Corrected Version

On the Utility of Higher Order Fault Models for Fault Injections

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

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

Dipl.-Inform. Stefan Winter

aus Offenbach am Main

Referenten:
Prof. Neeraj Suri, Ph.D.
Prof. Domenico Cotroneo, Ph.D.

Datum der Einreichung: 24. März 2015
Datum der mündlichen Prüfung: 7. Mai 2015

Darmstadt 2015
D17

Changes in the Corrected Version

The purpose of this document is to correct errors that were introduced in the publication process of my Ph.D. thesis. Due to unresolved and incorrectly resolved references, the version of my thesis that has been published under `http://tuprints.ulb.tu-darmstadt.de/4559/` differs from the version that I had originally submitted and that has been accepted and evaluated by the examination committee. The corresponding corrections have been typeset in red color to facilitate their identification in the text. In addition to these corrections, the only changes that have been introduced, and that are also highlighted, are the designation of the document as a corrected version on the title page and the explanation on this page.

Abstract

Fault injection (FI) is an experimental technique to assess the robustness of software by deliberately exposing it to faulty inputs and stressful environmental conditions specified by fault models. As computing hardware is becoming increasingly parallel, software execution is becoming increasingly concurrent. Moreover, to exploit the potential of increasingly parallel and interconnected computing systems, the complexity of the software stack, comprising operating systems, drivers, protocol stacks, middleware and distributed applications, also grows. As a consequence, the fault models classically used for in-lab robustness assessments no longer match the reality of fault conditions that modern systems are exposed to.

In my thesis I account for this development by proposing the construction of higher order fault models from classical models by different means of composition. To demonstrate the effectiveness of such higher order models, I define a set of four comparative fault model efficiency metrics and apply them for both classical and higher order models. The results show that higher order models identify robustness issues that classical models fail to detect, which supports their adoption in the assessment of modern software systems. While higher order models can be implemented with moderate effort, they result in a combinatorial explosion of *possible* fault conditions to test with and, more severely, they introduce an ambiguity to experimental result interpretation that results in an increased number of *required* tests to maintain the expressiveness of assessment results.

To mitigate the overhead that the adoption of higher order fault models entails, I propose to increase the experiment throughput by concurrently executing experiments on parallel hardware. The results show that such parallelization yields the desired throughput improvements, but also that care has to be taken in order not to threaten the metrological compatibility of the results. To account for resource contention in concurrent experiment executions that can lead to result deviations, I present a calibration approach that provides timeout values for the safe configuration of hang failure detectors.

The experimental environment to conduct concurrent experiments is based on a generic FI framework developed in the context of my thesis that is highly adaptable to a variety of different target systems and that has been released under an open source license.

Kurzfassung

Fehlerinjektion (FI) ist eine experimentelle Methode zur Bewertung von Softwarerobustheit, bei der Software gezielt fehlerhaften Eingaben und widrigen Betriebsbedingungen, gemäß spezifizierter Fehlermodelle, ausgesetzt wird. Mit zunehmend paralleler Hardware wird die Ausführung von Software zunehmend nebenläufig. Mit der Bestrebung das Potenzial dieser zunehmend parallelen und vernetzten Systeme auszuschöpfen, wächst die Komplexität des gesamten Softwarestacks über hardwarenahe Schichten wie Betriebsysteme, Treiber und Netzwerkprotokolle bis hin zu Middleware und verteilten Anwendungen. In der Folge ändern sich die Fehlermanifestationen, denen Softwarekomponenten in dieser geänderten Umgebung ausgesetzt sind, in einer Weise, die durch jeher zur Anwendung kommenden Fehlermodelle nicht abgedeckt wird.

Mit dem Ziel dieser Entwicklung entgegenzuwirken, entwerfe ich im Rahmen meiner Arbeit Fehlermodelle höherer Ordnung durch Kombination bestehender Fehlermodelle. Um die Nutzbarkeit dieser Modelle vergleichend bewerten zu können, entwickle ich vier Effizienzmetriken, anhand derer ich Modelle höherer Ordnung mit herkömmlichen Modellen vergleiche. Das Ergebnis der Studie zeigt, dass Modelle höherer Ordnung Robustheitsschwächen aufdecken, die bei der Verwendung herkömmlicher Modelle unentdeckt bleiben, was ihre Anwendung bei der Robustheitsbewertung moderner Softwaresysteme nahelegt. Wenngleich sich der Entwicklungsaufwand für Modelle höherer Ordnung in Grenzen hält, führt ihre Anwendung zu einer kombinatorischen Explosion der Anzahl *möglicher* Fehlerzustände und, gravierender, zu einer Mehrdeutigkeit der experimentellen Ergebnisse, die eine höhere Anzahl von Tests zwingend *erfordert* um die Aussagekraft der Bewertung zu erhalten.

Um den so mit der Verwendung von Modellen höherer Ordnung einhergehenden zeitlichen Mehraufwand bei der Bewertung zu mindern, verfolge ich einen Ansatz zur nebenläufigen Ausführung von Experimenten auf paralleler Hardware. Obwohl der Ansatz die gewünschte Durchsatzsteigerung erzielt, zeigen die Ergebnisse auch, dass er bei unachtsamer Anwendung die metrologische Kompatibilität der Ergebnisse kompromittiert. Um den ursächlichen Ressourcenkonflikten entgegenzuwirken, stelle ich einen Ansatz zur Kalibrierung von Zeitbeschränkungen zur Fehlerdetektion vor, der diese Probleme vermeidet.

Die Umgebung zur Durchführung nebenläufiger Experimente basiert auf einem im Rahmen meiner Arbeit entwickelten generischen Framework für Fehlerinjektionen, das für eine Vielzahl von Zielsystemen anwendbar ist und unter einer Open-Source-Lizenz veröffentlicht wurde.

Acknowledgments

First and foremost, I would like to thank my advisor Neeraj Suri, without whom I would have neither started a PhD program nor would I have had the persistence to actually finish it. I have seen many others, who were not as lucky as me, departing from this path prematurely. Thank you for giving me room to grow and for supporting even the crazy ideas. (Hierarchical microkernels, anyone?) I am also very grateful to Domenico Cotroneo for accepting to be my external reviewer and for what I learned from him through his publications and the short but intensive discussions we had in Darmstadt.

I would like to thank my co-authors, with whom I published, for spending long nights fixing my bugs and tweaking implementations and write-up. Thanks Thorsten, Olli, Roberto and Anna, Dinu, Brendan, Hyper-Michi, Benny, Tasuku, Habib, Andréas, Andreas, Daniel, Hamza, Majid, Qixin, Feng, Martin, Paul, and Suman. Thanks, Andi and Hatem, for proof-reading. Thanks to some old DEEDSians: Peter, Matthias, Dan, Marco, Piotr. I miss the Stammtisch evenings! Thanks, Davide, for making me running round in circles. I would also like to thank my HiWis Alex and Saeed and my former HiWis Holger and Martin for all their help exploring the mysteries of technology. During the six years I have spent at DEEDS, I had the great pleasure to work with and learn from many extraordinarily skilled and friendly people, far too many to fit on this page. You know who you are!

I am deeply indebted to the free and open source software movement. Not only is the research in this thesis, its write-up, and type-setting to a large degree based on FOSS. I have learned much of what I know today from reading others' code and I hope that others will similarly benefit from my work. Thank you for openly sharing your knowledge and thoughts this way!

Finally, I would like to thank my family Maria, Helmut, Christian, Christa, Ernst, and my "second family" Monika, Andreas, Hendrik, for all their support. Last, and most importantly: Thank you, Maria! Without your love, support, and patience this would have never worked.

Contents

1 Introduction

Computer systems increasingly permeate our daily lives. According to estimates, the number of mobile computing devices exceeded the world population for the first time in 2014 and a continuing sharp increase by more than 50 % is expected by 2019 [Cis15]. At the same time, these systems become highly interconnected. Until 2019 the fraction of *networked* mobile devices for the implementation of "smart" applications is estimated to increase (from 7 % in 2014) to 28 %, i.e., 3.2 billion networked devices, by 2019. As these numbers only cover estimates for mobile consumer electronics and neither include embedded controllers, which have become the main driver for innovation in automobiles and industrial applications, nor the powerful processing and storage infrastructure, which enables computationally demanding applications for the growing Internet of Things, they must be considered a gross underestimation.

The steeply increasing pervasiveness of information technology (IT), in large parts, is based on the expectation that we can depend on these systems to reliably provide the services expected from them, where the consequences of failures range from the nuisances of, for instance, calendar bugs [App15] to outages of critical infrastructures (e.g., [Gro+04]). Achieving dependable operation is challenging in a rapidly changing ecosystem of many, complex, and interconnected systems that heavily rely on software. The discrete nature of software renders classical approaches to reliability engineering of mechanical and other continuous systems inapplicable, as these heavily exploit continuity assumptions to statistically extrapolate universal reliability from a limited number of samples (*tests*). These continuity assumptions rarely hold for discrete software-based systems.

Moreover, to utilize complex and interconnected hardware systems and to implement feature-rich smart applications, software complexity is massively increasing. A popular example are operating systems[1] (OSs), as they are directly affected by changes in the underlying system hardware and provide the basic communication primitives for distributed applications. If the defect rate for these systems remains constant, their increase in size implies an increase in the absolute number of defects that need to be found and eliminated.

Furthermore, even if confidence in a software-based system's *correctness* is established (which is a hard problem by itself), the system can still behave undesirably in cases of unforeseen or unintended interactions with other systems or its operational environment, exemplified by effects labeled as *emergent (mis)behavior* [Mog06]

[1] The Linux kernel, for instance, on which the Android mobile operating system (OS) is based, has grown from around 6 million lines of code in version 2.6.0 (released 2003) to more than 14.5 million lines of code in version 3.0 (released 2011), from which only 20 % of the 2.6.0 source lines had remained unchanged [PTS+11].

or *transmogrification* [Dri10]. The ability of systems to gracefully handle such unforeseen, and often unforeseeable, interactions and to not fail in ways that threaten human lives, cause environmental damage, or massive financial loss, is termed *robustness*. *Fault injection* (FI) is an experimental technique to assess and improve system robustness by deliberately exposing systems to invalid inputs and other stressful operational conditions. The major challenge in FI-based robustness assessments is the choice of the right *fault model*, i.e., the specification of what "unforeseeable" stressful conditions are.

The problem of fault model selection has been an active field of research and a large body of work on the topic exists, covering perturbations originating from both hardware and software and leading to sophisticated fault taxonomies, as discussed in Chapter 2 of this thesis. As most of the existing work stems from a time when parallel and distributed computing was still uncommon in commodity systems, the utilized fault models and taxonomies are based on an IT ecosystem that became rapidly outdated over the last decade.

On this general background, this thesis

1. proposes to utilize *higher order fault models* as combinations of existing fault models to account for the impact of increasing software complexity and its execution on parallel hardware,
2. demonstrates the value of these fault models in a comparison with classical models, and
3. proposes a solution strategy to cope with the higher complexity of robustness assessments, which they imply.

The experimental studies conducted to validate these contributions focus on robustness assessments of complex mobile OSs, which constitute a relevant target as outlined above. The remainder of this chapter introduces basic terminology, details the problem statement, and lays out the research questions that have been driving the work presented in this thesis.

1.1 Basic Concepts and Terminology

Many terms used in this thesis have a technical denotation in addition to their common meaning in ordinary language. Moreover, most of them have *multiple* definitions that differ for different communities. The terminology in this thesis is mostly adopted from the dependability and fault-tolerance research community, mainly in line with Avizienis et al. [ALR+04].

1.1.1 Failures, Errors, Faults: The Threats to Dependability

The subject of study in this thesis are *software systems* and I use the term *system* synonymously, unless explicitly stated elsewise. Systems interact with other systems

(hardware or software) in their environment, from which they are distinguishable by their *system boundaries*. A system's intended behavior is specified as a function of system input and is, hence, referred to as *functional specification*. The *input* is a sequence of system-external stimuli that trigger its behavior. The *behavior* of a system results from the implementation of its functional specification and is described by a sequence of system *states* that result from exposing the system to an input. The fraction of a system's state at any point in time that is observable from outside the system boundary is called the *external state* of the system, whereas the externally not observable fraction is called its *internal state*. The sequence of external states resulting from exposure to an input is referred to as the system's *service*. Exposure to inputs and observation of external states are provided by system access points collectively referred to as the system's *interface*. As these access points enable service reception, they are also referred to as services in this thesis.

A provided service is *correct* if the system behaves according to its specified function. Any deviation from correct service constitutes a system *failure*. Being a property of system services, failures are part of the external state. The cause of a failure is a system's deviation from its correct internal state, termed *error*. The transformation of an error manifestation within the system's internal state and its ultimate progression to the system's external state are referred to as error *propagation*. The cause of an error is the *activation* of a *fault*. Figure 1.1 illustrates the causation chain implied by the provided definitions.

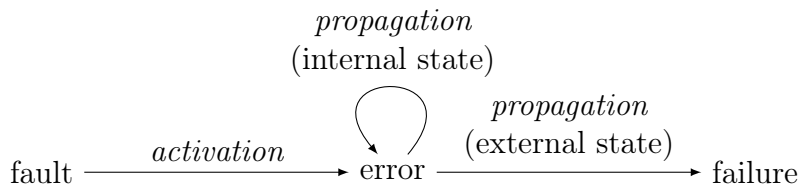

Figure 1.1: The causal relation of faults, errors, and failures

Faults are divided into two classes, *internal faults* and *external faults*, depending on the circumstances of their activation. Avizienis et al. [ALR+04] note the difference of internal and external faults, but their distinction is partially implicit. They state:

> *Faults can be internal or external of a system. The prior presence of a vulnerability, i.e., an internal fault that enables an external fault to harm the system, is necessary for an external fault to cause an error and possibly subsequent failure(s).*

By this definition, an internal fault is a necessary prerequisite for an external fault, but an external fault is not stated as a necessary prerequisite for an internal fault. This means that system external events that can lead to the activation of internal faults fall into two categories, (1) external faults and (2) "benign" input as *trigger conditions* of internal faults. The distinction is not explicitly stated by the authors, but follows

from the definition of what a fault is; a *potential cause*[2] of failure. Considering that a failure is the deviation of a system's service from its specified function, any input that is covered by the domain of that function and that leads to system service deviation from what the function specifies must be the sole consequence of an internal fault. The corresponding input must be considered benign, as it is explicitly covered by the functional specification. Therefore, I refer to external faults as system external conditions that are *not* covered by the domain of the corresponding system function, i.e., conditions for which the system behavior is unspecified.

This understanding is not without problems: If the behavior for a given input is unspecified, what constitutes a failure, i.e., a deviation from *specified* behavior, for that input? The general interpretation of unspecified behavior is that *any* behavior is valid (cf. "nasal demons" in ANSI C compilers [Ray04]), i.e., not a failure, although it comprises arbitrarily adverse and dangerous behavior. As this is clearly undesirable, I follow a different interpretation: Undefined behavior is considered a failure, if its manifestation in the sequence of external states that constitute the system service matches a *failure mode* specified by the user (a human or other system). This opens the possibility to state *negative requirements*, i.e., what a system should *not* do, as I elaborate in Section 1.1.2.

The alternative understanding of the relation between internal and external faults is that *every* fault necessarily has an external and internal fraction. In that case, where external faults are a necessary precondition for failure, any input eventually causing a failure is an external fault. This implies that there is no distinction between benign and faulty inputs. Hence, the concept of an external fault does not *require* the existence of unspecified behavior and external faults would also exist for systems specified by total functions, for which there (theoretically) is no unspecified behavior. This understanding has a different problem, however: If a fault always consists of internal and external fault fractions, a problem of attribution arises. If the input and the system implementation are equally necessary conditions of any observed failure, a change of the implementation and a change of the specification would be equally valid remedies and lead to the awkward situation where functional specifications would no longer be driven by required functionality, but by implementations. Hence, I follow the first interpretation of internal and external fault notions, where only internal faults are considered a necessary precondition for failures and external faults are unspecified inputs that trigger undesired behavior.

Having a clear understanding of failures and their causation, dependability can be defined as their absence: A system is *dependable* if it fails less frequently and less severely than is acceptable. As acceptability is subject to a system's application context and requirements, dependability may change if the system is applied in different contexts, by different users, or if the application context evolves over time. The absence of failures according to the definitions in this thesis is achieved by the combination of two properties, correctness and robustness, as discussed in the following.

[2] a necessary, but insufficient precondition

1.1.2 The Interplay of Correctness and Robustness

As stated in Section 1.1.1, I distinguish between the specified functionality of a system and its implementation. The consistency of a system's behavior with its specification is termed *correctness*. As detailed in the discourse on the distinction between external and internal faults, the provision of correctness is no sufficient precondition for dependability if the functional specification (implicitly or explicitly) includes partial functions. In these cases, the implementation provides unspecified behavior, of which a variable subset is dangerous, depending on the (differing) behavior semantics in different system environments. The absence of dangerous behavior upon undefined input is termed *robustness*. As the classification of behavior as dangerous depends on the system's application context, it is safe to consider all unspecified behavior as a (potential) *robustness issue*. Whether a robustness issue constitutes a *robustness violation* or not, is determined by the severity of failure modes. Failure modes are stated as *negative requirements* [Mus96]. In contrast to Musa [Mus96] and Koopman and DeVale [KD00], I avoid the term *severity class* for negative requirements specifications, as the severity of failure modes varies with application scenarios, whereas the behavioral description of failure modes is invariant. The scenario-agnostic description of failure modes allows to conduct robustness assessments of reusable and off-the-shelf systems, for which operational conditions vary and cannot be accurately predicted.

Dependability requires both correctness and robustness. The Venn diagram in Figure 1.2, which is inspired by a figure from Whittaker and Thompson [WT04], depicts the relation between specification and implementation and the dependability threats resulting from their deviation. Specification and implementation are depicted as sets of two-dimensional tuples that relate system input and system behavior. The system's specification relates specified input to specified behavior and the system's implementation relates possible input to implemented behavior. Note that what is labeled as correctness and robustness issues indicate internal and external faults. Correctness issues result if specified behavior is not implemented, whereas robustness issues result if a systems exhibit unspecified behavior. Both indicate internal faults and robustness issues additionally indicate external faults, i.e., input that triggers unspecified behavior, which leads to the violation of a negative requirement. Internal and external faults are necessary, but not sufficient preconditions for failures as per their definitions.

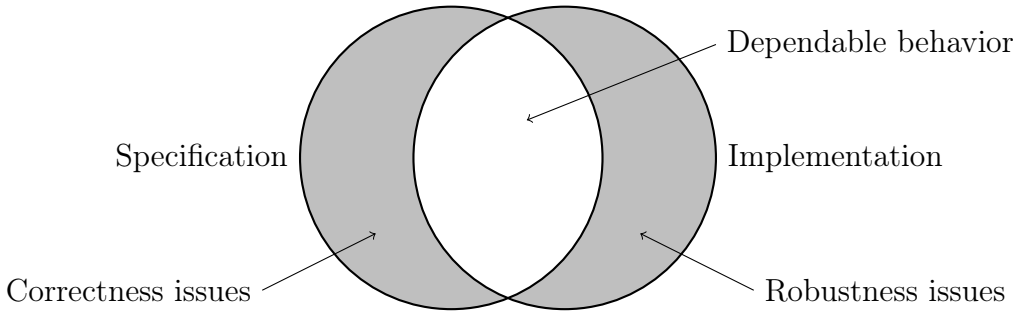

Figure 1.2: Specification, implementation, and the threats to dependable behavior

1.1.3 Assessing (the Absence of) Robustness: Fault Injections

Fault injection (FI) is a technique to assess and improve system robustness (or, in the context of this thesis, the lack thereof) by deliberately exposing systems to undefined inputs (called *fault load*), including stressful operational conditions in the system's environment. The system's response is monitored via failure detectors that are derived from the stated negative requirements and indicate if the system behaves in a potentially dangerous manner. If a detector indicates a failure in response to an injection, the applied fault load constitutes an external fault that has triggered an internal fault within the system under test (SUT). The corresponding information can be used to harden the system, either by fixing the internal fault or by preventing the external fault from affecting the system, e.g., by adding subsystems that filter the faulty input (e.g., by so-called fault containment wrappers). The major challenge in FI-based robustness assessments is the choice of the right *fault model*, i.e., a systematic description of how the fault load is chosen. Existing strategies fall in two broad classes, depending on the intention of their application, and the distinction is analog to that of operational and debug (correctness) testing [FHL+98]: While the purpose of operational testing is to provide reliability estimates for a given input distribution, the purpose of debug testing is to find internal faults.

Operational FI: FI has been advocated as a tool for dependability benchmarking (e.g., [TIJ96; MK01; KM02; VM03; DVM04; KKC+04; KCK+05]). The purpose is to assess the dependability of a system for a known application scenario and environment. Since the application scenario and environment are known, corresponding statistically representative distributions of external faults can be derived and applied in the tests [DM03a; DM06; NCD+10; NCD+13a].

Debug FI: FI can be used to identify robustness issues and apply counter measures. In contrast to operational FI, little guidance can be given for the selection of "good" fault models. Fault models that find higher numbers of robustness issues are better than those that find fewer, but no absolute measure of their quality can be given, as the total number of robustness issues is unknown (and may differ for different application scenarios). Similar to debug (correctness) testing, which is similarly lacking an absolute effectiveness measure for test case selections, test inputs are usually chosen based on experience. I provide a detailed discussion of the existing approaches to fault load generation for debug FI in Chapters 2 and 4.

A common assumption that all of the classically applied fault models share is the restriction to external faults that only manifest in a single location in the SUT's environment at a time. The hypothesized reason for this assumption is twofold. First, if an FI experiment reveals a robustness issue, identifying the causation chain that relates the injected external fault with the observed failure is usually a complex task that involves a great deal of manual investigation and the more complex the fault load is, the more complex becomes this analysis. Keeping the fault models simple

facilitates the isolation of the faults' effects. Second, most documented fault models have been used and analyzed in studies before 2005, when the commodity computing landscape began to change substantially, as discussed in the next section. While such old system models still apply for today's embedded controllers, they will be outdated soon, similar to the development of mobile devices, where high performance multicore CPUs have quickly become the norm.

