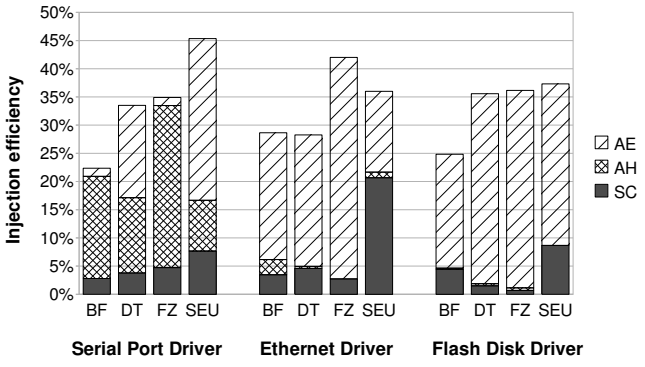

Figure 3: Comparison of the applied models's injection efficiencies for each targeted driver

An injection efficiency of 0 was obtained for AH failures of the Ethernet driver when exposed to FZ faults and for the flash disk driver when exposed to SEU faults, because no such failures were detected during the respective campaigns. The corresponding values are missing in the comparison of average execution times for the same reason.

Average Execution Time The results obtained for the execution time metric of injection runs with different experiment outcomes are listed in Table 5. The units are minutes and seconds (before and after the colon).

Experiments resulting in AH or SC failures tend to take more time than experiments resulting in AE failures or no failures as they usually imply failure detection by application or system response timeout. If a faster detection mechanism can be implemented for these failure modes, their execution

Table 5: Average execution times (m:s)

Target	Campaign	et_{NF}	et_{AE}	et_{AH}	et_{SC}
Serial Port Driver	c_{BF}	00:57	00:53	01:34	03:31
	c_{DT}	00:33	00:34	01:15	04:39
	c_{FZ}	01:06	01:14	01:55	03:37
	c_{SEU}	01:19	01:20	02:03	04:39
Ethernet Driver	c_{BF}	00:34	00:29	01:19	04:15
	c_{DT}	00:42	00:39	00:39	02:24
	c_{FZ}	00:51	00:48	–	03:12
	c_{SEU}	01:04	01:03	01:44	04:24
Flash Disk Driver	c_{BF}	00:43	00:43	01:25	03:41
	c_{DT}	00:36	00:37	01:16	03:53
	c_{FZ}	01:03	01:02	01:19	03:38
	c_{SEU}	02:01	01:57	–	05:09

Table 6: Implementation complexities

Model	ic_{DSI}	ic_{cyc}
BF	133	30
DT	635	222
FZ	272	58
SEU	259	48

time and overall evaluation efficiency will greatly improve. The detection of AE failures seems to add no complexity in terms of execution time; it is even less than the execution time for NF experiments in more than half of the cases. Note that we are considering SWIFI efficiency drivers for basic robustness evaluation needs. If the specific SWIFI objective is to comprehensively find robustness vulnerabilities, then longer execution times are also valid.

If the execution time metric is applied as proposed in Section 3, i.e. if models are compared with respect to the execution times for their lowest common failure modes, BF is the fastest model for the Ethernet driver and DT is fastest for the serial port and flash disk drivers. The least performant models in terms of injection efficiency perform best in terms of execution time and vice versa. This relationship suggests that the weakness of a model with respect to one metric may be compensated by its strength with respect to another one.

Implementation Complexity Table 6 provides an overview on the assessed implementation complexity of the models. From both, the DSI count and the McCabe complexity, DT is by far the most expensive model and BF is the least expensive model investigated. Being slightly less expensive, SEU's complexity hardly differs from FZ's.

We see that implementation complexity and execution time do not correlate in general. The two most expensive models in terms of execution time (SEU and FZ) are considerably cheap in terms of implementation complexity and the cheapest model in terms of execution time (DT) is the most expensive model in terms of implementation complexity. However, BF is cheap in terms of both execution time and implementation complexity. We do add the note that the use (and obtained rankings) of this metric is tied to the specific SWIFI tool being used. We do not assert completeness or the effort of setting up and configuring different SWIFI environments.

5. METRIC UTILITY

In the following we demonstrate the utility of our proposed metrics by comparing all applied fault models using the previously presented measurements. Table 7 gives a simplified overview of the results, assuming that all CUE failure modes and IUEs are of equal interest to the evaluator. The rankings were derived according to how often the models performed best among all investigated models. The details of these tendencies are highlighted and discussed onwards.