1.2 Evolution of Computing Systems: The Need for New Fault Models

As existing fault models for debug FI derive from research more than a decade ago, they do not reflect recent changes in hardware architectures that had, and continue to have, a high impact on the performance, but also scale and complexity, of software systems. I discuss two significant changes in this section and detail their expected influence on fault models for software system assessments.

1.2.1 Compositional Faults

While computing systems have been subject to significant changes at high pace ever since, few have had a similar impact on software systems as the shift from single- to multi-core computer architectures in the desktop market starting from 2005. Although parallel computing had steadily evolved since the 80s, when the advent of wide and local area networking technology and the availability of powerful microprocessors at reasonable cost fostered the development of distributed systems, the computational power of mainstream applications increased mostly with CPU clock rates for the next 20 years. By executing instructions faster, higher performance was achieved without necessitating software adjustments. This changed around 2005, when the cost for sufficiently powerful cooling systems to accommodate the high heat dissipation resulting from high clock rates exceeded the cost for replicating processing units.

To exploit the computational power of parallel computing architectures, existing software had to undergo, and is still undergoing, massive changes. In the following years, the steady improvement in broadband networking technology added to the evolution of an inherently parallel computing landscape and necessitated additional software infrastructure, such as protocol stacks and complex middleware, for communication and synchronization among asynchronous, parallel, and networked systems. The simultaneous execution of software systems across physically replicated computing hardware enables the simultaneous activation of faults, propagation of errors, and occurrence of failures in different parts of the system. The possible interaction of several faults in *compositional faults* are not addressed by classical fault models.

1.2.2 High Fault Densities

A second trend in hardware development is the increase of transistor and wire densities. As this development is known to cause increased soft error rates in MOS-based processors [SKK+02], it has fostered research on suitable counter measures in recent years [Bor05]. As the continuous reduction in transistor size is expected to reach its physical limits in the near future and alternative technologies are reported to yield even higher soft error rates [WNL+08], the development of software-implemented hardware fault tolerance (e.g., [KNS10; FGA+10]) and probabilistic/approximate computation paradigms (e.g., [CMR13]) have become an active field of research.

These developments of computer hardware necessitate software of increasing complexity to (1) manage hardware resources (e.g., device drivers and OSs) and (2) exploit the hardware's potential in sophisticated applications. In addition to this hardware-induced growth of software complexity, the demand for software-based control in an wide range of applications (advanced driver assistance, electronic stock trading, etc.) entails a feature-driven growth. A premium car in 2009 contained around 100 million lines of code [Cha09]. With an optimistically assumed rate of 0.1 residual defects per thousand lines of code (the exceptional defect rate achieved for the space shuttle control software [Ash95]), the software of such a premium car contains 10 000 defects. Given the numbers commonly reported for less critical and expensive systems than space shuttles, a more realistic estimate (for a safety-critical system) would be one or two magnitudes higher.

In summary, the combination of increased failure probabilities for individual components and their composition in increasingly parallel systems entails an increased risk of simultaneously occurring subsystem failures. This raises the question if the classical fault models for debug FI, which only consider effects of a single fault per experiment, are suitable for assessing contemporary and future systems.

1.3 Research Questions and Contributions

The question of fault model adequacy is not easily answered. While studies of fault distributions in existing software provide guidance for operational FI [DM03a; DM06; NCD+10; NCD+13a; LNW+14], the value of debug FI fault models lies in their ability to identify robustness issues. Although few corresponding metrics exist, none of them is suitable for the comparative assessment across diverse fault models as intended in this thesis. Consequently, a suitable set of metrics has to be found before the question of fault model adequacy can even be phrased in technical terms.

Research Question (RQ1): How can the effectiveness (and efficiency) of fault models for debug FI be assessed?

In order to assess and compare the suitability of classical fault models and the new compositional fault models proposed in this thesis, corresponding metrics need to be established. As stated in Section 1.1.3, measuring the utility of fault models for debug

FI is challenging. Existing metrics mostly focus on failure rates without considering failure root causes, whose identification is the main driver behind debug FI.

Contribution (C1): A set of metrics to quantify fault model effectiveness and cost

Inspired by structural coverage metrics used in correctness testing, I define a set of metrics to assess the effectiveness of fault models for debug FI and the overheads their usage implies in Chapter 4. I validate the metrics in an extensive robustness test campaign of the Windows CE kernel using four classical fault models to perturb interactions of the kernel and three widely used device drivers. The results have been presented at ICSE 2011 [WSS+11].

Research Question (RQ2): Is debug FI with complex fault conditions more effective than the classical fault models?

The question is a direct consequence of the development described in Section 1.2. Building on (C1) the effectiveness of debug FI can be quantified and a comparative study can be conducted.

Contribution (C2): A taxonomy for fault coincidence and framework for fault model combination

Based on the observed inconsistency between the system models that classical fault models were derived from and the systems found in wide spread use today, I develop

- a taxonomy for fault coincidence,
- a systematic approach to derive compositional *higher order fault models* from classical fault models, and
- a set of three higher order fault models to demonstrate the approach's utility in a case study.

In Chapter 5 of this thesis, which is based on results presented at DSN 2013 [WTS+13], I compare the effectiveness of these higher order fault models against the effectiveness of classical models. The results show that debug FI using higher order fault models yields significantly different results, but also exacerbates the *test case selection problem* due to a combinatorial explosion in the number of relevant faults to test with. This observation leads to the next research question.

Research Question (RQ3): Can parallel hardware help to increase the efficiency of debug FI to mitigate the overhead that higher order fault models entail?

In Section 1.2, I have argued that parallel hardware entails software complexity, which in turn leads to more complex fault conditions. By (C2), I have shown that these

conditions identify different robustness issues than classical models and, consequently, should be considered in debug FI tests. Unfortunately, higher order fault models also imply higher numbers of FI tests that can be conducted. As the root cause of the observed problem, i.e., parallel hardware, has the purpose of improving computing performance, I investigate if it can be exploited to mitigate the problem of higher experiment volumes by speeding up FI tests in parallel executions.

Contribution (C3): A framework for parallel FI experiments

In Chapter 6, which is based on an article accepted for presentation at ICSE 2015 [WSN+15], I propose PAIN (PArallel fault INjections), a framework for conducting FI experiments that exploits parallel hardware. Using PAIN, I conduct a set of higher order FI experiments on the Android mobile OS and assess both experiment throughput and the obtained *metrological compatibility*, i.e., result equivalence, of parallel and sequential test executions. The results show that significant throughput improvements are feasible without loss of result accuracy, when the implementation and configuration of failure detectors are adjusted. A corresponding approach for time-out adjustments, which are a critical factor in failure detector implementations, is proposed.

Contribution (C4): A generic FI tool

PAIN is based on a generic fault injection tool called GRINDER, developed to overcome the difficulty of adapting existing FI tools to new SUTs experienced during the work on this thesis. GRINDER defines an abstract interface for controlling arbitrary SUTs. I discuss GRINDER in Chapter 7, which is based on an article accepted for presentation at AST 2015 [WPS+15].

1.4 Publications

The following previously published material has been, in parts verbatim, included in this thesis.

- S. Winter, C. Sârbu, N. Suri, and B. Murphy. "The Impact of Fault Models on Software Robustness Evaluations". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. 2011, pp. 51–60
- S. Winter, M. Tretter, B. Sattler, and N. Suri. "simFI: From single to simultaneous software fault injections". In: *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2013, pp. 1–12
- S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo. "No PAIN, No Gain? The utility of PArallel fault INjections". In: *Proceedings of the International Conference on Software Engineering (ICSE) (to appear)*. 2015

- S. Winter, T. Piper, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo. "GRINDER: On Reusability of Fault Injection Tools". In: *Proceedings of the IEEE/ACM International Workshop on Automation of Software Test (AST) (to appear)*. 2015

The following publications are related to different aspects covered in this thesis, but have not been included.

- S. Winter, C. Sarbu, A. Johansson, and N. Suri. "Impact of Error Models on OS Robustness Evaluations". In: *Supplemental Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. 2009
- C. Sarbu, S. Winter, N. Nagappan, and N. Suri. "OS Driver Test Effort Reduction via Operational Profiling". In: *Supplemental Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. 2009
- S. Winter, D. Germanus, H. Ghani, T. Piper, A. Khelil, and N. Suri. "Trustworthiness Evaluation Of Critical Information Infrastructures". In: *Critical Infrastructure Security: Assessment, Prevention, Detection, Response*. Ed. by F. Flammini. WIT Press, 2012
- T. Piper, S. Winter, P. Manns, and N. Suri. "Instrumenting AUTOSAR for dependability assessment: A guidance framework". In: *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2012, pp. 1–12
- A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri. "An Empirical Study of Injected Versus Actual Interface Errors". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. 2014, pp. 397–408

2 Related Work

In line with the research questions and contributions stated in Section 1.3, related work falls in three main categories:

1. Work on FI and test efficiency metrics (Chapter 4)
2. Work on higher order fault models and injections (Chapter 5)
3. Work on test parallelization (Chapter 6)

The related work in these three areas is discussed in the corresponding chapters.

In this chapter I review work that builds the fundamental context for the work presented in this thesis, i.e., work on debug FI for robustness testing and assessments, FI approaches, and fault models.

2.1 The Origins of FI

FI has first been applied to assess the fault-tolerance of digital circuit designs in simulations [HS67; Arm72; UBW72] and was, therefore, later termed *simulation-based FI* [HTI97]. Consequently, the corresponding fault models comprised effects that were deemed representative for failing logic elements, in particular *stuck-at* logical 0 or 1.

FI was also used to test the resilience of computer systems against non-systematic hardware errors resulting, for instance, from electromagnetic interference [Bel89] or heavy-ion radiation [GKT89]. As the precise effects of these physical phenomena on the system implementation were difficult to predict, they could not be simulated in combinational logic designs. The corresponding approaches were termed *physical FI* [Avi78] or *hardware FI without contact* [HTI97].

As computer systems became more software-centric, FI was used to determine the impact of hardware faults on software execution, e.g., to develop mechanisms for error detection or recovery. For software assessments, simulation-based FI was not suitable, as the used hardware simulators did not provide any support for software execution on the simulated circuitry. Physical FI also suffers from a number of practical drawbacks for software assessments. The effects of heavy-ion radiation and other physical injections are difficult to control and repeatability of observed effects is hard to achieve. However, in order to efficiently validate software detection or correction mechanisms, experiments need to be deterministically repeatable. As a consequence, injection mechanisms to *emulate* the effects of physical perturbations on the hardware or software in a more repeatable manner, such as *pin-level FI* [AAA+90] (also called *hardware FI*

with contact [HTI97]) and *software-implemented FI* (SWIFI), gained popularity (e.g., FERRARI [KKA92], DOCTOR [HSR95], FTAPE [TI95], Xception [CMS98]).

2.1.1 From Hardware to Software FI

Although FI was traditionally intended to emulate hardware errors, the software-based injection techniques arising from SWIFI were soon utilized to emulate software defects as well. FIAT [SVS+88; BCS+90] was explicitly designed to emulate the effects of both hardware and software defects. The term *software fault injection* (SFI) [CLN+12; CM13; CN13; NCD+13a] was later used to distinguish the emulation of residual software defects from hardware error emulation in SWIFI. While for the emulation of *expected* hardware errors *only* operational FI was relevant[1], both operational and debug FI (the latter often also termed *robustness testing* [KKS98]) became valuable strategies for SFI with similar applications to operational and debug (correctness) testing [FHL+98].

A large body of research work covers fault models for operational FI. As this thesis focuses on fault models for debug FI, software defect models for operational FI are not discussed in further detail. I refer the interested reader to Roberto Natella's PhD thesis [Nat11] as a concise documentation and discussion of the state-of-the-art in operational FI models for emulating software defects.

2.2 Fault Models for FI: What, Where, and When to Inject

Although the goals of the approaches differ, fault models for both operational and debug FI have evolved from the same basic fault taxonomy, which classifies faults according to:

Type: How a fault manifests

Location: Where a fault is located in the system

Trigger: The circumstances that lead to the fault's activation

Persistence: Whether the fault is permanent[2], transient, or intermittent

The classification of software faults according to *defect types* [CKC91] and *defect triggers* [SC91] forms the foundation of *orthogonal defect classification* (ODC) [CBC+92]. The classification of defect types provides feedback on the development

[1] As all aspects of software execution depend on hardware, software systems may fail arbitrarily upon *arbitrary* hardware failures. Consequently, software failures cannot be expected to manifest as a finite set of failure modes. This limits the utility of debug FI, as no practical test oracle can be established.

[2] In other taxonomies, such as IEC 61508 [IEC99] and derived functional safety standards, these faults are called *systematic*, as they are deterministically reproducible.

process, whereas the defect trigger classification provides feedback on testing and other verification processes. The term "defect" denotes the same concept as "fault" in this thesis and the terms are used interchangeably to maintain consistency with different taxonomies in the discussed articles.

Fault persistence is an important fault property of hardware faults that may occur, vanish, and re-occur in a non-deterministic manner. While software faults are always permanent, unless evolving or self-modifying code has to be considered, transient or intermittent faults may be suitable to achieve accelerated failure causation for improved debug FI experiment throughput. The concept of such *failure acceleration* has first been proposed by Chillarege and Bowen [CB89] with the restriction that "the fault model is not altered" by such modifications. The authors, thereby, implicitly state that a fault model comprises any properties related to the fault occurrence, but not to its activation or propagation. They proceed to describe an injection scenario with a fault type that is chosen by its expected impact on the system and replicate it over an entire memory page to increase the likelihood for activation. The selection of the fault type according to a goal (the deliberate provocation of a failure) rather than resembling typical faults as closely as possible marks an early approach to debug FI.

Christmansson and Chillarege [CC96] use the ODC classification to specify a software fault model for FI and attempt to achieve high failure rates through modification of the injection mechanics. Besides selecting types and locations according to observations of residual faults in deployed software, they discuss injection timing strategies for software faults that aim for failure acceleration.

Motivated by empirical data indicating that rare events, which are by definition seldom covered by operational FI, are a frequent cause of critical failures in deployed systems [Hec93], Voas and Miller [VM95] propose to use *inverted* probability distributions to select test cases. This strategy is the exact opposite of operational FI or testing approaches and has been proposed to assess the error propagation sensitivity of different source code locations [VM95; VCM+97] within a SUT. For robustness assessments, however, it is important that the SUT is not modified to obtain dependable results on the SUT behavior. The SUT is, consequently, only exposed to external faults.

2.3 FI for Robustness Testing

A variety of robustness testing tools have used and are using FI to expose software to external faults. Ballista [KD00] exposes POSIX implementations to data-type-specific boundary and exceptional values. The SUT reaction is classified according to a set of failure modes classified by the CRASH scale [KKS98]. The acronym is constructed from the first letters of the failure mode names.

Catastrophic The injected fault has a system-wide effect on any provided service. For POSIX operating systems, a system crash or hang that renders the entire system unresponsive is catastrophic.

Restart The injected fault causes the invoked SUT service to hang, i.e., a service is requested and no observable behavior results within acceptable time. The amount of time that is deemed acceptable is usually derived from response times for fault free operation.

Abort The injected fault causes the abnormal termination of the system issuing the service request to the SUT. In the context of POSIX invocations, this is usually caused by a signal from the operating system. Segmentation faults are prominent examples of this class.

Silent The SUT service terminates without producing a valid result and without indicating an error, when it should have.

Hindering The SUT service terminates without producing a valid result and indicates a *different* error than it should have, thereby misleading debugging efforts.

The CRASH scale originates from correctness testing approaches and is well suited for classifying observed behavior according to *specified behavior* for invalid and stressful inputs. The silent and hindering failure classes explicitly refer to expected return values that require specification. The catastrophic, restart, and abort failure classes classify SUT responses to unspecified conditions. Consequently, the Ballista authors focus their robustness tests on these failure modes, which they summarize as "doesn't crash, doesn't hang" [Koo; KKS98; KDD08]. The CRASH scale and variants have been widely adopted for failure mode classifications in the robustness testing literature. The failure modes considered in this thesis are discussed in detail in the setup discussions for each conducted experimental study (Chapters 4 to 6).

Ballista has been applied to different operating systems that implement POSIX, such as different Unices and the microkernel-based QNX operating system. While the Windows operating system family does not implement POSIX, it provides similar functions via the `win32` system library. By mapping the data types commonly used in the Windows programming ecosystem to the POSIX data types, Ballista has been ported and applied to Windows systems as well [SKD00]. The RIDDLE tool [GSS98] followed a similar goal, but targeted Windows utility programs, rather than actual operating system functions. RIDDLE creates faulty inputs by randomly generating syntactically valid command line parameter strings. Fault models with a random component are commonly referred to as *fuzzing* approaches.

Following a similar philosophy as Ballista, HEALERS [FX02] automatically tests C library functions for robustness issues using a data type fault model and automatically applies protective function wrappers in case any robustness issues are identified. HEALERS starts with tests for the data types specified in the function signatures. If a robustness issue is detected (e.g., a segmentation fault), the data type from the signature is narrowed down to the next smaller[3] subtype and tests are executed for this subtype. This iterative process continues until no robustness issues are found. After identifying these *robust types* for the function parameters, function wrappers

[3] in terms of bit length

are generated that return error codes if arguments passed to the function exceed their robust types.

Approaches similar to those that were first implemented by Ballista and RIDDLE became popular for testing *commercial off-the-shelf* (COTS) systems. As these systems are applied in scenarios that were not necessarily envisioned by their creators, they may fail in dangerous ways if exposed to unanticipated input. Moreover, to protect their intellectual property and ensure the corresponding systems remain a continuous source of revenue, their source code is usually not made available for review by users. Hence, COTS systems are often integrated as black-box components within larger systems. For economical considerations COTS systems became widely applied, even in critical applications [Voa97; Dal00].

Ballista, RIDDLE, and HEALERS inject faults into parameters of individual function calls to the SUT and evaluate the SUT's response by classifying the behavior of this single invocation according to the CRASH scale. This restriction of failure observations to that fraction of the external state, which is only directly related to the perturbed invocation, bears both benefits and drawbacks. On the one hand, robustness tests of this type can be performed at high frequencies, as the results can be classified quickly, and the results from both projects have demonstrated that large numbers of robustness issues are found. On the other hand, it bears a high risk for false negatives, as actual robustness issues *cannot* be identified because only a subset of the external state is considered for failure detection and failures may manifest in the disregarded part. Furthermore, if robustness issues (i.e., internal faults) exist and they get activated by a test invocation, they may result in an erroneous SUT state that does not manifest as a failure immediately. Such issues would require further stimulation of the SUT to foster the propagation of errors to the SUT's interface. Finally, robustness tests that comprise only individual function calls do not account for SUTs that require *warm-up* phases to achieve their operational steady state (cf. *spurious faults* [GKT13]), such as operating systems and other complex and stateful SUTs. In order to compensate for these drawbacks of the proposed techniques for stateful systems, a number of approaches have been developed that implement fault injections by intercepting SUT interactions with its operational environment, rather than testing the SUT in isolation. These approaches, which I refer to as *robustness assessment* in contrast to *robustness testing*, are discussed in the following section.

2.4 FI for Robustness Assessments

MAFALDA [FSM+99; RSF+99] has been developed to assess the robustness and propagation properties of off-the-shelf microkernel-based operating systems often found in safety- and mission-critical systems. The tool provides two different injection mechanisms.

Parameter bit flips Upon microkernel invocations, a randomly selected bit in a randomly selected parameter is changed from logical 0 to logical 1 or vice versa.

Microkernel address space bit flips Upon memory access to a randomly preselected address in the microkernel, either for reading/writing data or fetching instructions, a random bit in the accessed memory is flipped.

While the second injection mechanism apparently implements classical SWIFI to emulate hardware defects and is not suitable to assess system robustness (as the SUT is altered), the first approach resembles the Ballista and RIDDLE robustness testing approaches. It mainly differs from these tools by the targeted SUT, the chosen fault type for injections, and the FI test setup. In contrast to Ballista and RIDDLE, MAFALDA intercepts SUT invocations from its actual environment and thereby circumvents the aforementioned dilemma of these tools that generated inputs are either very complex or of limited utility for testing stateful systems. Rather than crafting external faults as a possibly long sequence of artificial inputs, external faults are created by transforming subsequences of legitimate input. To mitigate identified robustness issues, the authors suggest to use fault containment wrappers for additional parameter validation at the COTS microkernel's interfaces.

Voas [Voa98] discusses the problem of COTS integration in critical systems on a more abstract level and proposes a general approach for testing the consequences of COTS system failures via FI. Similar to MAFALDA, the COTS system's interfaces are intercepted and erroneous behavior of the COTS system is simulated by modifying the interface-mediated interactions with other systems. The author discusses three mitigation strategies (static fault-tree analysis, slice testing, and fault containment wrappers) for COTS systems and recommends their comparative assessment using FI.

Following these composition-focused approaches, Hiller, Jhumka, and Suri [HJS01] developed a framework to quantify data error propagation across interacting components in a composition and implemented the approach as an FI-based tool for propagation analysis [HJS02]. Building on this conceptual framework, Johansson and Suri [JS05] built a fault injection tool to measure the robustness of a COTS operating system kernel against driver failures and to assess the propagation of COTS device driver failures to the operating system's application interface [JSJ+05; JS05; JSM07b; JSM07a]. The supported fault models, also termed *error models* in the referenced articles, are specified according to the fault properties identified by Christmansson and Chillarege [CC96], as discussed earlier in this chapter: The fault location (*where* to inject), the fault type (*what* to inject), and the fault timing (*when* to inject). As fault locations, any data transferred as function call parameters or function return values from drivers to the OS kernel are chosen. The fault types are those previously introduced and used by other robustness testing tools: Data-type-specific values (inspired by Ballista), parameter fuzzing (inspired by RIDDLE), and parameter bit flips (inspired by MAFALDA). The tool supports a large variety of fault timing configurations in terms of event-based fault triggers (cf. [CC96]) and fault latencies (*for how long* to inject). The published results mostly cover transient faults that are triggered upon first invocation of the targeted kernel or driver services. The influence of injection timing on propagation analyses is discussed in detail in [JSM07a].

Marinescu and Candea [MC09] applied fault injections to analyze error checking and recovery code for third-party library invocations in programs. Their tool (library fault injector, LFI) first identifies all shared libraries that a program uses and then creates a *fault profile* for each function exported by each library. The fault profile consists of all error codes that can possibly be returned to a caller. This set is determined by disassembling the library machine code, constructing the control flow graph (CFG) for each function (including functions of other libraries, if any are used), and searching for paths in the CFG that propagate error codes to the registers or memory locations via which return codes are passed back to a caller. The robustness notion, on which LFI is based, overlaps with the robustness notion in this thesis, but they are not identical. As LFI covers error codes returned by library functions, these would be considered valid inputs in this thesis *if they are documented.* As the documentation of third-party libraries may be incomplete, undocumented error codes would fall under the definition applied in this thesis.

I adopt the widely used fault model notion (e.g., [CC96; JS05; JSM07b; JSM07a; MC11]), where fault models for FI-based robustness assessments 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 2.1 lists a number of existing FI frameworks that have been applied for software robustness assessments similar to those considered in this thesis, along with their reported fault model support. Fault models are often discussed only implicitly in the literature and some FI frameworks provide mechanisms for flexibly extending the set of supported fault models, e.g., by user defined injection triggers or fault types. Thus, Table 2.1 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 [CMS98].

For the injection of external faults, two different injection locations exist, for which a detailed discussion in the context of injection mechanisms is provided in Section 3.1.1:

- External components interacting with the SUT (ext. comp.)
- A chosen interface under test (IUT) of the SUT, via which it interacts with external components

The spectrum of fault types in the related literature includes:

- 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)
- Substitutions by random bit patterns (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. The following triggers are used by the listed frameworks:

- Upon first execution of an injection location (1st occ.)
- After a defined number of n executions of an injection location (nth occ.)
- After a predefined amount of time has passed (Timer)
- After some predefined exception or function call occurs (X-call)
- After a predefined sequence of function calls (Call Block)

Table 2.1: Fault models for robustness evaluations in the literature

Framework/Authors	Fault Location	Fault Type	Fault Latency	Injection Trigger
MAFALDA [AFR02]	ext. comp. IUT	SEU MBU DT	Transient Permanent	1st occ.
Albinet et al. [AAF04]	IUT	DT	Transient	1st occ.
Kalakech et al. [KKC+04]	IUT	SEU DT	Transient	1st occ.
Xception [CMS98]	ext. comp. IUT	SEU MBU DT FZ	Transient Intermittent Permanent	1st occ. nth occ. Timer X-call
G-SWFIT [DM02]	ext. comp.	Coding mistakes	Permanent	1st occ.
Medonça & Neves [MN07]	ext. comp.	Coding mistakes	Permanent	1st occ.
Johansson [Joh08]	IUT	SEU DT FZ	Transient Intermittent Permanent	1st occ. nth occ. Timer X-call Call Block

3 System Model

In Section 3.1 of this chapter I provide an abstract description of the FI process and terminology assumed in Chapters 4 to 6. As the target platforms, which I used to validate the proposed metrics, models, and performance improvements, are commodity mobile operating systems, I discuss their common structure and abstractions and how these match the abstract system model applied in the FI process in Section 3.2. The detailed experiment configurations used for the studies in Chapters 4 to 6 are presented in the respective chapters. Parts of this chapter are based on material that has been published or accepted for publication [WSS+11; WTS+13; WSN+15; WPS+15].

3.1 FI Process and Abstract System Model

As detailed in Section 1.1.3, the subject of this thesis is the usage of FI for detecting robustness issues, which is termed debug FI. As discussed in Chapter 2, there are two different strategies to debug FI, which are both widely used, but differently suitable for different types of SUTs. I refer to debug FI for stateful, complex, and interconnected SUTs within their operational environment (OE), similar to the methodology applied by Fabre et al. [FSM+99], Voas [Voa98], Hiller, Jhumka, and Suri [HJS01], and Johansson et al. [JSJ+05], as *robustness assessments*. This differs from early *robustness testing* approaches, such as Ballista [KD00] and RIDDLE [GSS98], which test SUTs as isolated units (cf. Chapter 2). The relation between SUT and OE is depicted in Figure 3.1.

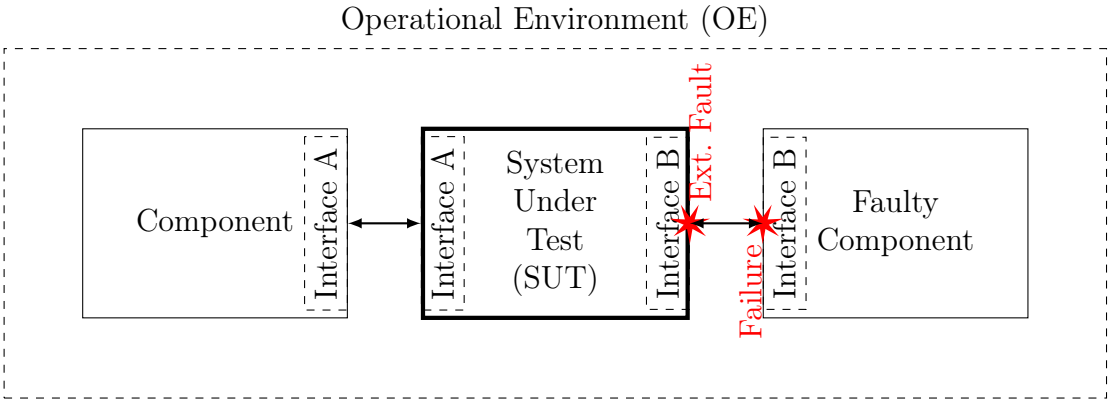

Figure 3.1: System under test (SUT) and operational environment (OE) in FI-based robustness assessments

Components in the OE interact by requesting (*invoking*) other components' *services*. *Interfaces* are sets of services and, therefore, constitute the means by which each component's functionality can be accessed. An interface via which a component provides services is called an *export interface* of the component. If a component C1 invokes services from an export interface of another component C2, these services belong to C1's *import interface*. Figure 3.2 illustrates interactions in compositions, where arrows indicate service invocations (more precisely, *possible* invocations, also termed references [Ryd79]). A component that provides services to the SUT is referred to as a SUT *server* component. A component that uses services that the SUT provides is referred to as a SUT *client* component.

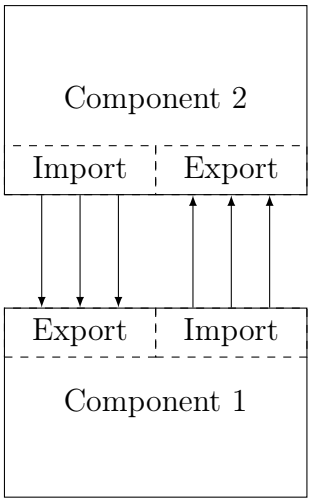

Figure 3.2: Import and export interfaces

In order to assess its robustness, FI exposes the SUT to external faults. As any external fault affecting the SUT must be mediated by its interfaces, the injection of external faults can be achieved by either provoking actual OE component failures through *code mutations* of these components or by directly perturbing interactions between OE components and the SUT, as I detail in Section 3.1.1. The interface via which the SUT is exposed to external faults is called the *interface under evaluation* (IUT). Injections into the SUT itself are not considered, since the robustness of a SUT is defined by its response to *external* faults. Figure 3.3 illustrates the different injection locations for external faults, represented by the filled circles.

Robustness assessments comprise three phases that are repeated for each chosen fault model, workload, and OE.

Configuration: A fault model and workload are selected and the SUT's OE is set up accordingly. The OE is instrumented with *interceptors* (e.g., interface wrappers) to perform injections during the SUT's interactions with the OE and to monitor the SUT's response to these interactions, including failure detection.

Campaign execution: FI *campaign* refers to a sequence of FI runs, pertaining to

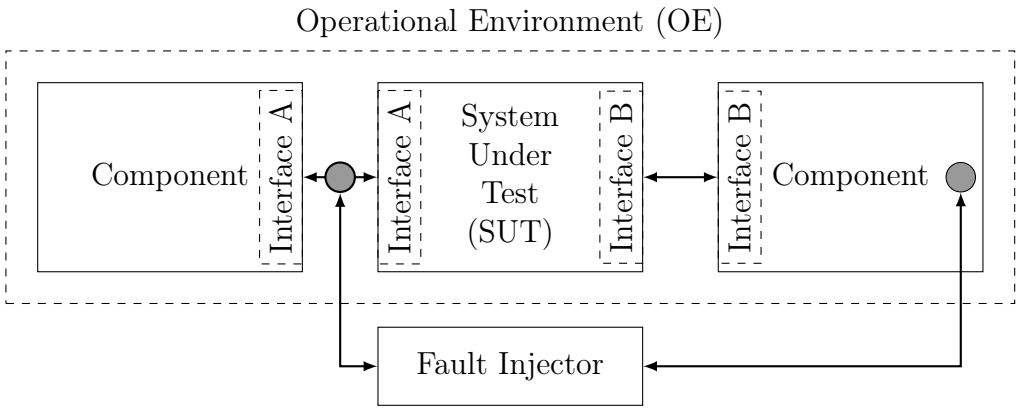

Figure 3.3: Injection locations in the SUT's OE considered in this thesis

a specific fault model, workload, SUT, IUT, and OE. A FI *run* comprises the following steps:

1. Starting the SUT and other components of the OE if necessary
2. (Re-)setting the SUT and other components of the OE to a known and reproducible state
3. Injecting a specific fault derived from the chosen fault model
4. Executing the workload
5. Detecting (and logging) the injection's effects
6. Stopping the SUT/OE, if required

The start and stop operations are usually only required for the first, respectively last, runs of a campaign. Between runs of the same campaign, the reset operation commonly suffices for software SUTs, unless the entire OE is rendered inoperable as a consequence of injections. I discuss the corresponding failure modes and their implications on the FI process separately for each conducted assessment in Chapters 4 to 6. I use the terms FI *run* and *experiment* interchangeably in this thesis.

Data analysis: After campaign execution, the detected injection effects are investigated to determine violations of the previously specified negative requirements. Identified robustness issues can be tracked back and fixed, if the source code of the SUT is available, or error handling code (e.g., fault containment wrappers [Voa98; SRF+99]) can be applied in the OE upon SUT deployment.

As performing FI runs in the campaign execution phase is a highly repetitive task, it is usually automated to optimize experiment throughput and to prevent human errors from threatening result validity. To illustrate the FI setup, the functionality of individual components, and their interplay, Figure 3.4 (adopted from Hsueh, Tsai, and Iyer [HTI97] with modifications) displays the general structure of FI tools. The *Fault*

Injector component exposes the SUT to external faults by performing fault injections into the OE. It is configured and triggered by a central controller, which also triggers SUT and OE initialization and exposes it to a workload via the *Workload Generator*. The exposure to a workload in addition to the faultload is required to trigger fault injections or activations in the interaction with passive server components that do not trigger SUT execution themselves. SUT execution is a prerequisite for robustness assessments, as internal faults need to be activated and their effects propagated to their external states for detection. The test result is evaluated by the *Data Analyzer* based on data that the *Monitor* component collected during the test.

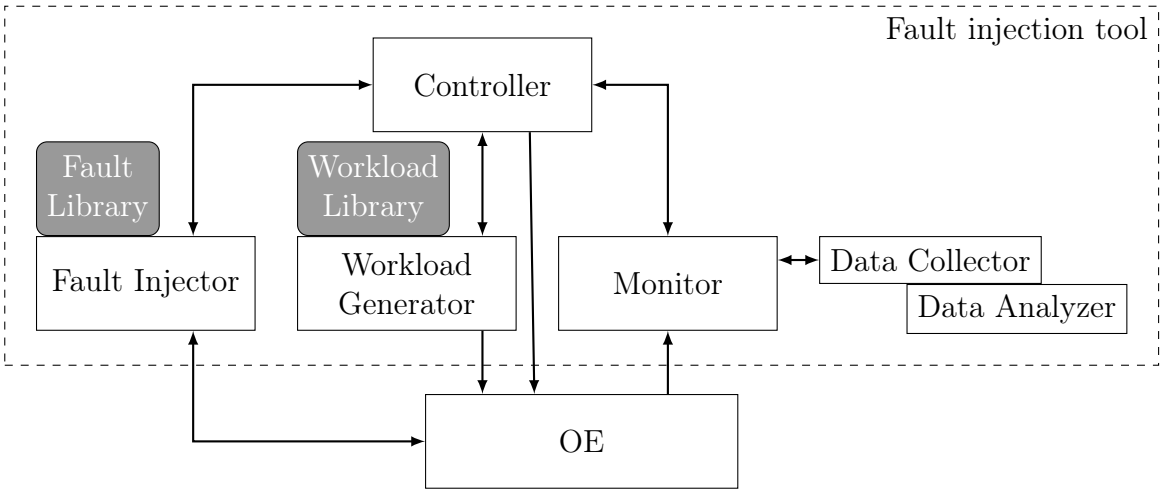

Figure 3.4: Common modules of a fault injection tool. Adopted from Hsueh, Tsai, and Iyer [HTI97] with modifications

Failure detectors, internal and external to the OE (not visualized in Figure 3.5), monitor the SUT, its clients, and its servers for symptoms of predefined SUT failure modes of interest. The corresponding information is forwarded to the monitoring component of the framework, where it is aggregated for failure mode inference. These inferred failure modes are reported to the data collector component as the results of the injection experiments and the data is then used for offline analysis by the data analyzer component, from which robustness properties of the SUT (more precisely, their violation in the case of debug FI) are derived.

3.1.1 FI Mechanisms: Code Mutations vs. Interface Error Injections

In order to expose a SUT to external faults in its OE, two different mechanisms are widely applied:

Code mutations The most straight-forward way to emulate the effects of faulty components in the SUT's OE is to inject faults into these components. The corre-

sponding modifications are referred to as *code mutations* [VM97], as they usually require changes to the components' source code.

Interface error injections The work on robustness testing (e.g., [GSS98; KD00; FX02]) inspired injections into data that is passed via the interfaces between the SUT and components in its OE to emulate the effects of faulty OE components. As these injections are supposed to reflect the result of *activated* intra-component faults, the approach is commonly referred to as *interface error injection*.

Both approaches have methodological strengths and weaknesses that carry more or less weight depending on their intended application. With code mutations, it is possible to closely resemble software fault distributions encountered in software development processes or actual operation after deployment [DM03a; DM06; NCD+10; NCD+13a]. This makes code mutations the preferred choice for operational FI, as widely used interface error models (such as bit flips, fuzzing, and data type models) lack representativeness of actual residual software faults [LNW+14].

However, interface error injections have a practical advantage over code mutations, as they do not require the availability of component source code. For interface error injections, access to the SUT's interface with its interacting components suffices and such access is usually given, as its lack would severely restrict its integration with other systems. Moreover, the use of interface injections does not require a different OE to be set up for each experiment. While code mutation results in a set of mutated components, which need to be loaded individually for each expeiment, the modification of interface data is a much simpler operation and the same interface injector can be configured at run time to perform a large variety of diverse perturbations. Although techniques exist to modify binary components dynamically at run time and without source code access, they often turn out to be imprecise (e.g., [NCD+10]). A corresponding technique is used for the SEU fault model introduced for the validation of the fault model efficiency criteria introduced in Chapter 4.

As representativeness of injected faults is a lesser issue for debug FI, with which this thesis is concerned, and, given the practical advantages of interface error injections, the metrics and fault models developed in this thesis focus on interface injection scenarios. Although this focus is reflected by the case studies conducted for experimental validation of the proposed concepts, the metrics, models, and performance tuning mechanisms at the core of this thesis do apply for code mutations as well with little or no modification. Although interface injections are often referred to as *error injection* in the literature, as it evolved as a variant of robustness testing, I equally use the terms fault, fault model, and fault injection for code mutations and interface error injections in this thesis, as my interest is not an evaluation of the OE, its fault tolerance or propagation characteristics, but on the SUT. For the SUT, injections at the interface do not constitute errors, but external faults.

Interface injections are assumed in Chapters 4 and 5 due to their wide adoption in debug FI. The achievable performance improvements for debug FI investigated in Chapter 6 equally apply for code mutations and interface injections and I chose to

perform code mutations instead of interface injections. The fault injection framework presented in Chapter 7 is capable of performing both types of experiments.

3.2 Operating System Robustness Assessments

The metrics, models, and approaches introduced in this thesis are validated in debug FI campaigns for robustness assessments of mobile commodity operating systems (OSs). The conducted campaigns specifically target the kernels' device driver interfaces for two reasons:

Practical relevance Device drivers have been shown to contain significantly higher numbers of defects than other kernel subsystems [CYC+01; PTS+11] and at the same time their failures have severe consequences on overall system stability [Sim03; GGP06]. Although this vulnerability of monolithic kernels has been identified early [CYC+01; Sim03] and considerable efforts have been made to improve device driver testing and verification [BCL+04; BBC+06; SJF+06; SJS+10; KV10], the defect counts of drivers have not decreased [PTS+11]. As device driver faults are apparently difficult to find and eliminate, other approaches have focused on fault tolerance mechanisms to prevent the effects of faulty device drivers from affecting overall system stability [SBL03a; LUS+; ZCA+06; TCF+07; WRW+08; CCM+09; HBG+09; JKJ+10]. FI is commonly applied to assess whether these mechanisms operate correctly and in all relevant locations where driver faults manifest as errors (e.g., [FSM+99; JSJ+05; JS05; MN07]). In summary, debug FI for identifying robustness issues in OS driver interfaces has practical relevance. If robustness assessments with higher order fault models uncover issues that are not identified in assessments with traditional models, these results should be taken into consideration for the design and validation of fault tolerance mechanisms in OSs.

Higher order fault impact In Section 1.2, I have motivated the need for new fault models by the increasing degree of parallelism in computer hardware and its impact on software that is operating such systems. Given that the main purpose of OSs is to abstract from hardware configurations and hide their details from application programs, any changes in the computing hardware landscape directly affect OS implementations. The change from sequential to parallel computers required OS kernels and any subsystems executing in kernel mode, such as device drivers, to become thread safe to fully exploit the potential of parallel hardware. While thread safety has been a concern for *multiprogramming* OSs (e.g., [Dij68]) executing on sequential hardware as well, it was achievable (and in most cases achieved) by coarse-grained locking in the kernel. In Linux, for instance, the *big kernel lock* (BKL) originally prevented the simultaneous execution of kernel code for different processes. Although such coarse-grained locking provides thread safety, it hampers performance by making the execution of unnecessarily large code portions exclusive for a single process or thread at

a time. Although fine grained locking better exploits performance gains from parallel processing, it bears the risk of unprotected code portions that would actually require protection by locks. For device drivers this risk is higher than for other kernel code portions, as drivers are commonly developed and maintained by third parties, usually hardware manufacturers, rather than core kernel developers. Moreover, a number of previously common mechanisms to achieve locking do not equally work on sequential and parallel hardware. Disabling interrupts, for instance, allows for exclusive execution on single processors, but not on multiprocessor systems, where interrupts are usually processor-local. Incorrect locking is known to lead to data races and other synchronization problems, which are difficult to identify and debug (e.g., [XPZ+10; LVT+11; KZC12a; KZC12b]).

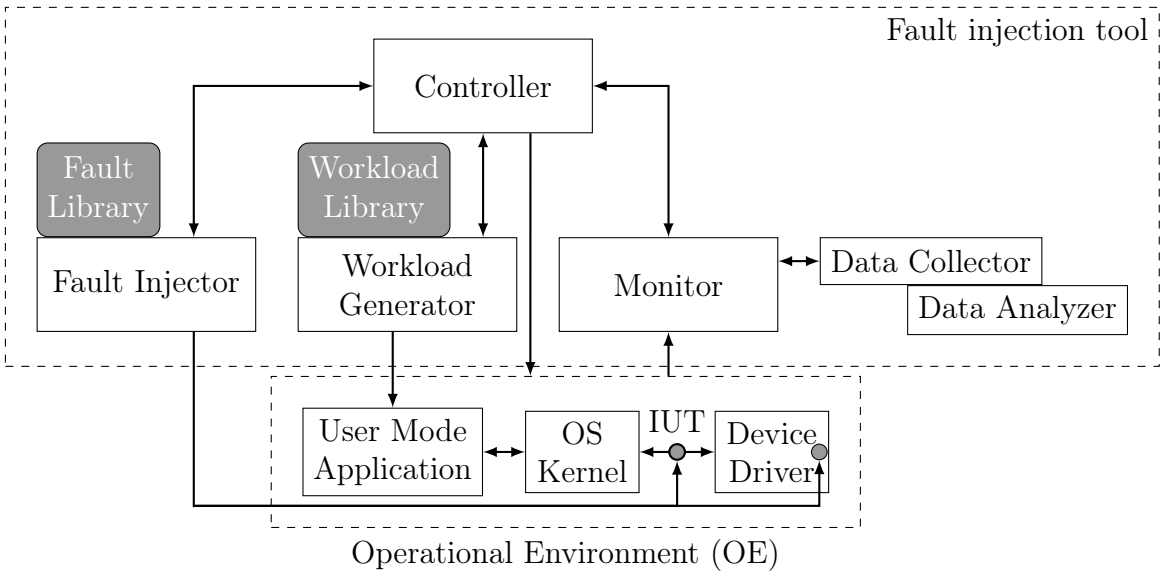

Figure 3.5: Robustness assessment setup

To perform robustness assessments as described in Section 3.1 for OSs, the kernel (the SUT) is exposed to external faults at its device driver interface. Figure 3.5 displays the experiment setup, taking the OE structure and considered injection locations from Figure 3.3 into account and mapping OE components to relevant entities in the OS assessment setup. Besides the SUT, the OE comprises the drivers that are targeted for injections and other drivers that are required to execute the system on the chosen platform. While injections can be performed by code mutations, the focus of this thesis is on interface injections, as discussed in Section 3.1.1. The injection of interface errors is performed by interceptors that are implemented as interface wrappers at the driver interface. To trigger faults injected into driver code or to trigger the execution of interface injections, the SUT is exposed to a workload generated by user space applications. These applications specifically request kernel functions that are related to the device drivers targeted for injection, which is a common practice in

debug FI to achieve failure acceleration (for a discussion of failure acceleration see Section 2.2). Apart from injection interceptors and monitors, all experiment control logic is separated from the target system by hardware-supported virtualization or an "air gap" (an actual physical separation of hardware) to prevent it from being corrupted by the effects of injected faults. The experiment controller is responsible for storing experiment configurations and data, orchestrating the interplay of OE, SUT, injections and detection during experiment runs, and performing experiment campaigns in an automated manner, including OE/SUT start-up and reset between runs.