Parameter Bit Flips (BF) BF performs comparatively well at identifying service vulnerabilities that lead to AE failures in all targeted interfaces. However, it only detects vulnerabilities that are also detected by other models. The model performs best at identifying service vulnerabilities leading to AH and SC failures in the NDIS and (especially in the case of SC failures) CORE interfaces, where it also detects vulnerabilities that none of the other investigated models manages to detect. BF requires the least implementation effort, but performs worst in terms of injection efficiency. In terms of execution time it is a fairly cheap model to use.

Data Type Dependent Parameter Corruption (DT) Although it manages to identify a few unique AE vulnerabilities, DT provides average to poor coverage compared to the other investigated models. DT has the highest implementation complexity. This result is intuitive, since separate logic is required for each considered data type used for data exchange in the OS/driver (CUE/Server) interface, leading to a higher number of DSIs and a higher cyclomatic complexity. Overall, DT has a fair injection efficiency. It requires the least amount of time per injection run, meaning that it allows to perform more experiments than any other model in a given amount of time. Although the DSI count for the model is high, only small fractions of its code are actually executed for a parameter corruption of a specific data type. This is also indicated by the high cyclomatic complexity.

Parameter Fuzzing (FZ) The FZ model identifies a large number of vulnerabilities for all considered failure modes and most targeted interfaces. Except for the EXP, CORE and NDIS interfaces (the latter two only for SC failures), it outperforms all other models in terms of unique coverage, i.e. it detects large numbers of service vulnerabilities that remain undetected by other models. In terms of implementation complexity, the FZ model comes at almost twice the cost of the BF model, but is still less than half as costly as DT. FZ outperforms BF and DT in terms of injection efficiency, but is outperformed by these models in terms of execution time.

Single Event Upsets in Binaries (SEU) The SEU model performs very well at identifying vulnerabilities in the EXP interface but poorly at identifying vulnerabilities in any other interface. This result is intuitive as, according to the SEU model, faults are only injected to services belonging to this interface, i.e. into the the driver binary and thus into services exported by the driver. The implementation complexity for the SEU model is slightly less than for the FZ model. However, SWIFI mechanisms for supporting code mutations with flexible injection triggers require a considerable implementation effort. The fraction of code that is solely related to code mutation mechanisms accounts for a total of 1050 DSIs with a cumulated cyclomatic complexity of 262 in the applied SWIFI tool, in addition to about 50 lines of assembly code. However, if code mutations are natively supported by a chosen SWIFI framework, SEU is a modest model to implement. From an injection efficiency point of view SEU is, similarly to FZ, a very efficient model with a particular strength in provoking SC failures, but is

Table 7: In-practice observation on fault models

Model	Coverage	Implementation Complexity	Injection Efficiency	Execution Time
BF	★ ★ ★ ⋆	★ ★ ★ ★	★ ⋆ ⋆ ⋆	★ ★ ★ ⋆
DT	★ ★ ⋆ ⋆	★ ⋆ ⋆ ⋆	★ ★ ⋆ ⋆	★ ★ ★ ★
FZ	★ ★ ★ ★	★ ★ ⋆ ⋆	★ ★ ★ ⋆	★ ★ ⋆ ⋆
SEU	★ ⋆ ⋆ ⋆	★ ★ ⋆ ⋆	★ ★ ★ ⋆	★ ⋆ ⋆ ⋆

Legend: ⋆ ⋆ ⋆ ⋆ Poor, ★ ★ ★ ★ Good.

(also like FZ) expensive in terms of execution time.

Derived Metrics As already hinted at in the previous section, the comparative evaluation points out a trade-off between injection efficiency and execution time for three out of four models (DT, FZ, SEU). In order to quantify this trade-off more precisely for cross-model comparison, we derive a combined metric displaying the number of observed failures per unit of evaluation time. This metric should be of particular interest to evaluators in the software industry, where testing ends when resources (time, money) are depleted, since it enables the selection of a model that maximizes the number of internal fault activations for a given resource constraint.

The results displayed in Figure 4 show that, when we combine the metrics, the ranking for overall failure stimulation and also the failure mode distribution from Figure 3 change, in this case in favor of DT for AE failures. However, since we have observed weak coverage capabilities for the DT model, the presented combination of only two metrics alone must not be considered sufficiently expressive. The obtained coverage values suggest to spend at least some evaluation effort using the FZ and SEU models, since these two models identify large numbers of vulnerable services that remain undetected with any other model. If vulnerable services have been identified using these models, the degree and criticality of these vulnerabilities can further be quantified using the DT and BF models as time and available resources permit.