4 Efficiency Metrics for Robustness Assessments

While for operational FI the fault models to be used are dictated by the intended application scenario for the SUT, fault model selection criteria for debug FI are rarely discussed in the research literature. In order to assess the comparative effectiveness of higher order fault models, suitable metrics for their effectiveness need to be determined and put into relation with the expected costs that their usage entails. This chapter proposes a set of four metrics to capture both effectiveness and costs of fault models for debug FI. Their suitability is evaluated in a case study of an off-the-shelf real-time operating system that is exposed to faults at the driver interface. The metrics presented in this chapter form the basis of the fault model efficiency evaluation of higher order fault models in Chapter 5. This chapter is based on a conference paper presented at ICSE 2011 [WSS+11].

4.1 The Fault Model Selection Problem

Under a constant feature-driven market pressure and due to their ever increasing complexity, many software systems are often released without being sufficiently tested. Even if a software component is considered to be sufficiently tested for one application scenario or operational environment, it may be insufficiently tested for reuse in another one. Especially off-the-shelf (OTS) software systems 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. Moreover, the same component may be applied in numerous almost arbitrarily different application scenarios and configurations that are impossible to foresee, let alone test, by developers. Thus, failures frequently result when the operational environment of a deployed system differs from the pre-release lab configurations used in the testing phase of the software development process. As a consequence, released software may exhibit potentially dangerous behavior in operation due to deviations from specified behavior or undesired unspecified behavior (cf. Section 1.1.2).

To cope with this deployment variability aspect, FI has become a popular technique for robustness assessments of OTS software systems with respect to unexpected operational conditions. In order to assess whether a software component is "sufficiently" robust for release, it is exposed to external faults in a controlled manner while its

reactions are closely monitored. The downside of this approach is that it is well suited for demonstrating the *presence* of robustness issues, but not the *absence* thereof. In order to show their absence using FI, it would be necessary to expose the SUT to every possible external fault. 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 [Hua75]. FI-based robustness assessments, therefore, require the selection of suitable faults from the set of all possible faults. This raises the question *which faults to select for the assessment.*

A common solution to this problem is provided by application-specific *operational profiles*, i.e., probability distributions for stimuli that the SUT is exposed to during operation. Their application to the fault selection problem is based on the argument that a total absence of robustness issues is too strong a condition, since a robustness issue (an internal fault) does not necessarily result in a failure mode that is of importance for a given application context. It does so only if the system is exposed to an external fault which triggers the issue. This relaxes the problem to proving the absence of any internal faults that can be triggered by external faults during operation. For this purpose, operational profiles of the SUT are derived from its intended operational context, including (but not limited to) *typical* classes of faults, i.e., faults that are expected to be frequently encountered during operation.

This approach, which I refer to as *operational FI* (cf. Section 1.1.3), has two major drawbacks. First, the robustness evidence derived from such an evaluation is only valid for the considered operational context of the evaluated system. This implies that a system 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, unanticipated conditions that are by definition not covered by operational profiles [Hec93; VCM+97; LM03].

Considering these drawbacks, I investigate an alternative fault selection criterion based on fault effects. Instead of selecting *typical* faults for robustness assessments, preference is given to those faults that make evaluations *most efficient* in terms of detected robustness issues and the required effort for their detection. I refer to this approach as *debug FI* (cf. Section 1.1.3). Debug FI also relies on application scenario specific information, as it relies on negative requirements (i.e., how the SUT should *not* behave) for the identification of relevant *failure modes* and implementation of suitable failure detectors. However, different application scenarios, and even different SUTs, often share a common subset of negative requirements for safe operation, as the wide adoption of the CRASH scale for robustness testing demonstrates (cf. Section 2.3).

It is easy to state a perfect fault model selection criterion for debug FI, but difficult to find a criterion that is practical and approximates the perfect criterion well. The perfect criterion for debug FI selects faults that uncover the largest fraction of all robustness issues in a system with the smallest experiment overhead. As the number of total robustness issues in a system is unknown, the fraction of uncovered robustness issues cannot be known, either. Consequently, the criterion is not practical, as it can-

not be applied in practice. A similar problem is known in (debug) correctness testing, where the goal is to uncover the largest possible fraction of correctness issues (cf. Section 1.1.2). For correctness testing, a large number of practical *test adequacy criteria* and even an axiomatic theory of test quality have been proposed to approximate the impractical perfect criterion. Inspired by these criteria, which I review in Section 4.2.2, I propose a set of imperfect but practical metrics for the comparative evaluation of fault models to aid the selection of suitable models for debug FI experiments.

This chapter makes the following contributions:

- A method to compare the efficiency of different fault models for FI-based robustness assessments
- A set of fault model efficiency metrics for this purpose based on structural criteria applicable in both black-box and white-box scenarios[1]
- A pragmatic set of guidelines for the application of these metrics in FI-based robustness assessments

As a case study, I perform FI experiments on an embedded operating system (OS) kernel (Windows CE 4.2) and demonstrate the effectiveness of the proposed approach, comparing four commonly applied fault models.

The chapter is organized as follows: Section 4.2 introduces basic terminology and related work. Section 4.3 introduces the metrics developed for cross-model efficiency evaluations, whose application is demonstrated in an experimental evaluation presented in Section 4.4 along with a demonstration and discussion of their utility in Section 4.5.

4.2 Related Work

Although the fault model selection problem has conceptual commonality with the test case selection and prioritization problem, it has attracted much less discussion compared to its correctness testing counterpart. Following a discussion of fault model selection and comparison in Section 4.2.1, I review testing adequacy criteria in Section 4.2.2, as the metrics proposed in this chapter are inspired by structural criteria from correctness testing.

4.2.1 Comparative Assessments of Fault Models

Although a multitude of fault models for software robustness evaluations are discussed in the literature, most of them have been developed for operational FI assessments. Consequently, their adequacy derives solely from similarity with actual operational faults that they are intended to mimic. Sophisticated analyses have been conducted

[1] i.e., no source code access is required to apply the structural criterion

to assess and ensure the representativeness of injected faults [DM03a; DM06; NCD+10; NCD+13a].

For debug FI, in contrast, where the most important property of a fault model is the identification of robustness issues, fault model assessments and comparisons are surprisingly rare. Moraes et al. [MBD+06] compare the effects of fault injections at different locations in the SUT's environment. To assess whether fault injections into systems that interact with the SUT yield different results than direct injections into SUT services, the authors perform injections of representative software defects using G-SWFIT [DM06] and compare the effects with those of injections into interface services using a data type fault model inspired by Ballista (see Section 2.3). The tests are performed on two SUTs: A database management system and a mission-critical real-time system configuration based on RTEMS [OnL]. The authors conclude that service-level injections, which are generally less costly in terms of implementation complexity and execution time, have different effects than software defects in interacting components. Although the study provides limited insight into the nature of and reasons for the observed deviations, this is an important result, which indicates that operational FI models (emulated software defects) do not automatically yield high detection rates for robustness issues, as required for effective debug FI. Except for the RTEMS system startup interface, interface injections provoke higher numbers of SUT failures. While undesirable for operational FI due to apparent lack of representativeness, the interface injections are favorable for debug FI, as their higher failure rates entail a higher probability for identifying robustness issues. Consequently, the adequacy criteria for operational FI cannot be easily adopted for debug FI.

With the goal to assess the degree by which the Windows CE kernel propagates driver errors to the user space API interface and the effect of fault models on the assessment, Johansson, Suri, and Murphy [JSM07b] compare the effectiveness of three different fault models (a single bit flip model, a Ballista-style data type model, and a parameter fuzzing model) in interface injections between device drivers and the Windows CE kernel. The conducted propagation analysis constitutes a special case of robustness assessments, where the service interfaces for test input and failure detection differ. The applied effectiveness criteria are (i) the observed propagation probability (*driver error diffusion* [JS05]) and (ii) the number of observed system outages (crash failures) in response to injections (*error impact*). In order to account for overheads the different models imply, their (iii) *implementation complexities* and (iv) entailed *execution times* are also measured and compared. Using the error impact criterion, the bit flip model is identified as most effective in detecting robustness issues, but also as prohibitively costly in terms of execution time. The fuzzing model entails a significantly lower execution time, but also fewer detections. As a consequence, the authors propose a composite model that combines few effective bit flip tests with 10 fuzzing injections. The composite model saves approximately half of the injections, while providing similar diffusion results as the individual models it is composed from.

While the chosen metrics match the intuitive understanding of fault model effectiveness and performance well, the chosen metric definitions have a number of drawbacks for comparative fault model evaluations.

1. None of the used metrics is normalized. The bit flip model, for instance, is identified as *expensive* because of the overall experiment run time for all tests and the model simply yields the highest test counts. At the same time, the high number of test counts also contributes to the model's higher failure rate.

2. Driver error diffusion is a metric specific to propagation analysis and not directly applicable to measure the suitability of fault models for general robustness assessments. Moreover, the metric is not used to comparatively evaluate and rank different models in the assessment, but only as an argument for the composite model adequacy.

3. Rather than driver error diffusion, the effectiveness of the bit flip model derives from the number of services identified as exhibiting non-robust behavior in the paper. However, the comparison is only based on an implicit procedural description and no metric is defined.

4. Implementation complexity is only informally discussed without giving any definition.

I propose a set of metrics that is based on the ideas from Johansson, Suri, and Murphy [JSM07b], but addresses the aforementioned issues. As the total number of robustness issues is unknown, it is impossible to derive a coverage metric that is both accurate and normalized. Since the same problem applies for test adequacy criteria in correctness testing, I review related work on the issue in the following.

4.2.2 Correctness and Security Test Adequacy Criteria

While the purpose of debug FI effectiveness criteria in this thesis is the comparison of fault models, the main purpose of assessing the effectiveness of test cases for correctness testing are *test adequacy/end criteria*. In order to decide when software is *sufficiently* tested, test adequacy criteria are commonly stated in terms of achieved test effectiveness. As the purpose of software testing traditionally is the detection of software faults [Dij72], software is adequately tested if all or a large fraction of its faults have been identified by the tests. However, as the total number of faults in a program are unknown, no normalized *fault coverage* measure of test effectiveness can be defined that is both accurate and practical. In order to achieve practical normalized measures, requirements on accuracy are relaxed and approximations to fault coverage are proposed and comparatively evaluated in the literature.

Existing work distinguishes between specification based and program based test adequacy criteria. Although it is intuitive to base test adequacy criteria on specifications, because failures are violations of the specification and faults are the attributed causes for failures, program based approaches are more widely used, as they can be derived in an automated manner by program analyses, for instance, on control flow graph (CFG) representations [Wey86]. Specification analyses usually require human interaction, as they commonly rely on natural language comprehension, with the exception of formal

specifications [ROT89]. In their comprehensive review of test adequacy and coverage metrics, Zhu, Hall, and May [ZHM97] include a combination of specification and program based adequacy criteria in their classification. They further classify adequacy criteria as (1) structural, (2) error-based, and (3) fault-based.

In contrast to the structural and error-based criteria, which are detailed in Sections 4.2.2.1 and 4.2.2.3, the metrics proposed in this thesis also consider test *outcomes*, i.e., the SUT's behavior in response to test stimuli, rather than purely syntactic elements. In that sense they are similar to fault-based criteria, which are based on test outcomes for behavioral deviations of the SUT, as discussed in Section 4.2.2.3. However, the proposed metrics do not entail the high measurement overhead that fault-based criteria impose. A similar trade-off has been achieved for the state-based correctness test adequacy criterion discussed in Section 4.2.2.4. The criterion cannot be directly applied for FI effectiveness assessments, as it relates adequacy (i.e., test quality, respectively experiment effectiveness) only to a coverage property of the applied test assertion. The test assertion, which compares the observed SUT behavior with specified behavior, corresponds to SUT failure detectors applied in debug FI. As the detectors are defined by application-specific negative requirements (cf. Section 1.1.1), state-based criteria would be suitable to assess how accurately failure detectors test these requirements. The aim of this thesis, in contrast, is to assess the effectiveness of debug FI fault models, rather than failure detectors. These fault models are part of the input to the SUT in debug FI. An approach that relates test adequacy to test input for fuzzing is discussed in Section 4.2.2.5. In this approach test adequacy is directly derived from coverage of a chosen fraction of the test input space. Although a positive correlation with test effectiveness is empirically demonstrated, the criterion does not resemble effectiveness, as highly effective (i.e., fault revealing) tests may constitute a subset of the input space that has little or no overlap with the deliberately chosen fraction of the input space.

4.2.2.1 Structural Adequacy Criteria for Correctness Testing

Structural adequacy criteria are based on the coverage of syntactic elements of programs. Three widely adopted structural adequacy criteria are *statement adequacy*, *branch adequacy*, and *path adequacy*, which are based on the coverage of program statements (nodes in the CFG), branches in the program (edges in the CFG), or execution sequences (paths in the CFG) respectively. Which adequacy criteria are suitable and if a degree of coverage achieved by a test set is considered "sufficient" or not usually depends on the test policy for a given software project. Full coverage of all statements, branches, or paths is a common requirement for adequacy [ZHM97]. While they can be easily calculated, structural adequacy criteria have a number of shortcomings.

Coincidental correctness: Control flow based adequacy criteria assume that any test data that lead to the coverage of a certain statement, branch, or path are equally suitable for revealing faults. This leads to insufficient detection of, for instance,

neglected conditions [CPY07], i.e., missing branching instructions to handle exceptional cases. Such issues could only be detected if the exceptional cases were by coincidence chosen as test values. For other values, structural adequacy criteria likely give a false sense of safety. Such *coincidental correctness*, i.e., the observation of a seemingly correct result despite presence of a fault in the covered statements/branches/paths, has been recognized as a significant threat to the soundness of structural adequacy criteria and is increasingly addressed in recent work [WCC+09; CLZ+10; WGY+13; ACD+14; ZXZ14].

Unachievable adequacy: In some cases, full coverage of a structural adequacy criterion may not be achievable. Obviously, full statement, branch, and path coverage cannot be achieved if a program contains dead code. A less obvious example is lazy evaluation of branching conditions in some programming languages (e.g., [HGW04; RWH08]), which leads to reduced coverage for condition based adequacy criteria, such as MC/DC [Mil94], which is a common requirement in functional safety standards [Aer92; Aer12; ISO11c].

Adequacy axiom violations: With the goal to provide a general theoretical framework for reasoning about test adequacy, Weyuker [Wey86] introduces a set of eight axioms that are derived from intuitive arguments on properties that a "good" set of tests should fulfill. The introduced axioms are validated in a discussion of weaknesses that several adequacy criteria expose. For instance, the unachievable adequacy for statement, branch, and path coverage discussed above is phrased as the violation of an axiom (termed *Applicability Property* in the article). These structural adequacy criteria (and others as well) furthermore violate the *Anticomposition Property* and the *Antidecomposition Property*, according to which adequacy for a composition does not derive from adequacy for its components and vice versa [Wey86].

4.2.2.2 Error-based Adequacy Criteria for Correctness Testing

Error-based adequacy criteria focus on the selection of test inputs to a SUT that are likely to result in failures. In contrast to fault-based adequacy criteria, there is a stronger emphasis on the observability of fault *effects*. Based on conditional branching instructions in the SUT's implementation, *subdomains* are identified, for which all output is derived from the input in an equivalent manner. Any deviation from correct service is attributed to either computation or domain faults. *Domain faults* result from incorrect implementations of branching conditions. As a result, the border between adjacent subdomains derived from path predicates in the source code does not match the specification and some input is processed by the wrong sequence of statements. *Computation faults* are faults within the basic blocks[2] of the SUT's implementation.

[2] "sections of a program which have the property that if any instruction in the block is executed all must be" [Coc70]

Error-based adequacy criteria focus on the detection of domain faults, i.e., misplaced subdomain boundaries. To detect the misplacement of a subdomain boundary, test values are derived from path predicates. Such values are referred to as boundary (or *ON*) test values, as they lie on the boundary between two subdomains. Boundary values always belong to one of the adjacent domains and the domain containing the boundary values is referred to as the *domain being tested*. In order to detect a boundary shift within the domain being tested (the domain being tested is smaller than it should be due to a misplaced boundary), values in its *adjacent domain* (*OFF* test values) also need to be tested. Error-based criteria provide upper bounds for undetected domain faults under the assumption of continuous input domains and linear borders if ON and OFF values are carefully chosen. This considerate selection of ON and OFF value combinations has been the main focus of error-based adequacy criteria [WC80; CHR82].

Three common criteria are $N \times 1$, $N \times N$, and $V \times V$, where N and V are integer variables. The first number in the notation specifies the number of ON tests and the second the number of OFF tests to be executed. The value N denotes the dimensionality of the input of the SUT. The value V denotes the number of vertices resulting from domain border intersections, if domain borders are geometrically interpreted as hyperplanes and lower dimensional structures in the N-dimensional input space. While error-based testing provides analytical guarantees for the detection of domain faults (cf. White and Cohen [WC80] and Clarke, Hassell, and Richardson [CHR82] for the discussed criteria), the approach makes a number of assumptions that limit its applicability.

1. The analytical guarantees only hold if coincidental correctness is excluded by assumption.

2. Borders are linear. If the borders between adjacent subdomains are non-linear, the analytical guarantees provided by the discussed criteria do not hold and the provision of similar guarantees would require an "inordinate number of test points" [CHR82] in the general case.

3. Input domains are continuous. If they are not, test points cannot be chosen from subdomains without additional analysis. Borders could, for instance, lie within undefined "gaps" of the input space, in which case no border test points could be selected and the above discussed adequacy criteria could not be fulfilled. Furthermore, undefined spots in the input domain would intuitively require additional tests for adjacent values, thereby increasing the required numbers of tests.

4. The input domain can be partitioned into a reasonable number of subdomains. If each input value was treated as its own subdomain, the $N \times 1$ and $N \times N$ criteria could only be fulfilled for functions of dimensionality 1.

These assumptions severely limit the practical applicability of error-based criteria for a large faction of modern software systems.

4.2.2.3 Fault-based Adequacy Criteria for Correctness Testing

In contrast to structural and error-based adequacy criteria, fault-based adequacy criteria are based on the capability to detect behavioral deviations in programs, so-called *mutations*. Under the assumption that mutations resemble actual software faults closely, fault-based adequacy criteria closely approximate fault detection capabilities of tests. The downside of fault-based adequacy is that tests need to be evaluated against a usually large number of automatically created mutants. Furthermore, mutation testing cannot be entirely automated in the general case. The general undecidability of program equivalence[3] directly implies the necessity of human intervention (1) to avoid the execution overhead for equivalent mutants and, more importantly, (2) to decide if a mutant that was *not* identified by the tests is functionally equivalent to the original program. Interface mutation [DMP+01] is a fault-based adequacy criterion focused on integration testing. Interface mutation employs data corruptions at software (sub-)system interfaces similar to the debug FI fault models in this thesis. While interface mutations are an effectiveness measure for integration tests, the effectiveness metrics proposed in this thesis are intended to rate the effectiveness of mechanisms similar to interface mutations.

4.2.2.4 State-based Adequacy Criteria for Correctness Testing

The motivation for the development of state-based criteria [KK07] is to combine the behavior-focused nature of fault-based criteria with the simplicity of measuring structural criteria. For this purpose, tests and test suites are assessed according to the fraction of covered *output defining statements*, where an output defining statement is one that is part of the dynamic program slice (see [Tip95] for an overview) of the test assertion, the variables it references, and the chosen test input. The test assertion checks the result of the SUT's execution by comparing its output (or a fraction thereof) against the expected output (fraction) derived from the SUT's specification. The last defining statement of a variable in the SUT that is within the dynamic slice of such an assertion is influencing the SUT's external state and must be considered a possible contributor to a failure. As a fault is defined as necessary but insufficient precondition for a failure, fault detection requires failure observation. The larger the fraction of the checked external state is, the likelier is the observation of a failure and, hence, the detection of a fault.

4.2.2.5 Input-based Adequacy Criteria for Security Testing

Whereas structural and error-based adequacy criteria focus on syntactic properties of the SUT and fault- and state-based criteria on the tests' ability to identify incorrect SUT behavior, a third possibility is to state adequacy criteria based on the coverage of the SUT's input space. Full coverage of all possible SUT inputs during test is

[3] Program equivalence requires both programs to halt (respectively, not halt) on the same input and thereby necessitates a solution to the halting problem.

referred to as *exhaustive testing* and is commonly considered impracticable [Hua75]. Tsankov, Dashti, and Basin [TDB13] propose an input-based adequacy criterion called *semi-valid input coverage* (SVCov) for fuzzing. Instead of relating the number of test inputs to the number of all possible inputs to the SUT, the authors propose to relate the number of test inputs to the number of semi-valid inputs, which are considered particularly effective for fuzzing. Semi-valid inputs are inputs that violate a single input constraint, where input constraints specify requirements for inputs to be deemed valid. An input that fulfills all input constraints of a SUT is valid. The rationale behind the authors' focus on inputs that are only "slightly off" from valid input is that completely invalid inputs are likely to be rejected if only a single input constraint is validated properly by the SUT. Slightly off inputs are more likely to pass the first input validation barrier of the SUT and reveal security issues.

4.2.2.6 Discussion

In order to determine test adequacy, a large variety of practical measures to approximate the effectiveness of correctness tests have been proposed. Structural test adequacy criteria relate test coverage to elements of the control flow graph, which represent syntactic elements of programs. While structural criteria imply modest measurement effort, they have several drawbacks. Full coverage may not always be achievable, even by exhaustive testing. Moreover, arbitrary numbers of present faults in a software may remain undetected despite high structural coverage if the test cases do not activate faults or fail to propagate the effects of activated faults to the checked SUT output (coincidental correctness). Error-based adequacy criteria relate test coverage to path predicates and the subdomains of basic block sequences they define. While they suffer from the same weaknesses as structural criteria, the literature on error-based criteria explicitly states the assumptions under which these issues do not occur. Full coverage may not be achievable with error-based criteria if subdomains are too small or not executable due to contradicting predicates in the path predicate. Differing from the purely syntactic measures used in structural and error-based criteria, fault-based adequacy criteria focus on the ability to identify behavioral deviations introduced by source code mutations. While fault-based criteria arguably have a smaller semantic gap between what they measure and what they are intended to measure, the actual extent of this gap highly depends on how representative mutations are of real faults. Similar to structural and error-based criteria, full coverage may not be reachable if mutants are created that are equivalent to the original program. However, determining program equivalence is considerably harder than identifying dead code or contradicting branching conditions, as it requires comprehension of the entire program, and cannot be automated in the general case. Furthermore, fault-based criteria entail considerable run time overhead, as the entire test suite has to be evaluated against a usually large number of mutants.

To fill the gap between behavior-based, but difficult to measure, fault-based adequacy criteria and easy to measure, but purely syntax-based, adequacy criteria, state-based criteria that assess the covered fraction of the SUT's external state have been

proposed. The dynamic slice of the test assertion used to identify the set of output-defining statements that *should* have been checked by the assertion depends on the SUT's code structure, its external state, and the chosen test input. The covered fraction of the external state depends on the test assertion. As the test assertion is commonly derived from a *test oracle*, such as the SUT's specification, there are two possibilities for imperfect state coverage: (1) The oracle is flawed and does not specify relevant success/failure conditions. (2) The test assertion does not accurately resemble the oracle. While the test assertion is an important part of any test, an extensive check of the SUT's external state alone does not determine test effectiveness in the general case. Even if the external state of a SUT is always thoroughly assessed and perfect state adequacy is achieved, this does not indicate the extent by which the SUT's specification has been covered. If all tests happen to execute the same path, defects in other paths may remain undetected despite perfect state coverage. Therefore, the achievement of perfect state adequacy does not necessarily imply proper testing, but its absence implies test deficiencies. Consequently, state-based criteria could be used in debug FI to assess failure detectors, but do not reflect FI effectiveness well.

SVCov derives test effectiveness from achieved input coverage of a specific input space subset that is considered more effective for finding security vulnerabilities issues than other subsets. The intention of finding security vulnerabilities is similar to the goal of this thesis, as security issues are mostly related to unspecified (undesired) behavior of a SUT, rather than incomplete implementations of the SUT's specification [WT04] (cf. Section 1.1.2). Any reasonable restriction of the SUT's input space to a known effective subset requires detailed knowledge (or assumptions) about the interaction between the input generation process (as specified by fault models in the case of debug FI) and the SUT. While this may be feasible for individual fault models, such as fuzzing, it is highly SUT-dependent for other models. Bit flips, for instance, have been shown to be most effective if applied to bits at both "ends" of considered data words [JSM07b]. However, this sensitivity is due to data representations and the meaning of individual bits in these representations. As data representations commonly differ for different platforms, so do the input subsets that are expected to be effective as injections.

In order to provide widely applicable measures for the effectiveness of debug FI, structural metrics are proposed in this thesis. The chosen syntactic properties of the SUT are defined on the level of interfaces and services that SUTs implement and should apply for most software components and compositions. Moreover, the proposed metrics have the same benefits as other structural metrics, i.e., their measurement entails modest overhead. Similar to other structural metrics, they violate the applicability, anticomposition, and antidecomposition properties postulated by Weyuker [Wey86] (cf. Section 4.2.2.1). A more severe concern about the adoption of structural metrics is the semantic gap between syntactic properties and SUT behavior. To address this issue, I include failure mode observations in the definition of structural coverage, similar to the state-based adequacy criterion discussed in Section 4.2.2.4, which combines structural coverage of output-defining statements and their occurrence in test assertions.

4.3 Fault Model Efficiency Metrics

To capture the efficiency of fault models in injection campaigns that utilize them, I propose a set of four different metrics. 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 for 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).

The proposed metrics are defined in a way that reflects the considered failure mode of the SUT, i.e., most of the metrics provide measures for each failure mode individually. 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 campaign.

4.3.1 Metric 1: IUT Coverage

The *IUT Coverage* of a fault model is the extent to which it enables the identification of non-robust services provided by the SUT. A service of the SUT is non-robust, also termed *vulnerable* with respect to a certain failure mode if there is a known injection run targeting this service that results in a respective SUT failure. A model's coverage cannot directly be used for comparisons across different SUTs, because no relative measure can be established due to the unknown total number of vulnerable services. It may, however, be used for the comparison of different fault models applied in robustness assessments of the same SUT. *Unique coverage* denotes the number of vulnerable services detected by only one model among all compared fault models.

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 $S_t^{\mathrm{IUT}}(c)$ denote the set of all services in the IUT that are targeted during campaign c with* $\mathrm{IUT}(c)$ *denoting the IUT targeted in c. The* IUT coverage $\mathrm{cov}_{\mathrm{fm}}^{\mathrm{IUT}}(c)$ *of this campaign with respect to* fm *is then defined by the cardinalities of V_{fm} and $S_t^{\mathrm{IUT}}(c)$ as*

$$\mathrm{cov}_{\mathrm{fm}}^{\mathrm{IUT}}(c) = \frac{|V_{\mathrm{fm}}(c)|}{|S_t^{\mathrm{IUT}}(c)|} \tag{4.1}$$

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*

$$\overline{\mathrm{cov}}_{\mathrm{fm}}^{\mathrm{IUT}}(c_i) = \frac{|V_{\mathrm{fm}}(c_i) \setminus \bigcup_{\substack{c_j \in C \\ j \neq i}} V_{\mathrm{fm}}(c_j)|}{|S_t^{\mathrm{IUT}}(c_i)|} \tag{4.2}$$

I have defined coverage to reflect the targeted IUT to prevent structural properties, such as the number of services an interface comprises, from affecting the metric. This information is lost when coverage is defined globally for all services that the SUT uses for interaction. Consider, for instance, the case where a SUT 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. While this is perfectly valid if all interfaces are equally used and important in the SUT's intended application scenario, it masks important information in cases where this is not the case. Often only a fraction of OTS systems' functionality is required and used in specific application scenarios and it would be misleading if coverage indicated preference for a model due to an interface that is actually never used, while a relevant interface remains completely uncovered.

If an injected fault triggers multiple internal faults, the effect of one internal fault may be masked by the effect of the other one 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. This property is particularly relevant for the evaluation of higher order fault models presented in Chapter 5. As coverage denotes the ability of a fault model to trigger present robustness vulnerabilities, it is a benefit-oriented metric and thus preferably maximized.

4.3.2 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' 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 assessment, the obtained injection efficiencies have to be weighted according to (application-specific) robustness requirements of the SUT'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*

$$\mathrm{ie}_{\mathrm{fm}}(c) = \frac{f_{\mathrm{fm}}(c)}{r(c)} \tag{4.3}$$

4.3.3 Metric 3: Average Execution Time

The average *execution time* of an injection run shows how efficiently a robustness assessment can be performed with a specific model. Since an absolute measure (for an entire assessment) 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 in the detector and monitoring components of the FI setup. 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 lesser 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*

$$\mathrm{et}_{\mathrm{fm}}(c) = \frac{\mathrm{cet}_{\mathrm{fm}}(c)}{f_{\mathrm{fm}}(c)} \tag{4.4}$$

4.3.4 Metric 4: Implementation Complexity

The *implementation complexity* of a fault model indicates the required effort to implement 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 to the model's requirements or properties. If, for example, the injection mechanism of one model requires the

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

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.

I have considered several approaches for estimating the implementation effort *a priori* on the basis of functionality-related properties, e.g., *IFPUG function points* [ISO03b] and *COSMIC full function points* [ISO03a], but none of them seems to be sufficiently accurate without experience in terms of either expert knowledge or statistical data on previous projects. In lack of practical alternatives, I propose to use *physical SLOC* to measure implementation complexity a posteriori. A number of different definitions and counting policies exist for physical SLOC and I follow the definition applied in David A. Wheeler's SLOCCount tool [Whe], i.e., "non-blank, non-comment lines". Being a SLOC metric, physical SLOC exhibit 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, physical SLOC is chosen as a more easily applicable alternative. Restrictions to the validity of measurement comparisons based on SLOC metrics apply if they are performed across different developers, implementation languages, or develpment environments and tools. If these are the same for all compared models, SLOC metrics impose no restrictions for cross-model comparisons. In the targeted scenario, the application of different fault models with one given injection framework maintained by one (or few) evaluators is being considered.

Another a posteriori measure used to compensate the lack of a direct functional measure is McCabe's *cyclomatic complexity* [McC76]. While physical SLOC measure implementation complexity as the amount of 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 (1) the physical SLOC sum* $\mathrm{ic}_{\mathrm{pSLOC}}$ *of all related source code files and (2) the sum of the cyclomatic complexities of all associated functions* $\mathrm{ic}_{\mathrm{cyc}}$.

Physical SLOC fulfill COCOMO's [BCH+00] 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.

The definition of implementation complexity does not include efforts for instrumenting software components or for setting up the SUT's OE for the assessment. The reason is that many modern fault injection frameworks perform instrumentation for fault injection online to enable more flexible injection triggers. In order to inject a

transient fault into a SUT server component after a certain time interval, for instance, the SUT server needs to be modified during operation in the course of the evaluation. Such instrumentation overhead is, therefore, captured and reflected by the execution time metric. Any generic instrumentation is usually automated with SUT- and OE-specific tools and its overhead, thus, rather depends on the assessed SUT and its OE than the applied fault model. Furthermore, efforts related to the installation, integration, or configuration of third-party fault models are not considered, as fault injection frameworks cannot integrate existing fault model implementations from other frameworks. The integration of new fault models, thus, require re-implementation, which is reflected by the definition of the implementation complexity metric. Efforts related to the initial setup of the injection framework are also not considered, because they depend more on the actual framework than on a specific fault model.

While SLOC-based complexity metrics are often criticized for their inaccuracy, surprisingly few alternatives exist that arguably provide better accuracy. In lack of better alternatives, the implementation complexity metric is also SLOC-based and augmented with cyclomatic complexity to compensate for the limitations of purely SLOC-based metrics to a certain degree. Both physical SLOC and cyclomatic complexity provide ratio scale measures. Implementation complexity is a cost-oriented metric and, thus, preferably minimized.

4.4 Metric Application

I have validated the applicability of the proposed metrics, applying four different fault models in a robustness assessment of the Windows CE 4.2 kernel. This section discusses the experiment setup and presents the results that were obtained using the previously introduced metrics.

4.4.1 Experiment Setup

Figure 4.1 provides an overview on the experiment setup. The SUT is the OS kernel of Windows CE .NET 4.2. The SUT server components targeted for injection are device drivers, which are used by the SUT to access the system's hardware. The OS's driver interfaces are targeted as IUTs for evaluation, as drivers constitute a major cause for OS outages (see Section 3.2 for a discussion).

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

Test applications constitute the SUT clients intended to trigger the execution of targeted services. Test applications that were specifically designed to trigger executions of the targeted drivers are used as the workload these clients implement.

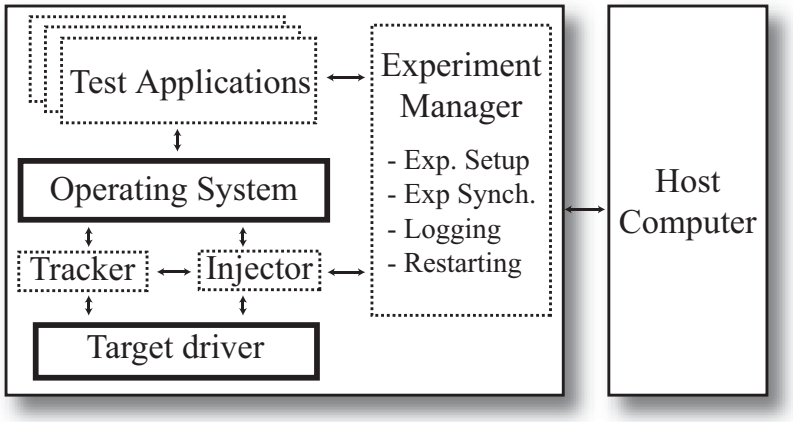

Figure 4.1: Setup for debug FI assessments on the Windows CE .NET 4.2 kernel, adopted from [JSM07a]

The fault injectors are implemented as software wrappers located between the SUT and the targeted SUT servers, in order to intercept mutual service invocations. Faults are injected either by corrupting parameters of intercepted service invocations or by modifying drivers' binary images.

Using this setup, We evaluate the performance of four different fault models for a robustness assessment of the SUT are comparatively evaluated using a modified version of the FI tool from [Joh08]. Three models have been used previously for extensive experimentation [JSM07b; JSM07a]. The fourth model was additionally implemented to enable comparisons of models with differing fault locations, such as in [MBD+06]. The applied fault types are representative for a large class of fault models (cf. Table 2.1 in Section 2.4) spanning:

Parameter Bit Flips (BF) A parameter of an intercepted service call in the IUT 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 SUT servers to the IUT. Injections of these faults are frequently performed at the IUT for efficiency considerations.

Data Type Dependent Parameter Corruption (DT) A parameter of an intercepted service call in the IUT 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 system 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. Data type fault models have proven effective for revealing robustness issues in robustness testing approaches (cf. Chapter 2).

Parameter Fuzzing (FZ) Parameters of intercepted service calls in the IUT are replaced by random values, uniformly selected across all possible values for the con-

sidered parameter's bit length. 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 SUT 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.

The conducted comparative study is restricted to fault models with equal fault timing properties in order to single out the effects of differing fault types and locations more clearly. I deliberately chose to inject transient faults occurring on the first call to a targeted service in order to minimize the run time of the comparison.

The *experiment manager* component controls the execution of experiments. While the interceptors (i.e., tracker and injector in Figure 4.1) are monitored for experiment progress, the SUT (operating system) and its interacting components (test applications and target driver) 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. The implications of this practice are discussed in greater detail in Chapter 5.

We consider four different SUT failure modes commonly considered in related literature (cf. [KSD+97; JSM07b; JSM07a]).

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 indication of the error is given. Incorrect results returned by an OS service according to its functional specification fall in this category. These types of failures, also referred to as *silent data corruption* (e.g., [LPG+14]), correspond to the *silent* failure mode in the CRASH scale (cf. Chapter 2). As there is no indication of failure by the OS service, clients use returned data as if it were correct. This often leads to erroneous behavior of the client, as data obtained from the OS is inconsistent with data provided to the OS and the OS service specification. In the experimental setup, AE failures are detected by executable assertions that correspond to the OS API specifications and the data provided to API services to detect these failures early.

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 in an obvious manner. The detectable effect is either the abnormal termination or an absence of intended termination of the workload application, where the effect is limited to a single application in the system. In either case a client component in the system becomes unresponsive as a consequence of the

OS's behavior. AH failures are detected by a watchdog timer for clients that signals detection if an application's execution time is significantly exceeded.

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 hardware reset. SC failures are detected by a watchdog timer in the experiment controller that is external to the OE.

4.4.2 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.

Three of the proposed metrics (IUT coverage, injection efficiency, execution time) were directly derived from the same data that was collected for evaluating the robustness of the SUT. Additional tools were used to assess the implementation complexity of the implemented fault models. Physical SLOC were counted using SLOCCount 2.26 [Whe]. McCabe's cyclomatic complexity was measured using SourceMonitor 2.5 [Sof]. Following an overview of the experimental results for the four targeted metric types, I discuss their implications in Section 4.5. The results were obtained by conducting more than 300 injections per model and driver.

IUT Coverage The obtained coverage values are grouped by detected failure modes in Tables 4.1 to 4.3. For each IUT (DDK, CORE, NDIS, EXP), the coverage $\mathrm{cov}_{\mathrm{fm}}^{\mathrm{IUT}}$ and unique coverage $\overline{\mathrm{cov}}_{\mathrm{fm}}^{\mathrm{IUT}}$ are given in the tables according to Definition 1. The presented numbers reflect the fraction of all services in the respective IUT that were covered by each model. The first column of Table 4.1, 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 IUT 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 4.2 illustrates the cumulated injection efficiencies for each model and every targeted driver. Regarding the SC failure mode,

Table 4.1: 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 4.2: 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 4.3: 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

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 SUT server, the highest efficiency values are either obtained for the Fuzzing model or the SEU model. 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 4.4. 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 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.

If the execution time metric is applied as proposed in Section 4.3, 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

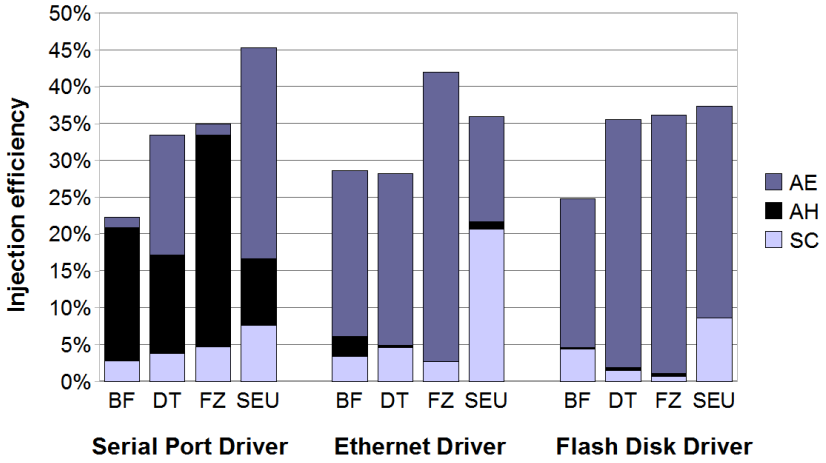

Figure 4.2: Comparison of the applied models's injection efficiencies for each targeted driver

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 4.5 provides an overview on the assessed implementation complexity of the models. From both the physical SLOC 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 costliest models in terms of execution time (SEU and FZ) are of considerably low implementation complexity and the least costly model in terms of execution time (DT) caused the highest implementation effort. However, BF exhibits low overheads in terms of both execution time and implementation complexity. It is worth noting that the use (and obtained rankings) of this metric are bound to the specific FI tool being used. However, the use of different tools for the same SUT are a rare occurrence, which renders the metric applicable for most debug FI scenarios.

4.5 Metric Utility

In the following the obtained results are used to demonstrate the utility of the proposed metrics by comparing the applied fault models. Table 4.6 gives a simplified overview of the results, assuming that all SUT failure modes and IUTs are of equal interest to the tester. 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.

Table 4.4: 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 4.5: Implementation complexities

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

Parameter Bit Flips (BF) The BF model 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 entails modest overhead.

Data Type Dependent Parameter Corruption (DT) Although it manages to identify a few unique AE vulnerabilities, the DT model provides average to poor coverage compared to the other investigated models. Moreover, DT entails the highest implementation complexity among the used models. This result is intuitive, since separate logic is required for each considered data type used for data exchange in the OS/driver (SUT/server) interface, leading to a higher number of physical SLOCs 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

Table 4.6: Fault model ranking obtained from the conducted assessment

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

Legend: ⋆ ⋆ ⋆⋆ Poor, ★ ★ ★ ★ Good.

than any other model in a given amount of time. Although the physical SLOC 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, FI 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 physical SLOC with a cumulated cyclomatic complexity of 262 in the applied FI tool, in addition to about 50 lines of assembly code. However, if binary code mutations and run time patching are natively supported by a chosen FI 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 (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, a combined metric displaying the number of observed failures per unit of evaluation time can be derived. 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 4.3 show that, when the metrics are combined, the ranking for overall failure stimulation and also the failure mode distribution from Figure 4.2 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 be further quantified using the DT and BF models as time and available resources permit. From Figure 4.3, we also see that a general preference of IUT injections over SUT 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 chapter are intended to guide fault model selections for robustness evaluations. The results of the conducted 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. The proposed coverage criterion approximates the detection effectiveness of robustness issues by structural coverage of interface services of a SUT and classified SUT behavior (failure modes). As the total number of services with robustness issues is unknown, the total number of services per interface are used to normalize the obtained counts of vulnerable services. As a consequence, perfect coverage is not achievable if any service of an interface is perfectly robust. Therefore, the proposed coverage metric violates the applicability property postulated by Weyuker [Wey86]. As discussed in Section 4.2.2, the violation of this property by a large variety of practical adequacy criteria does by no means hamper their effective application in successful software projects. The proposed metric is sound, as it is based on the number of misbehaving services found, which is the

defined intent of debug FI. If a model is identified as effective by the coverage metrics, it is capable of identifying non-robust services.