From Figure 4 we also see that a general preference of IUE injections over CUE Server injections due to efficiency considerations is not always justified. Bit flips that are injected into drivers instead of the OS/driver interface are apparently more efficient if both injection efficiency and execution time are considered.

Contributions and Limitations The metrics proposed in this paper are intended to guide fault model selections for robustness evaluations. The results of our case study demonstrate the applicability and validity of the presented approach and the discussion above stresses that each of the proposed metrics provides valuable insights. In any concrete evaluation, *better-than* and *worse-than* relations can be established for every individual metric by simply applying numerical *greater-than* and *less-than* operators respectively to benefit- and cost-oriented metrics, as we have shown throughout the presentation and discussion of our results. The individual measures of each metric are well suited for combinations, as all metrics provide ratio scale measures and can thus be quantified precisely. If, for instance, a model performs twice as good as another one in terms of injection efficiency and execution time, but only half as good in terms of implementation complexity and coverage, they are equally efficient for the robustness evaluation if all metrics are weighted equally.

Caveats: The presented approach has some limitations that evaluators should be aware of for objective usage. Our approach is restricted to evaluation methodologies that follow the system model outlined in Section 2.1. We assume explicit

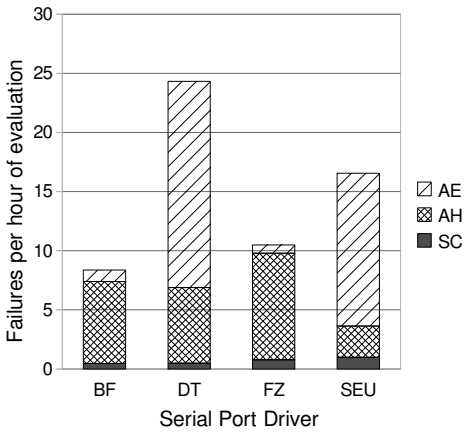

Figure 4: Cumulated failures of the serial port driver per hour of evaluation time

interactions of the CUE with its environment and thereby exclude certain classes of robustness evaluation approaches (such as emulations of hardware-induced software errors *in* the CUE) as well as certain classes of CUEs that heavily rely on implicit interactions with their runtime environments.

We demonstrate the applicability of our approach in a case study using a specific CUE and specific selections of CUE Servers and Clients. We cannot claim the universal validity of inferences drawn from the results of this case study. For instance, we have seen that the injection efficiency metric does not solely depend on the applied fault model, but also on the targeted CUE Server. For the serial port driver, the obtained injection efficiencies for AH failures are significantly larger than for the other drivers, independent from the applied fault model. It is also unclear whether the obtained results are comparable for other CUEs. However, as mentioned in the metrics introduction in Section 3, our main concern is to provide a method to comparatively evaluate contending fault models for a *given* evaluation attempt, i.e. a fixed CUE.

Up to now, the results of fault model evaluations according to the proposed approach can only be applied for feedback on an ongoing evaluation with injection campaign granularity. A campaign needs to be planned, executed, and evaluated before any feedback on the efficiency of the applied fault model can be derived and used for planning subsequent campaigns.

6. CONCLUSION AND FUTURE WORK

Considering the importance of robustness testing approaches for COTS software components, this paper addresses the problem of robustness testing sufficiency in absence of both a single application scenario and source code access. To this end it presents an approach to comparatively evaluate the efficiency of different fault models applied in SWIFI-based

robustness evaluations in order to guide their selection.

Our proposed evaluation metrics cover both cost and benefit aspects of fault model implementation and usage. Except for the implementation complexity of fault models, for which existing assessment tools are referenced, the proposed metrics do not require additional measurements beyond those required for actual robustness evaluations. We have demonstrated our approach in a robustness evaluation of the Windows CE 4.2 kernel, comparing the efficiency of four commonly applied fault models as a case study.

We are currently attempting to transfer our fault model evaluation approach to different software platforms (other OS kernels as well as regular, user-space software) to investigate the proposed quantifiers's performance in cross-platform model comparisons. Currently, we are investigating which modifications of the metric definitions are required to increase the adaptivity of robustness evaluations. For this purpose we consider increasing the granularity of the feedback provided by our metrics, (i.e. we aim at providing feedback on a model's efficiency per injection run instead of per injection campaign and make this information directly accessible to the SWIFI framework), so that the robustness evaluation efficiency can be optimized on-the-fly. We are also considering introducing further feedback loops into the generic SWIFI-based robustness evaluation approach that reflect the influences of other factors, such as the applied workloads and targeted CUE Servers.