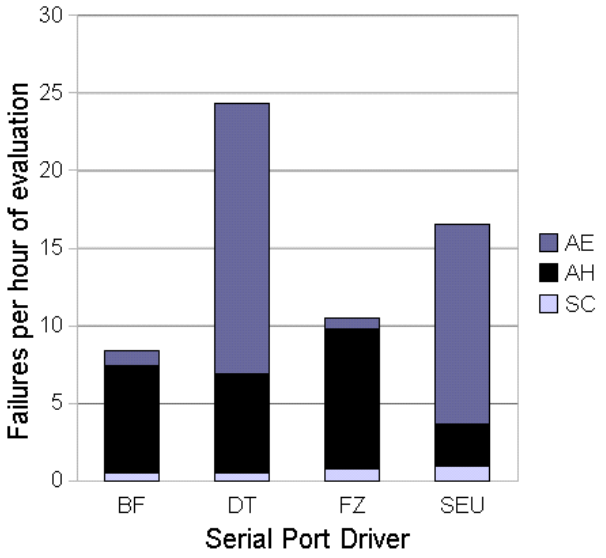

Figure 4.3: Cumulated failures of the serial port driver per hour of evaluation time

Caveats The presented approach has some limitations that testers should be aware of for objective usage. The proposed metrics and measurement approach is restricted to assessment methodologies that follow the system model outlined in Chapter 3. The introduced model assumes explicit interactions of the SUT with its environment via interfaces and, thereby, excludes certain classes of robustness assessment approaches, such as emulations of hardware-induced software errors *in* the SUT. Certain classes of SUTs that heavily rely on implicit interactions with their runtime environments can only be used if the implicit interactions can be intercepted for injections or monitoring.

The applicability of the approach has been demonstrated in a case study using a specific SUT and specific selections of SUT servers and clients. The validity of fault model rankings obtained from the results of this case study is restricted to the particular experimental setup, including the configuration of the OE and the selection of failure modes to detect. For instance, we have seen that the resulting injection efficiency measures do not solely depend on the applied fault model, but also on the targeted driver. 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 SUTs, as this was neither targeted by the presented fault model selection approach nor evaluated in the presented case study. However, as mentioned in the metrics introduction in Section 4.3, the main concern of this chapter is to provide a method to comparatively evaluate contending fault models for a *given* robustness assessment, i.e., a fixed SUT.

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. While it is compelling to reduce this effort by reverting to simpler robustness testing strategies or simplifying these required pre-assessment campaigns by other means, the attempt to speed up experimentation presented in Chapter 6 shows that such simplifications can easily skew the results if not applied with due care.

4.6 Conclusion

Considering the importance of robustness assessments for OTS software components, this chapter addressed the problem of robustness assessment 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 FI-based robustness assessments in order to guide their selection.

The proposed evaluation metrics cover both cost and benefit aspects of fault model implementations and usage. Except for the implementation complexity of fault models, for which existing measurement tools are referenced, the proposed metrics do not require additional measurements beyond those required for the actual robustness assessments. The viability of the approach has been demonstrated in a robustness evaluation of the Windows CE 4.2 kernel, comparing the efficiency of four commonly applied fault models in a case study.

5 Higher Order Fault Models

While a large number of FI frameworks and a considerably larger number of supported fault models exist, virtually all of them are based on a *single fault assumption*, i.e., interactions of simultaneously occurring independent faults are not investigated. As software systems containing more than a single fault often are the norm than an exception and newer safety standards require the consideration of so-called "multi-point faults" as a consequence of increasingly parallel computing platforms, the validity of this single fault assumption is at question for contemporary software systems. To address the issue and support simultaneous injections, this chapter provides a discussion how independent faults can manifest in a software composition, derives corresponding higher order fault models by fault composition, and demonstrates their utility in a robustness assessment of the Windows CE kernel. The developed higher order fault models prove highly efficient for identifying robustness issues in the assessment. The contents of this chapter is based on a conference paper presented at DSN 2013 [WTS+13].

5.1 The Single Fault Assumption in Robustness Assessments

As previous studies by Moraes et al. [MBD+06] and Johansson, Suri, and Murphy [JSM07b] and, in particular, the results from Chapter 4 of this thesis indicate, the conclusiveness of robustness assessments fundamentally depends on the proper choice of the applied *fault models* that specify how faults manifest in the OE, spanning fault dimensions of timing, duration, and location [CC96; JSM07b].

Despite the large number of fault models behind the various proposed FI frameworks (see Table 2.1 in Section 2.4, and [Nat11] for a more comprehensive overview), virtually all of them consider only injections of a single fault per experiment run. The implicit assumptions driving this design choice usually are that

1. apart from error propagation [ALR+04], no error interactions occur (i.e., all errors resulting from different fault activations are independent) or that

2. such effects are of no or negligible relevance for the outcome of the experiments.

Therefore, experimental assessments with existing tools do not account for the possibility of interactions between multiple faults. However, if the consideration of multiple coincident faults significantly affects the outcome of a robustness evaluation, it would

be a fallacy to disregard them. Given that software systems containing more than a single fault are a common occurrence (e.g., [CGN+13; CYC+01; CDC+03; PTS+11]) and even highly critical systems compliant with the DO-178B functional safety standard have been found to contain between 4 and 170 faults per thousand lines of code (kLOC) [Ger03; KM06], the independence of their effects appears to be a strong assumption. Moreover, as systems become more complex, even exceptionally small fault rates of 0.1 faults/kLOC result in thousands of residual defects in deployed software systems[1].

As a consequence, newer safety standards, such as the ISO standard 26262 for the functional safety of road vehicles, explicitly consider *multiple-point failures*, i.e., failures resulting from the combination of several independent faults [ISO11a]. In order to assess the degree to which systems are vulnerable to such multiple-fault conditions, corresponding testing approaches and fault models are needed.

The goal of this chapter is to (a) conceptually develop a notion of what *fault coincidence* actually implies in the context of FI-based robustness assessments, and (b) to experimentally assess the impact (positive or negative) of its consideration on the assessment effectiveness and applicable overheads. On this background, the chapter contributions are:

Coincidence notion Section 5.3 establishes fault coincidence notions and discusses their applicability within a general model of FI-based robustness assessments for software compositions (cf. Section 3.1).

Higher order fault model design The utility of coincident injections is shown via the design and implementation of three higher order fault models (Section 5.5) complying with the developed fault coincidence notion.

Quantitative evaluation of higher order models The proposed models are quantitatively evaluated and compared against classical fault models using the metrics from Chapter 4.

5.2 Higher Order Fault Impact on FI-based Software Assessments

In cases where multiple independent faults can affect the dependable operation of a software system, the application of tools and processes based on a single fault assumption is likely to generate incorrect or imprecise results. In addition to this obvious impact on the validity of obtained results, the consideration of higher order faults in FI-based assessments may have a number of conceptual and technical implications on experimental assessments, which are discussed in the following.

[1]For instance, premium cars in 2009 contained 100 million lines of code [Cha09].

5.2.1 Likelihood vs. Criticality of Rare Events

Assuming that the probability of some fault f being activated is p_f, the probability of n independent faults $f_1 \dots f_n$ being active at the same time is $\prod_{i \in 1 \dots n} p_{f_i} \leq p_{f_j}$ for any $j \in 1 \dots n$. The activation of n faults is less probable than the activation of an individual fault f_j if at least one other fault has an activation probability less than 1. This makes the co-occurrence of multiple fault activations less likely than individual fault activations. Thus, the risk they impose may be regarded negligible for very low probabilities of occurrence, especially when contrasted with high testing efforts required to cover all possible cases. This resembles the fundamental assumption driving test case reductions in combinatorial testing [KWG04]: Complex fault interactions are so rare that they can be neglected during tests without significantly affecting the outcome in the common case.

However, in system safety engineering, critical failures (*accidents*) are often considered to result from unlikely, rare, and unconsidered combinations of events, which is also reflected by the explicit consideration of multi-point faults the ISO 26262 safety standard [ISO11a; ISO11b]. Due to their improbability, such conditions often escape the focus of testers and are sometimes very difficult to construct in the test environment. Empirical work shows this to apply for critical software failures as well [Hec93; LM03].

5.2.2 Experiment Acceleration vs. Counts

Single fault injections are capable of identifying at most one internal fault per injection run. If a higher order fault injection reveals multiple internal faults at the same time, this implies a considerable speed-up in terms of experiment time required for the identification of the same amount of internal faults. However, the outcome of a higher order fault injection run may also lead to more ambiguous results.

If multiple faults are injected, one activated fault (i.e., an error) can potentially *mask* another fault or error. If, for instance, a fault f_1 leads to a corrupted data value in memory and by activation of a different fault f_2 that particular memory location is never referenced or overwritten before it is referenced, the effect of f_1's activation never becomes visible although it would have if f_2 had not been activated. From such an experiment, it is not possible to determine whether the activation of f_1 would have been tolerated or whether it would have caused the component to fail.

If the injection of a higher order fault results in a system failure, a similar ambiguity can occur: It is unclear whether the activations of *all* injected faults it is composed of are actually necessary to trigger the observed failure. This makes it difficult to identify the internal fault(s) responsible for the failure. Note that this problem is different from error masking, as error masking may lead to an error being tolerated or mitigated unlike the presumed software failure in this case. In both cases, the ambiguity can be resolved at the cost of additional experiments with the single faults that constitute the combined higher order fault. However, this also adds considerably to the total number of overall required experiments.

Even without the potential need for additional experiments to resolve ambiguities, the consideration of higher order faults by itself leads to higher numbers of possible experiments, thereby aggravating test case selection: While the individual injection of n faults yields n experiments, this expands to $\binom{n}{k}$ experiments for the combination of k simultaneously occurring faults, assuming all possible combinations are feasible.

I experimentally investigate the impact of coincident faults on the evaluation efficiency in Section 5.5.

5.3 Modeling Higher Order Faults

The main goal of this chapter is to relax the single fault assumption omnipresent in FI-based robustness assessments by combining several faults of the same or different fault models in a *higher order fault model*. While there are many possibilities to combine faults for injection, not all of these combinations result in an *effective* violation of the single fault assumption. For instance, if the same location and the same time for a parameter fuzzing are chosen, the same parameter is altered twice for the same SUT service invocation. The result would be identical to a single injection, as both modifications would be performed in sequence, even on a multiprocessor system, where memory access for data of certain size is mutually exclusive for each bus cycle due to the shared memory bus. As the resulting fault exposure to the SUT would not differ, such a model is not suitable to assess the validity of the single fault assumption as the SUT is exposed to only one of the injected faults. Fault interference is legitimate (and desired) if it creates previously unconsidered stimuli for the SUT, i.e., interference in the sense of *fault effects* on the SUT. This is the case if either different internal faults of the SUT are activated by the external fault combination or if the same internal faults are activated repeatedly and the resulting errors interfere such that a different error state of the SUT results. Fault interference is undesired if it occurs before affecting the SUT's interface(s). In order to prevent fault interference of the undesired type, the combined faults must remain distinguishable from each other. In the following, I define two *fault coincidence* criteria that higher order fault models must satisfy to achieve this. According to the following definitions, combined faults must differ in either injection time or location (or both) to qualify as higher order faults.

5.3.1 Temporal Spread, Temporal Resolution, and Spatial Coincidence

Definition 5 *An external fault of a SUT is termed* temporally spread *if it covers more than one temporal usage sequence of the SUT.* Temporal usage sequence *may refer to an individual service invocation during an injection run, an injection run, an injection campaign, or an assessment that consists of several campaigns. The choice of the usage sequence is referred to as* temporal resolution. *Faults that are temporally spread, but always target the same injection location, are termed* spatially coincident.

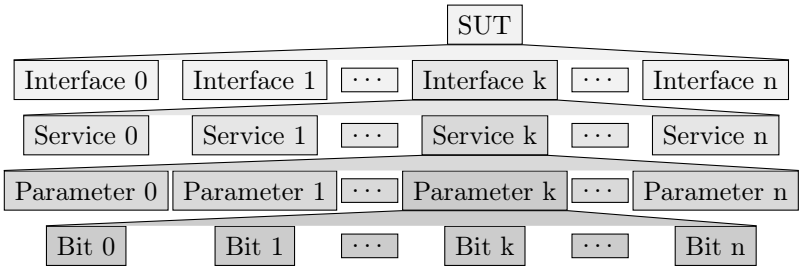

Figure 5.1: Relationship of spatial entities

An example for a temporally spread fault with coarse-grained temporal resolution is the injection into a component that maintains some state across different evaluations, e.g., injections that lead to accumulating corruptions in persistent storage. An example of a fine-grained temporal resolution is the repeated injection of the same external fault into the same service over multiple service invocations.

Temporally spread and spatially coincident fault injections (e.g., the injection of intermittent or permanent faults) are more widely applied than spatially spread and temporally coincident fault injections discussed in Section 5.3.2. They are also applied to investigate fault effect accumulation, often referred to as "software aging". While I do not develop temporally spread fault models in this thesis, I discuss existing work addressing temporally spread injections in Section 5.6.

5.3.2 Temporal Coincidence and Spatial Resolution

Definition 6 *An external fault is termed* temporally coincident *if it covers more than one spatial entity in the SUT's interface during the same temporal usage sequence.* Spatial entity *may refer to a component interface, a service provided by the component, a parameter passed to the component upon service invocation, or a data element of the parameter. The choice of the spatial entity is referred to as* spatial resolution.

Figure 5.1 depicts the relationship between the considered spatial entities. An example for a temporally coincident fault with a spatially coarse-grained resolution is when services of different component interfaces are invoked with illegal parameter values within a single injection run. An example of a fine-grained spatial resolution is the simultaneous modification of two individual bits within the same parameter.

The definition of temporal coincidence uses the the previously introduced notion of temporal usage sequences. As a consequence, the temporal and spatial dimensions (when and where to inject) of higher order fault models are not independent. This restriction is necessary to prevent the aforementioned issues that could cause higher order fault models to effectively revert to single faults (at the cost of higher order models, as discussed in Section 5.2). Depending on the considered temporal resolution (i.e., the choice of the temporal usage sequence), only a subset of spatial entity choices (i.e., spatial resolutions) is possible. If, for instance, an individual service invocation is the temporal scope, only different parameters of that service or multiple locations

within one parameter are possible choices for temporally coincident spatial entities. I develop and evluate higher order fault models for these two cases in the remainder of this chapter.

5.4 Fault Models: From Discrete to Higher Order

In the following, I define three higher order fault models according to the temporal coincidence notion developed in Section 5.3.

The injection framework, for which the higher order models are developed, supports a large number of classical fault models (in the following also referred to as *discrete*) with a variety of injection triggers and latencies. While various injection locations are supported, i.e., injections into device driver binaries as well as injections into service parameter values, we only consider the latter in our evaluation because of the strong effects this difference in location has on the evaluation outcome (see [MBD+06] and Section 4.4.2). For parameter value corruptions, the framework supports three classical discrete fault models.

- Bit flips (BF): An individual bit in the binary representation of the passed parameter value is being flipped, i.e., altered to one if it was zero and vice versa.
- Fuzzing (FZ): A passed parameter value is replaced by a random value of the same binary length.
- Data type-specific (DT): Similar to the fault model applied in Ballista [KD00], a passed parameter value is replaced by a data type-specific boundary value, e.g., MAXINT, MININT, -1, 0, or 1 for the integer data type.

I use the BF and FZ models as a basis for constructing three temporally coincident fault models, i.e., models of faults that are activated within the same temporal usage sequence in different spatial locations. The considered temporal usage sequence are individual service invocations. Higher order faults can then manifest as faults covering distinct parameters of the same service invocation or covering distinct faults within one single parameter of the targeted service call, depending on the spatial resolution.

5.4.1 The FuzzFuzz Higher Order Fault Model

While the FZ fault model mandates the replacement of an individual parameter value of a service call, the FuzzFuzz model applies fuzzing to two distinct parameters of the same service invocation. As mentioned in Section 5.3, fuzzing is only usefully applicable for spatial resolutions lower than the individual parameter level: If multiple fuzzing operations were applied to the same parameter, the resulting injection would not differ from a single fuzzing injection (one parameter value is replaced by a random value). Consequently FuzzFuzz mandates fuzzing *different* parameters of the same service invocation, thereby implicitly restricting its applicability to services that are taking at least two parameters as inputs.

For a service with n parameters $p_1 \dots p_n$, $\sum_{i=1}^{n} 2^{bl(p_i)}$ possible FZ injections and $\sum_{i=1}^{n-1}(2^{bl(p_i)} \cdot \sum_{j=i+1}^{n} 2^{bl(p_j)})$ possible FuzzFuzz injections exist, where $bl(p_i)$ denotes the number of bits (the *bit length*) of the parameter value p_i. Even for the FZ model an exhaustive assessment covering all possible input combinations is impracticable. We restrict ourselves to 30 experiments for each parameter targeted by FZ and each parameter combination targeted by FuzzFuzz. This restriction is based on a stability finding from Johansson, Suri, and Murphy [JSM07b], who report error propagation measures to stabilize for the same SUT after approximately 10 injections.

5.4.2 The Simultaneous FuzzBF Model

Johansson, Suri, and Murphy [JSM07b] report the fuzzing and bit flip models to perform well in robustness assessments of the same SUT considered in this thesis and derive a *composite fault model* from these two models. Despite its compositional character, this composite fault model is not a higher order fault model in the sense of this thesis, as it combines faults on a campaign level with SUT resets between individual runs, thereby adhering to the single fault assumption. I combine these models in a higher order model for injection experiments: Upon invocation of services with two or more parameters, FZ is applied to one parameter value and BF to another. The spatial resolution of the FuzzBF simultaneity is, as for the FuzzFuzz model, different parameters of the same service call. The considered temporal usage sequence is an individual service invocation.

For a service with n parameters $p_1 \dots p_n$, there exist $\sum_{i=1}^{n} 2^{bl(p_i)}$ possible test cases for individual fuzzing injections and $\sum_{i=1}^{n} bl(p_i)$ possible individual bit flips. For the FuzzBF model the amount of possible test cases expands to

$$\sum_{1 \le i \le n} (2^{bl(p_i)} \cdot \sum_{\substack{1 \le j \le n \\ j \neq i}} bl(p_j)) - \sum_{1 \le i < n} \sum_{i < j \le n} bl(p_i) \cdot bl(p_j)$$

This considers that all single bit flip cases for a parameter are contained in the number of possible Fuzzing values. The part before the minus counts the number of combinations for fuzzing any parameter p_i with bit flips in any *other* parameter p_j. Fuzzing can result in any possible value that a parameter can take, including those that would result from single bit flip injections. Therefore, $bl(p_i) \cdot bl(p_j)$ cases are identical among the injections produced when fuzzing p_i is combined with single bit flips in p_j and vice versa. These cases are easily detectable when injection campaigns are defined and, thus, subtracted from the total number of injection runs.

5.4.3 The Simultaneous SimBF Model

As opposed to the FuzzFuzz and FuzzBF models, the SimBF model has a spatial resolution of an individual parameter value in a service invocation: Two bits are flipped in the binary value of the same parameter.

For a service with n parameters $p_1 \dots p_n$, there exist $\sum_{i=1}^{n} bl(p_i)$ possible single bit flips and $\sum_{i=1}^{n} \binom{bl(p_i)}{2}$ possible SimBF test cases. While this may result in large test case numbers for large numbers of parameters or high bit lengths, exhaustive testing is generally feasible. Although multi bit fault models have been applied in previous work (see Chapter 2), they have not been discussed as simultaneous single bit faults and their efficiency has, therefore, not been assessed comparatively. Moreover, as multi bit faults in the literature are mostly applied to simulate hardware burst faults, they usually only consider flips of adjacent bits, whereas the SimBF model considers arbitrary combinations of dual bit flips.

From the differing applicability of the fuzzing and bit flip models to different spatial resolutions, we observe that, for higher order fault models, the fault type and the fault location (what and where to inject) are not independent from each other. If the spatial resolution (i.e., the considered location) of an injection is an individual parameter, simultaneous fuzzing would not differ from non-simultaneous fuzzing as previously discussed. For bit flips instead, it is possible to differentiate between simultaneous and non-simultaneous injections into the same parameter value. The reason is that the non-simultaneous fuzzing and bit flip fault models are defined according to different spatial resolutions.

5.5 Experimental Evaluation

I investigate the impact of the proposed higher order fault models on the assessment efficiency using the metrics developed in Chapter 4. In order to analyze masking effects of higher order faults discussed in Section 5.2, I investigate where combinations of two individual faults lead to an outcome that differs from the outcomes for the individual injections, reflected by the failure class distributions for each spatial entity for which higher order FI experiments are conducted. The obtained distributions are compared against the failure class distributions of experiments with the discrete faults from which the higher order faults have been composed. If fewer failures are observed for higher order faults than for any of the corresponding discrete faults, masking has taken place. If higher failure rates are observed in the higher order cases, I denote this effect as *amplification*. Amplification, therefore, implies that services are quicker identified as non-robust on average, whereas masking implies that higher order models take more experimentation effort on average to identify non-robust behavior. If the aim of a robustness assessment is the removal of robustness issues or the efficient placement of fault-tolerance mechanisms in the system, the occurrence of masking and amplification indicate that both discrete and higher order models should be used for injections to precisely identify the trigger condition for observed failures. In order to obtain reliable results for the masking and amplification analyses of models that involve fuzzing, additional single injection experiments have been performed for each random value used in a simultaneous fuzzing experiment. This is necessary as the application of fuzzing may result in different random values for the single and higher order injections, which would lead to incomparable results and, hence, render the proposed approximations

of masking and amplification invalid. Before going into the details of the developed models and their performance, I briefly describe the experimental setup for the case study.

5.5.1 Experimental Setup

For comparability of results, I apply FI to the driver interface of the Windows CE 4.2 operating system (OS) as previously described in Chapter 4. For readability I briefly summarize the setup. A fuller discussion of the failure mode definitions and other details of the experimental setup can be found in Section 4.4.1. The serial port and Ethernet drivers are used for the assessment. Both drivers export a Windows stream driver interface [MSDd]. The serial port driver imports the CEDDK [MSDb] and COREDLL [MSDa] kernel interfaces. The Ethernet driver imports COREDLL and NDIS [MSDc].

External faults are injected at the kernel interface by intercepting service invocations between the drivers and the kernel, and modifying parameter and return values passed to the kernel. Service invocations at the kernel/driver interface are triggered by synthetic workloads designed to trigger executions of the targeted drivers. The applied failure detectors are capable of detecting four different outcomes of an injection run. Upon fault injection the system may respond with the following:

- *Not Failing (NF)* in any detectable manner
- Failing without violating OS specifications by delivering wrong, yet plausible results (*Application Error, AE*)
- Violating the service specification of an individual OS service upon which the application fails and becomes unresponsive (*Application Hang, AH*)
- *System Crash (SC)*, rendering the whole system unavailable and often necessitating a manual restart.

5.5.2 Experimental Results: Exploring Higher Order Injections

The comparative assessment of all single and higher order fault models is mainly based on interactions of the serial port driver and the OS kernel. A number of additional experiments using an Ethernet driver have been performed for all fault models apart from the SimBF model. As discussed in Section 5.4.3, exhaustive tests with this model are considered feasible and, consequently, an exhaustive campaign has been conducted for the serial port driver. However, this resulted in more than 34 000 individual injection runs taking up more than two months of experiment run time for only this model and a single driver. Thus, the experiments with this model have been restricted to the serial port driver.

5.5.2.1 Coverage and Unique Coverage

Figures 5.2 to 5.4 show the coverage and unique coverage measures for the three considered failure classes (AE, AH, SC) obtained from a comparative evaluation of all six fault models (grouped by the three targeted interfaces) using the serial port driver. The unique coverage is indicated as a fraction of the coverage in the bar diagrams. Neither the FuzzFuzz model nor the FuzzBF model achieve particularly high coverages. This occurs as these models target injections into two distinct parameters and, therefore, (by definition) cannot cover any service taking less than two input parameters, while the other evaluated models can. For the same reason, FuzzFuzz and FuzzBF cannot cover any services of the export interface. Data communicated to the kernel via this interface are return values of services offered by the driver. Since return values are considered as single parameters, FuzzFuzz and FuzzBF cannot be applied for injections in the export interface.

SimBF, in contrast, outperforms all other models in most of the cases, showing coverages of up to 100 % for the CEDDK interface in Figures 5.2 and 5.3, i.e., it identifies a robustness issue in *each* service of this interface. The obtained low coverage values for the simultaneous FuzzFuzz and FuzzBF models indicate that they should not be applied as substitutes for discrete models. This conclusion is supported by a direct comparison of the unique coverages in Figures 5.5 to 5.7. For the COREDLL interface, we see substantially higher unique coverages for vulnerabilities that result in AE or AH failures (Figures 5.5 and 5.6), when the FZ model is applied. However, Figures 5.5 to 5.7 also show that the higher order models cannot be neglected when testing for defects that lead to system crashes (SC), as all three higher order models identify some services as vulnerable that would go undetected by their discrete counterparts. This is supported by results obtained for the CEDDK and export interfaces that show clear polarizations for the different failure modes (Figures 5.8 to 5.11).

5.5.2.2 Injection Efficiency

Figures 5.12 and 5.13 illustrate the injection efficiency in terms of achieved failure class distributions using the serial and Ethernet drivers. The SimBF model achieves an overall injection efficiency of almost 100 % for every interface used by the serial driver. SimBF provokes AE failures particularly well. While the FuzzFuzz and FuzzBF models perform well for the COREDLL interface, they are outperformed by single fault models in the CEDDK interface, when applied to serial driver interactions. Surprisingly, these models outperform the other models for the Ethernet driver both in terms of overall failure probability and in particular AE failure probability.

In general, the results demonstrate increased coverage and high failure probabilities, which means that the increased coverage can likely be gained with a modest number of additional experiments using higher order models. Given these expected benefits, I investigate the costs for these additional experiment runs and the higher order fault model implementations next.

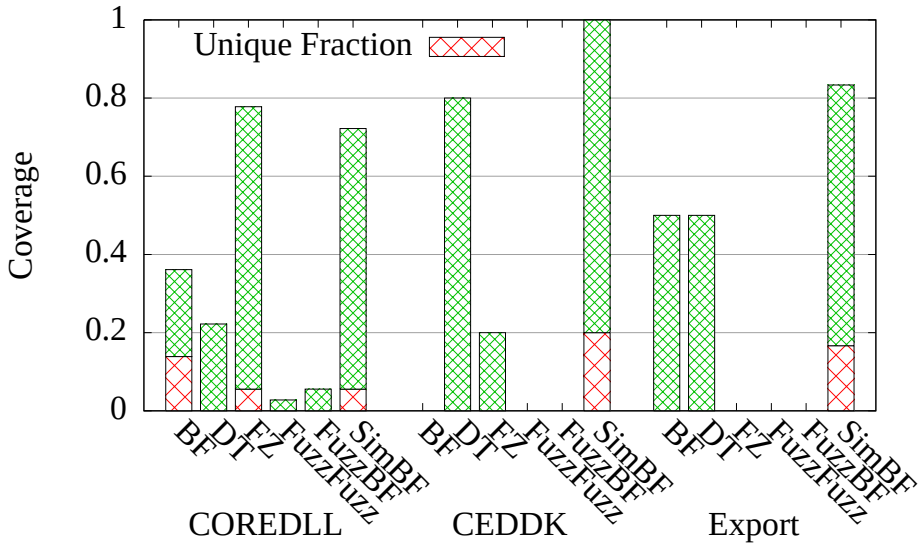

Figure 5.2: (Unique) coverage of AE vulnerabilities using the serial driver

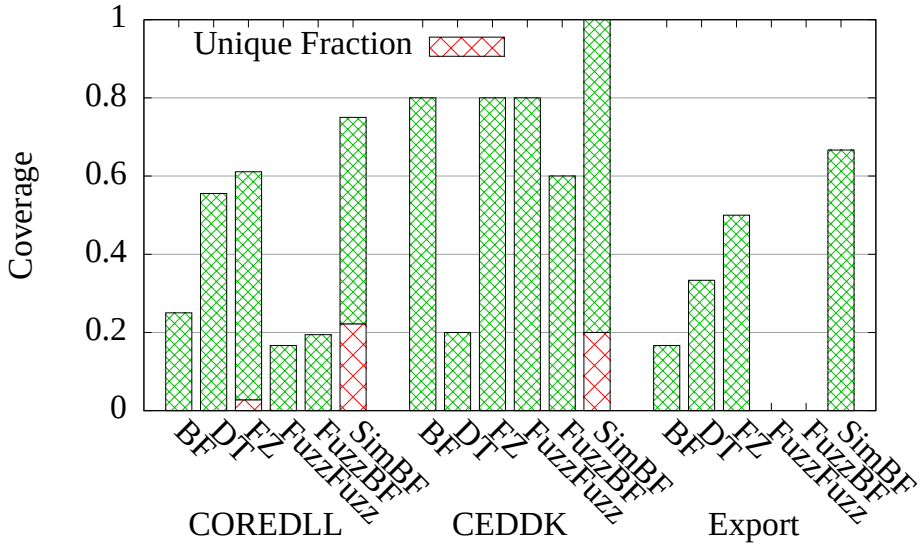

Figure 5.3: (Unique) coverage of AH vulnerabilities using the serial driver

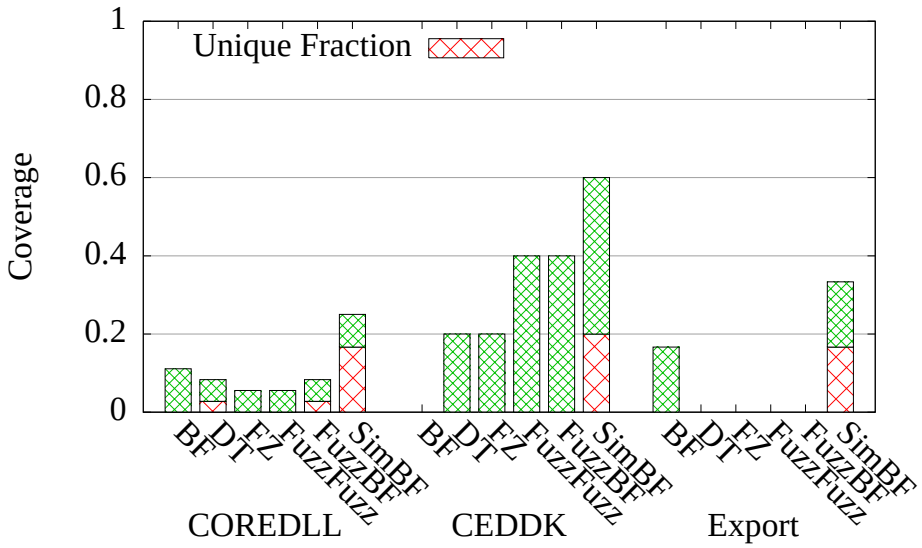

Figure 5.4: (Unique) coverage of SC vulnerabilities using the serial driver

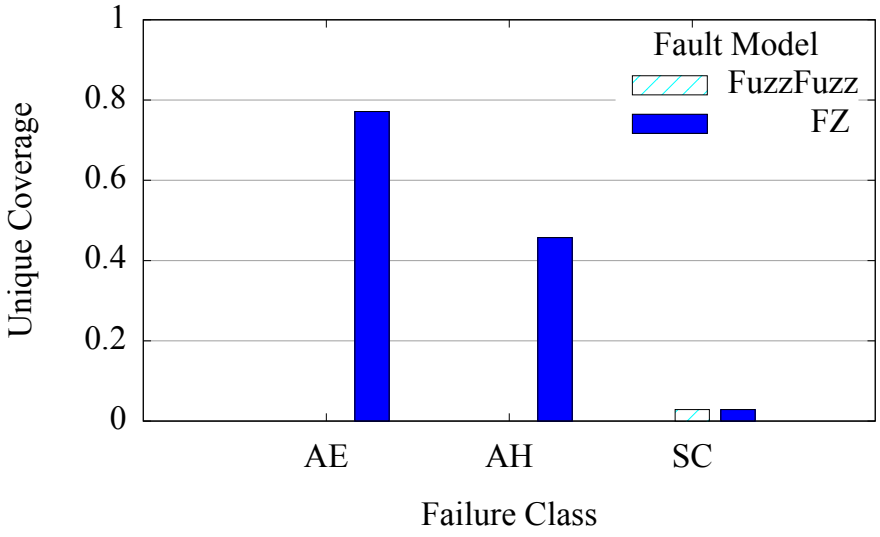

Figure 5.5: Unique coverages of FuzzFuzz & FZ evaluating the COREDLL interface using the serial driver

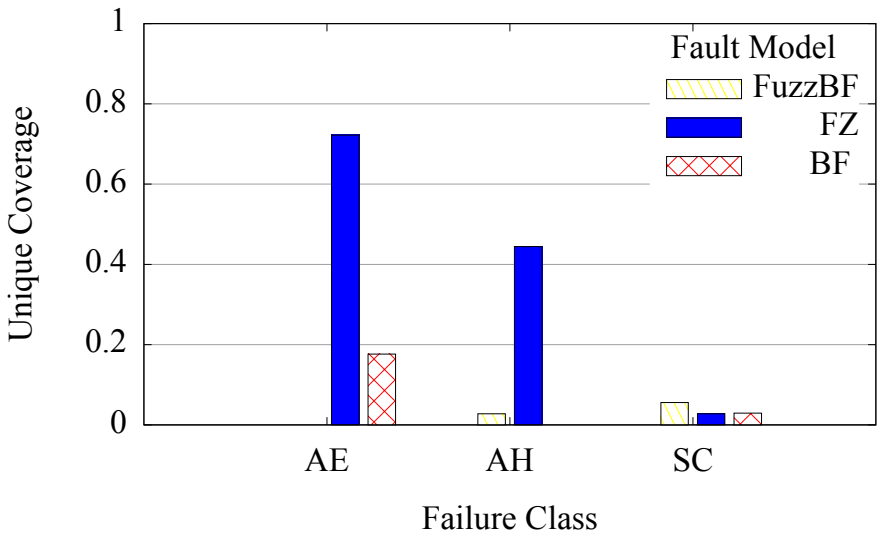

Figure 5.6: Unique coverages of FuzzBF, FZ & BF evaluating the COREDLL interface using the serial driver

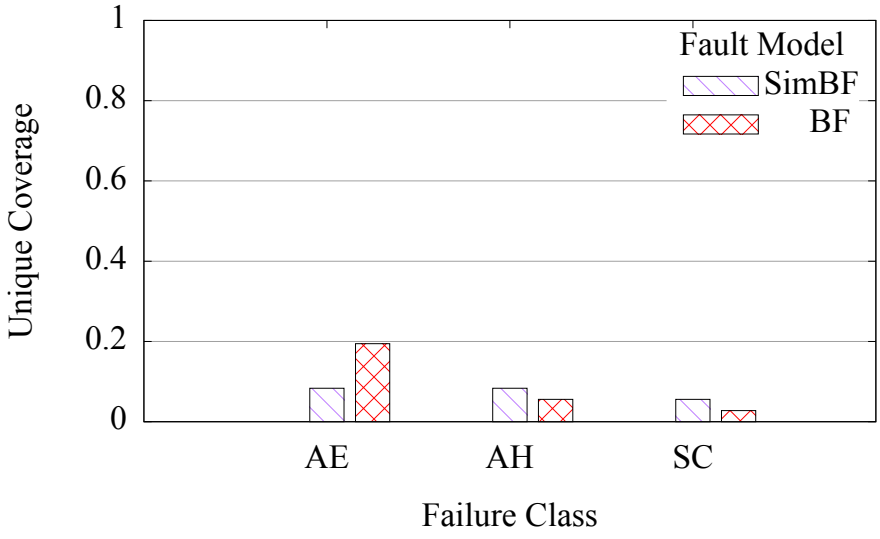

Figure 5.7: Unique coverages of SimBF & BF evaluating the COREDLL interface using the serial driver

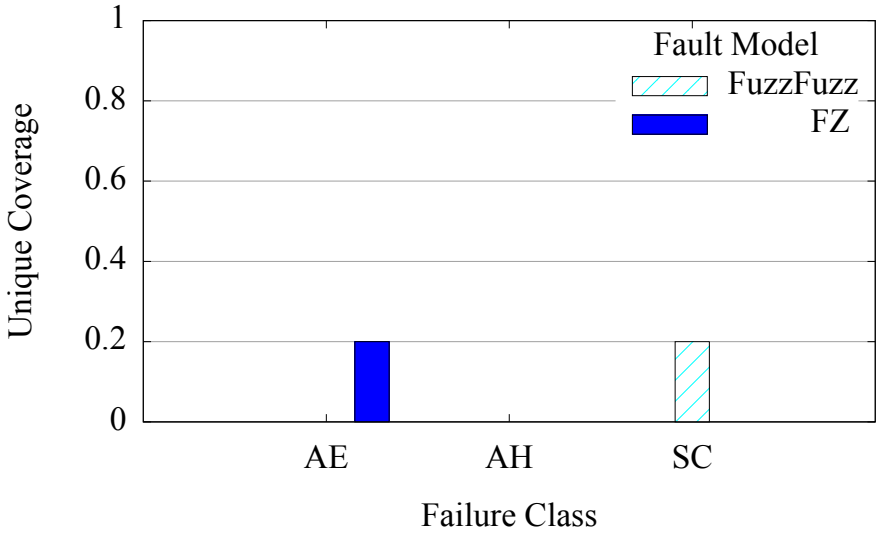

Figure 5.8: Unique coverages of FuzzFuzz & FZ evaluating the CEDDK interface using the serial driver

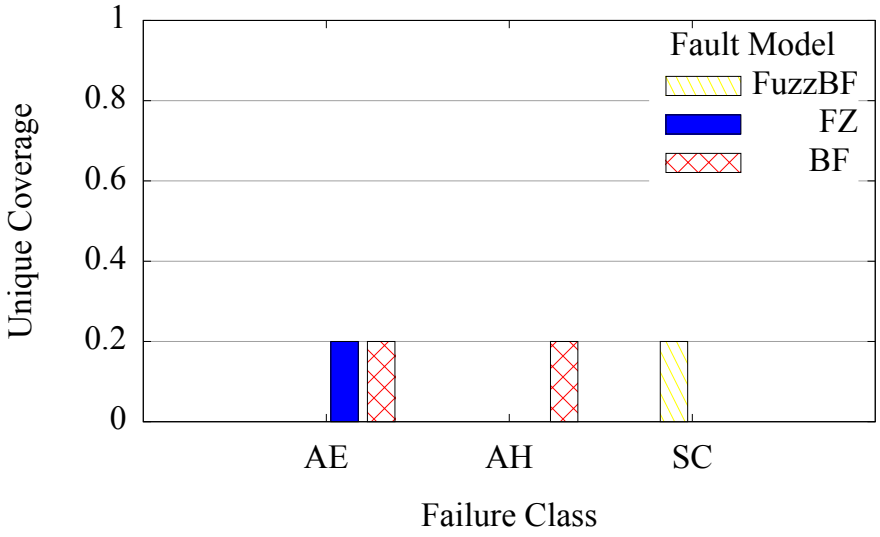

Figure 5.9: Unique coverages of FuzzBF, FZ & BF evaluating the CEDDK interface using the serial driver

Figure 5.10: Unique coverages of SimBF & BF evaluating the CEDDK interface using the serial driver

Figure 5.11: Unique coverages of SimBF & BF evaluating the export interface of the serial driver

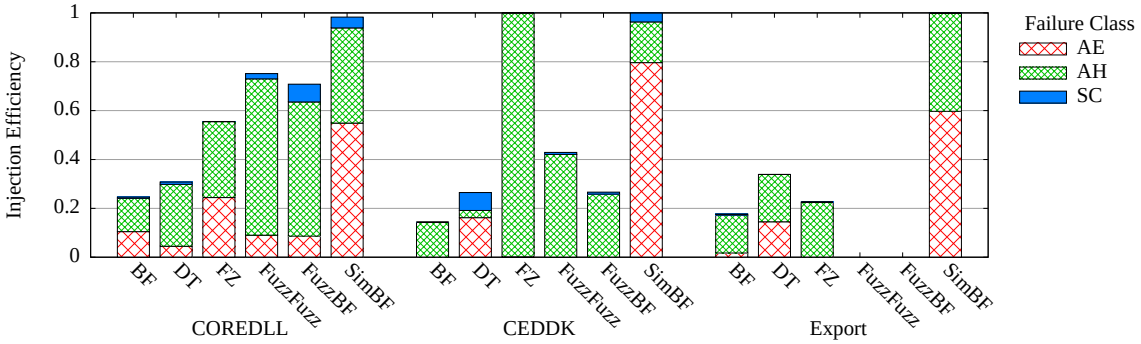

Figure 5.12: Injection efficiencies of experiments using the serial driver

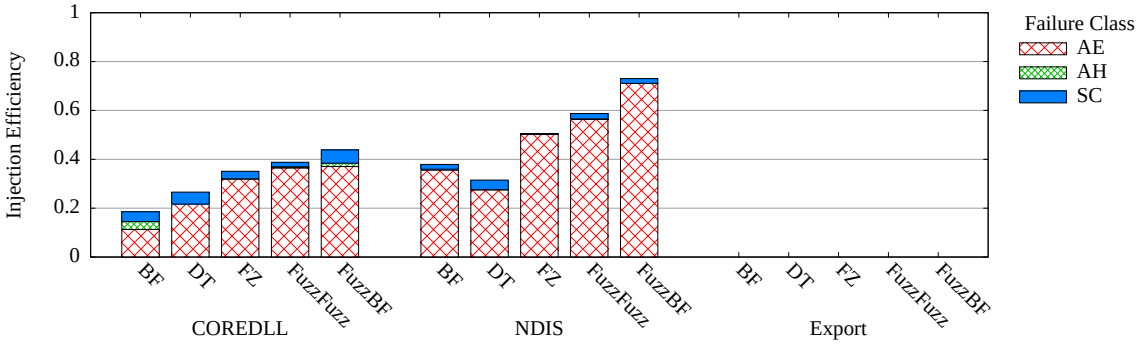

Figure 5.13: Injection efficiencies of experiments using the Ethernet driver

5.5.2.3 Average Execution Time

Figure 5.14 shows the average execution time for injection runs using the serial driver grouped by failure classes. The higher order models always take the longest execution times. As the actual injection overhead from the implementations of the higher order models is negligible, the observed delays likely result from parsing and processing the more complex test case definitions necessitated by these models. The increased execution times for the AH and SC failure classes result from the application of timeouts for detecting these failure types. The diagram shows a relatively stable order for the execution times among the models, except for the SC failure class. The reason for this is that, after a system crash, the system sometimes manages to restart autonomously, in which case the time until the autonomous restart is measured. Otherwise, it requires a manual reboot, in which case the timeout expiration is measured.

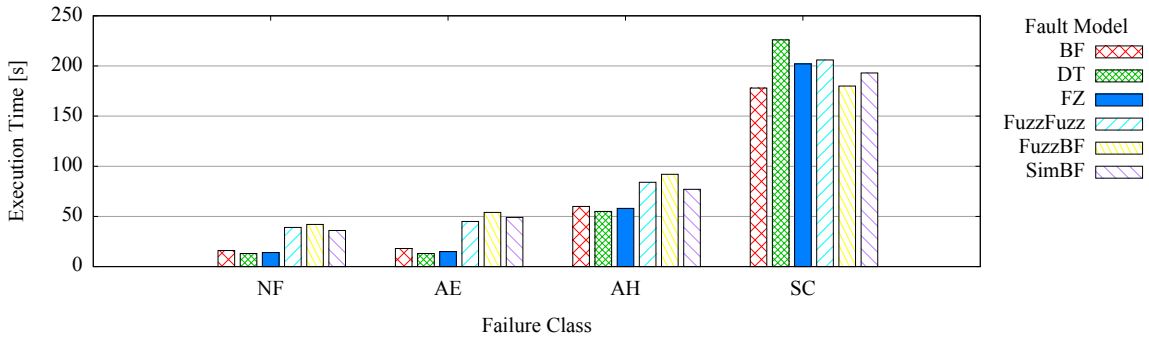

Figure 5.14: Average experiment execution times using the serial driver

5.5.2.4 Implementation Complexity

The implementation complexities in terms of physical SLOC and accumulated cyclomatic complexity are presented in Table 5.1. Being the least complex higher model among the assessed models, SimBF also is the second-least complex model overall and only outperformed by the BF model. The physical SLOC and cyclomatic complexity counts for the FuzzFuzz and FuzzBF models include the respective counts for the FZ and BF models (we assume that these are not previously implemented). The actual mechanism for combining any two existing models for higher order injections into different parameters of the same service invocation accounts for only 105 physical SLOC and an accumulated cyclomatic complexity of 18. In conclusion, higher order fault injections entail moderate overhead for implementation and execution time.