Acknowledgement: Research supported in part by TUD CASED and Microsoft.

7. REFERENCES

[1] A. Albinet, J. Arlat, and J. C. Fabre. Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel. In *Proc. DSN*, pages 867–876, 2004.

[2] J. Arlat, J. C. Fabre, and M. Rodriguez. Dependability of COTS Microkernel-based Systems. *IEEE Trans. Comput.*, 51(2):138–163, 2002.

[3] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secure Comput.*, 1(1):11–33, 2004.

[4] Boehm, Abts, Clark, Horowitz, Brown, Reifer, Chulani, Madachy, and Steece. *Software Cost Estimation with Cocomo II with CD-ROM*. Prentice Hall PTR, 2000.

[5] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1981.

[6] J. Carreira, H. Madeira, and J. G. Silva. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. *IEEE Trans. Softw. Eng.*, 24(2):125–136, 1998.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proc. SOSP*, pages 73–88. ACM, 2001.

[8] J. Duraes and H. Madeira. Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation. In *Proc. PRDC*, pages 201–209, 2002.

[9] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP Kernel Crash Analysis. In *Proc. LISA*, pages 12–22, 2006.

[10] H. Hecht. Rare Conditions – An Important Cause of Failures. In *Proc. COMPASS*, pages 81–85, 1993.

[11] J. C. Huang. An Approach to Program Testing. *ACM Comput. Surv.*, 7(3):113–128, 1975.

[12] IEEE. Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, page 1, 1990.

[13] ISO/IEC 19761:2003. *Software Engineering – COSMIC-FFP – A Functional Size Measurement Method*. 2003.

[14] ISO/IEC 20926:2003. *Software Engineering – IFPUG 4.1 Unadjusted Functional Size Measurement Method – Counting Practices Manual*. 2003.

[15] A. Johansson. *Robustness Evaluation of Operating Systems*. PhD thesis, TU Darmstadt, 2008.

[16] A. Johansson and N. Suri. Error Propagation Profiling of Operating Systems. In *Proc. DSN*, pages 86–95, 2005.

[17] A. Johansson, N. Suri, and B. Murphy. On the Impact of Injection Triggers for OS Robustness Evaluation. In N. Suri, editor, *Proc. ISSRE*, pages 127–126, 2007.

[18] A. Johansson, N. Suri, and B. Murphy. On the Selection of Error Model(s) for OS Robustness Evaluation. In *Proc. DSN*, pages 502–511, 2007.

[19] A. Kalakech, K. Kanoun, Y. Crouzet, and J. Arlat. Benchmarking the Dependability of Windows NT4, 2000 and XP. In *Proc. DSN*, pages 681–686, 2004.

[20] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. Comparing Operating Systems using Robustness Benchmarks. In *Proc. SRDS*, pages 72–79, 1997.

[21] R. R. Lutz and I. C. Mikulski. Operational Anomalies as a Cause of Safety-Critical Requirements Evolution. *Journal of Systems and Software*, 65(2):155 – 161, 2003.

[22] T. McCabe. A Complexity Measure. *IEEE Trans. Softw. Eng.*, SE-2(4):308–320, 1976.

[23] M. Mendonça and N. Neves. Robustness Testing of the Windows DDK. In *Proc. DSN*, pages 554–564, 2007.

[24] 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 R. Barbosa, editor, *Proc. EDCC '06*, pages 53–64, 2006.

[25] MSDN. coredll Module. `http://msdn.microsoft.com/en-us/library/aa448387.aspx`.

[26] MSDN. Implementing CEDDK.dll. `http://msdn.microsoft.com/en-us/library/ms898217.aspx`.

[27] MSDN. Network Driver Functions. `http://msdn.microsoft.com/en-us/library/ms895631.aspx`.

[28] D. Simpson. Windows XP Embedded with Service Pack 1 Reliability, January 2003.

[29] C. Software. SourceMonitor Version 2.5. `http://www.campwoodsw.com/sourcemonitor.html`.

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

[31] E. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting how badly "good" Software can behave. *IEEE Softw.*, 14(4):73–83, 1997.

[32] D. A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.