5.5.2.5 Masking and Amplification

Masking and amplification effects highly depend on the individual injection target, i.e., parameter for the SimBF model and service for the FuzzFuzz and FuzzBF models. These effects need to be discussed for each such target individually, and I restrict the

Table 5.1: Implementation complexities

Model	pSLOC	cyclomatic complexity
BF	133	30
FZ	272	58
DT	635	222
FuzzFuzz	377	76
FuzzBF	510	106
SimBF	245	39

detailed discussion to two illustrative examples. Detailed plots for all tested services are available on the public web site for the higher order FI project [DEE]. In summary, the obtained results for the masking and amplification analysis show that both masking and amplification are a common occurrence and that both effects can occur for the same service if different fault models are used or different failure modes are considered.

Figure 5.15 shows an example of masking using the FuzzFuzz model. While the FZ model provokes SC failures in more than 80 % of the injections into parameters 0 and 1 and achieves injection efficiencies of 100 % for both parameters, their combination in the FuzzFuzz model yields an injection efficiency of less than 10 %, all of which are AE failures. Also for combinations with injections into parameter 2, the SC failure causes for the first two parameters do not dominate the experiment outcome.

In contrast, Figure 5.16 shows an example of amplification for the same model targeting a different service. While single fuzzing only results in a modest amount of SC failures in parameter 2, the higher order injections into parameters 0 and 2 result in higher SC failure rates. To avoid sampling effects in the comparisons, the presented results of the masking and amplification analysis are based on additional experiments, as discussed in the beginning of this section.

5.5.3 Discussion

The higher order models add value to the performed evaluation, as they detect service faults that are not covered by other models. This result demonstrates that the single fault assumption, on which many frameworks and studies are based, does not generally hold. Given that many of the systems for which FI assessments are applied are employed in safety-critical scenarios, neglecting possible fault interactions is dangerous and their exclusion should be based on thorough analysis rather than by assumption. On efficiency considerations, the SimBF model outperforms discrete fault models in most cases, which shows that effective higher order injections do not need to entail hight execution overheads. The implementation overhead of all considered higher order models is moderate compared to the discrete models. Therefore, the consideration of higher order faults threatening dependability may well be worth the

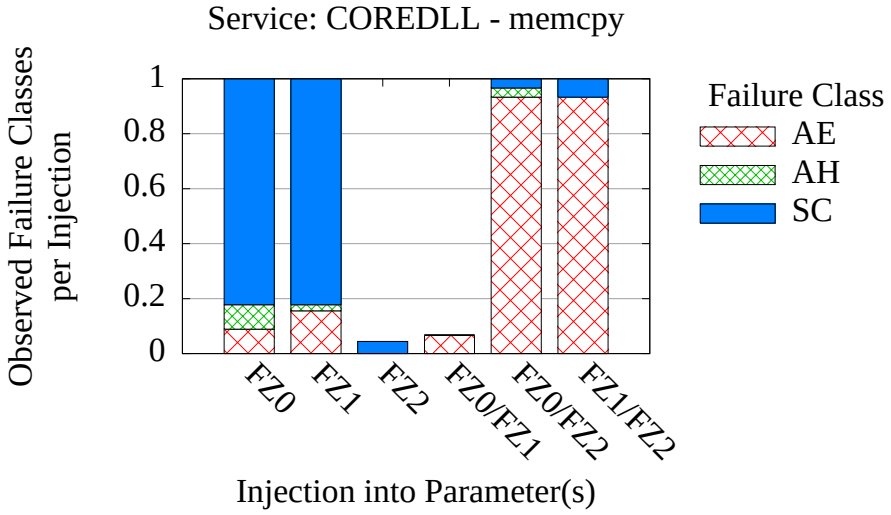

Figure 5.15: Example for masking effects of the FuzzFuzz model

effort, especially for critical applications.

The biggest downside of higher order fault injections lies in the larger number of experiments. As discussed for the FuzzFuzz and FuzzBF models, the number of possible faults to test against is significantly larger. As a consequence, many more runs would need to be performed to obtain a relative coverage of the entire fault space similar to what is achieved for classical models. On the other hand, given that classical models fail to uncover robustness issues that higher order models reveal, the value of such "input-based" coverage criteria appears questionable. A less debatable increase in experiment numbers is mandated by the findings on masking and amplification effects. While higher order fault models prove effective in identifying robustness issues that remain undetected by classical models and, therefore, *cannot be neglected* in robustness assessments relying on debug FI, they require *additional experiments with the classical models* to isolate cause-effect relationships in the experiment results.

Caveats: As previously discussed, exhaustive tests are feasible for the BF and SimBF models, while they are not for the FZ-based models. Consequently, the presented results are based on exhaustive injections with BF and SimBF, whereas a fixed number of test cases has been selected for the FZ-based models. Interestingly, the BF and SimBF faults that were injected are also among the injection candidates of the FZ model. However, the results for the (higher order) bit flips differ significantly, indicating that there are strong effects depending on the chosen samples for the FZ-based models. An investigation of why and how systematically derived faults, e.g., the BF-based models, give better evaluation results is beyond the scope of this thesis and most likely also SUT-specific, as the discussion of the role of bit locations for bit flip models by [JSM07b] indicates.

The performance of the evaluated fault models differed for different drivers. Fur-

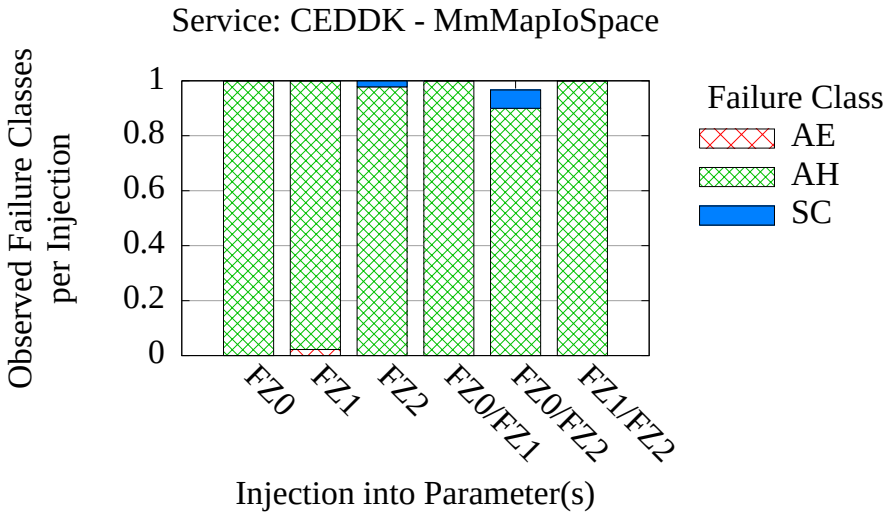

Figure 5.16: Example for amplification effects of the FuzzFuzz model

thermore, some interfaces are only used by some drivers and not by others. Therefore, the efficiency of an assessment depends on an interplay of fault models and the precise system configuration, comprising fault loads and applied workloads. Consequently, details of the presented results, such as the perceived rankings among fault models for the different criteria, cannot be simply extrapolated and likely do not equally apply for other systems. As the single fault assumption is uniformly applied, the goal of the conducted study was to investigate whether there are systems for which it does not apply. The presented results clearly show this and the conclusion that should be drawn from this result is that the validity of the single fault assumption has to be carefully investigated for each conducted robustness assessment.

5.6 Related Work

Compositional and higher order fault models have been used in previous studies, e.g., to evaluate the fault-tolerance of systems with respect to coincident subsystem failures. I discuss these cases in the following and argue why I consider them higher order models. The main differences of these studies and the work presented in this thesis are that (1) the used models are commonly derived as classes of representative faults for operational FI[2], whereas this thesis develops a taxonomy and framework for fault composition, and (2) the effects of fault composition on experimental results are not assessed.

[2]often even without notice of their parallel nature

5.6.1 Higher Order Injections into Function Call Parameters

In the Ballista project (cf. Section 2.3), higher order injections into multiple parameters of a function call have been performed with a data-type fault model [KKS98]. The authors do not differentiate between valid and invalid parameters that are combined in the tests and do not quantitatively analyze the impact of higher order injections, i.e., the combination of invalid inputs. However, they report that higher order injections have not had a significant impact on the detection probability of robustness issues [Pan99; KDD08], which contradicts the results presented in Chapter 5. There are four possible reasons for this deviation:

1. Ballista and the case study conducted for the validation of the higher order models' applicability and effects in Chapter 5 target different interfaces of the OS (API vs. driver interface).

2. The quoted Ballista results were obtained more than a decade ago and may not hold for newer systems.

3. The results may significantly differ due to the different fault models the compositional models were composed from.

4. The experimental methodologies for robustness testing and debug FI differ (cf. Chapter 2).

Although not discussed explicitly, automated fault injections applied for the determination of the robust argument type for n-ary functions in HEALERS [FX02] (see also Chapter 2) probably also use higher order injections into different parameters: If the robust types of multiple arguments depend on each other, this dependency can only be reflected by higher order injections. If there is no such dependency, simultaneous tests for robust types of independent arguments can reduce the test effort.

5.6.2 Validating Software-Implemented Hardware Fault-Tolerance

Software replication (e.g., N-copy programming, *NCP* [AK88], or process-level redundancy, *PLR* [SBM+09]) is a prominent technique to flexibly tolerate the effects of non-deterministic hardware errors on software, e.g., single-event upsets, at run-time. In order to validate the effectiveness of setups with more than two replicas, multiple hardware faults are emulated to provoke the coincident failure of several replicas. Joshi, Gunawi, and Sen [JGS11] discuss the problem of exponentially increasing test candidates if multiple hardware failures are considered in a distributed setting and suggest strategies to reduce them. In contrast to our evaluation of fault coincidence effects on experiment outcomes, their pruning heuristics use static properties that do not account for test results.

Another motivation behind the emulation of multiple correlated or uncorrelated hardware errors affecting software is the increasing error rate expected from future

hardware architectures with ever increasing transistor densities as motivated in Section 1.2. In order to evaluate the impact increasing error rates would have on software built on the assumption of perfectly reliable hardware, the effects of coincident hardware errors on software are emulated, e.g., by modifying machine instructions generated by a compiler [KS09]. The work presented in this thesis differs from other work in that the developed models are not motivated by representativeness for any operationally observed manifestations. Hence, in contrast to other multi bit flip models from related literature, the flips in this thesis are not confined to consecutive bit positions, as expected by typical hardware burst models (e.g., [BSS13]).

5.6.3 Higher Order Mutation Testing

As discussed in Section 4.2.2.3, mutation testing [JH11] is an approach to assess the quality of test suites. *Mutants* of programs are created using mutation operators. A test suite that is capable of distinguishing the original program from the mutant is considered better than one that cannot. Program mutation can be considered fault injection, as it alters software in a controlled manner to assess defect detection. Jia and Harman [JH09] introduced higher order mutants resulting from consecutive applications of multiple traditional first-order mutation operators.

Rather than focusing on the simultaneous occurrence of faults within one single subsystem (e.g., multiple mutations *in the same program*), the compositional fault notion in this chapter is concerned with the simultaneous effect of any subsystems' faults *on the SUT*. Due to this difference, the presented framework for higher order fault models can be applied in both white-box and black-box assessment scenarios.

5.6.4 Accumulating Fault Effects

If fault activations do not immediately result in a detectable failure, errors may remain dormant in the system. In such cases, the effects of multiple fault activations can accumulate in the system. Software aging is an example of minor deviations accumulating over time. For accumulating fault effects, related work on fault injections falls in two categories. One class of work (e.g., [CVM+03; VLM07]) tries to avoid accumulating effects for methodological reasons, as these complicate the identification of effects that an individual fault activation has on the system. The second class of work (e.g., [KRK+98; KLB00; KDD08]) considers accumulating fault effects and either accepts that no direct causal relation between an individual fault activation and observed failure can be established or performs additional experiments to establish these relations. The approaches presented in this thesis fall in the latter category.

5.7 Conclusion

Surprisingly, the interaction of coincident fault effects is often neglected in software robustness assessments. In this chapter, I have investigated the benefits and costs that

come with a consideration of higher order faults in terms of evaluation efficiency.

The proposed coincidence notions enable a large number of novel fault models for FI experiments and although the discussion in this chapter is limited to the design and assessment of only three models, the experimental results indicate that coincident fault models can have significant effects on the coverage in a software robustness evaluation. Depending on the temporal and spatial resolution, there may be defects that cannot be covered by certain coincident models *by design*, as discussed for the FuzzFuzz and FuzzBF models. However, even these models cover a number of defects that are not covered by discrete models in our experiments.

The conducted analyses of the Windows CE driver interface have shown significant fault interactions in kernel functions. This finding contradicts a fundamental assumption, i.e., the *absence* of fault interactions, on which most contemporary fault injection frameworks and combinatorial testing [KWG04] as a test case reduction technique are based on. Thus, our results indicate that such reductions could potentially be fallacious, and results based on such assumptions may need careful re-investigation.

The conducted analyses have also shown significant masking and amplification effects, which prevent higher order fault injections from replacing classical models. As a consequence, robustness assessments should be conducted using both higher order models and the single faults that the higher faults are composed of to establish reliable cause-effect relations between injections and observed failures and to prevent relevant robustness issues from escaping detection through masking effects. I propose the parallel execution of fault injection experiments on parallel hardware as a remedy for these increased experiment counts necessitated by higher order fault models in Chapter 6.

6 Coping with the Combinatorial Explosion for Higher Order Faults

In Chapter 5, I have introduced higher order fault models that challenge the single fault assumption underlying most debug FI frameworks. An evaluation of these models' performance in robustness assessments using the metrics introduced in Chapter 4 revealed that the single fault assumption is not generally valid and that the proposed higher order fault models prove effective for revealing robustness issues that classical models fail to detect. However, a number of robustness issues that were identified by classical models were missed by the higher order models. Moreover, an analysis of fault interactions resulting from higher order injections showed that additional experiments with classical models are required to establish accurate cause-effect relations for injected faults and observed failures. In conclusion, while higher order fault models proved efficient for the conducted robustness assessments, they are no substitute for classical models and should be used *in addition*. Together with the higher number of fault conditions that fault compositions in higher models impose, this leads to a significant increase in the number of injection runs.

In order to counteract this increase, I propose to exploit parallel hardware to execute multiple experiments at the same time. While such PArallel fault INjections (PAIN) yield higher experiment throughput, they are based on an assumption of non-interference among the simultaneously executing experiments. I therefore investigate the validity of this assumption and determine the trade-off between increased throughput and the accuracy of experimental results obtained from PAIN experiments to ascertain the feasibility of experiment speedup through parallel execution. The contents of this chapter is based on a conference paper accepted for presentation at ICSE 2015 [WSN+15].

6.1 The Performance-Validity Trade-Off for PAIN Experiments

Given the high complexity of modern software systems, debug FI typically requires a significant number of experiments to cover all relevant faults in a SUT's OE, with studies reporting thousands, or even millions, of injected faults [KD00; AFR02; DAS+12; NCD+13a]. The problem of high experiment counts is exacerbated by the result that

higher order fault injections, i.e., combinations of several injected faults, need to be considered as well (see Chapter 5). Moreover, recent studies show that failure recovery protocols in distributed systems exhibit vulnerabilities to simultaneously occurring faults that can only be uncovered by injecting fault combinations [GDJ+11; JGS11], and that software faults cause the simultaneous corruption of several interface parameters as well as shared memory contents [LNW+14]. A "combinatorial explosion" of the number of experiments is the consequence.

Finally, the injection of several faults (namely, higher order mutations, HOMs) is being increasingly investigated in the field of mutation testing[1], as HOMs have proven effective at improving the quality of test suites [JH09; PM10]. Despite the demonstrated utility of higher order fault injections, the entailed increase in numbers of experiments remains a considerable challenge for their applicability.

In order to cope with the high number of FI experiments, two different strategies can be adopted. The first strategy attempts to reduce the number of experiments that need to be performed. Search-based techniques and sampling strategies for large test sets (e.g., [JH08; SAM08; JGS11; NCD+13a]) fall into this category. The second strategy attempts to utilize the increasing computational power of modern hardware, where several experiments are executed at the same time on the same machine (*parallelization*) to better utilize the parallelism of the underlying hardware (e.g., [Las05; DCB+06; OU10]). While parallelization ("throwing hardware at the problem") is less elegant, it is an appealing solution, as it is generally applicable, whereas the applicability of sampling and search-based techniques depends on domain-specific knowledge in most cases. Parallelization, therefore, seems to be a promising solution to cope with the high number of experiments, especially as it can be combined with domain-specific sampling and pruning.

Nevertheless, parallelization relies on an implicit assumption that *executing several experiments in parallel does not affect the validity of results*, although the validity of this assumption is not trivial. Even if the experimenter takes great care to avoid interference between experiments (e.g., by running them on separate CPUs or virtual machines), there is a number of subtle factors (such as resource contention and timing of events) that can change the behavior of the OE and SUT (e.g., faults can lead to different failure mode distributions than those observed under sequential execution), thereby invalidating the results and negating the benefits of parallelization. This is a concern especially for embedded, real-time, and systems software, which are an important target of FI experiments, and where studies have shown that faults often exhibit non-determinism and time-sensitive behavior [CGN+13; AFR02].

Hence, in order to conduct *efficient and accurate* parallelization, this chapter proposes PArallel fault INjections (PAIN) as an FI framework. As FI is applied mostly for the assessment of critical systems, a major concern that outweighs performance considerations is the confidence in the validity of the experimental results; it is of utmost

[1] FI and mutation testing use similar techniques to achieve their objectives, but these objectives differ with mutation testing primarily targeting the quality of test suites versus robustness assessments for FI. The interested reader is referred to [JH11] for a fuller discourse.

importance to avoid interference of PAIN experiments that affects their outcome. In addition to experiment throughput, the validity of results from parallel experiments is also assessed. The chapter makes the following contributions:

- A qualitative and quantitative analysis of the impact of parallelism on the experiment throughput and the validity of the results using results from extensive PAIN experiments on the Linux kernel under an Android emulator environment
- Guidelines to tune the main factors that affect experiment throughput and the validity of PAIN experiments, including the degree of parallelism and failure detection timeouts

Following the related work in Section 6.2 the hypotheses that the conducted analyses are based on are presented, followed by the corresponding experiment design and setup discussion over Sections 6.3 and 6.4. Section 6.5 presents the obtained results and provides an in-depth analysis of the factors contributing to the observed effects. Sections 6.6 and 6.7 summarize the lessons learned and discuss threats to the validity of the empirical study.

6.2 Related Work: Fault Injection & Test Parallelization

Similar to higher order mutation, higher order fault models for debug FI lead to massively increased numbers of experiment candidates. To cope with these effects, I propose an FI framework for test parallelization. Similar ideas have been explored for correctness testing, as discussed in Section 6.2.1. For debug FI, the result accuracy is of particular concern, as the discussion in Section 6.2.2 emphasizes, whereas the discussed approaches to parallel correctness testing mostly disregard this issue.

6.2.1 Perspectives on Test Parallelization

In the software community, the idea of test parallelization has been mostly driven by the advent of increasingly parallel system architectures, such as multiprocessor and networked systems [Sta00]. Kapfhammer proposed parallel executions to complement sampling techniques for improving the performance of regression testing [Kap01]. Lastovetsky used parallel computing to achieve a throughput improvement (*speedup*) of factor 6.8 to 7.7 compared to sequential execution [Las05] for testing of a complex CORBA implementation on several platforms. Duarte et al. developed GridUnit as an extension of the JUnit testing framework to support parallel test execution in a grid computing environment, achieving speedups ranging between 2 and 71.11 [DCB+05; DCB+06; DWB+06]. Parveen et al. reported a speedup of 30 for a 150-node cluster using their MapReduce-based unit testing tool HadoopUnit [PTD+09]. Li et al. describe the design of a grid-based unit testing framework and comparatively evaluate the performance of different task (i.e. test) scheduling mechanisms [LDZ+06]. Oriol

and Ullah ported the York Extensible Testing Infrastructure (YETI) to MapReduce architectures using Hadoop [OU10]. They reported a speedup of 4.76 for a fivefold increase in computational resources. In contrast to other work, which only reported on performance, they also compared the results of sequential versus parallel tests and reported that the numbers of detected defects are equal in both cases. However, they did not specify whether the same defects were detected or just equal numbers, thereby leaving the question of effects on result accuracy open. Other recent approaches dealt with Testing-as-a-Service (TaaS) for both dynamic tests [YZX+09; YTC+10] and program analyses [SP10; CBZ10; CZB+10; MEK+12].

While multiple test parallelization approaches have been advocated, their primary focus was to increase test throughput. Interferences between parallel tests were not investigated in these studies, because tests were performed on individual software units rather than integrated systems, and the execution of test cases was not influenced by non-deterministic factors such as timing and resource contention [DCB+05; OU10]. In other cases, tests that contended for shared resources were executed sequentially or on distinct hardware machines to conservatively avoid any interference [Las05]. Interferences between parallel experiments are a potential, yet unexplored, issue for FI experiments, since they usually target complex, integrated systems rather than individual components, and since injected faults (like real faults) can result in unforeseen and non-deterministic behavior [Gra86; GT07].

6.2.2 FI Validity in Parallel Execution Environments

There are numerous factors that can affect the accuracy and validity of FI, possibly leading experimenters to false conclusions about the dependability of a system if they are disregarded. As pointed out by Bondavalli et al. [BCF+07], FI tools and experiments should be regarded as *measuring instruments* for dependability attributes and designed with the principles of *measurement theory* in mind.

For this reason, much research in this field is focused on metrological aspects of FI. Several techniques were developed to reduce the *intrusiveness* of FI tools on the target system, by taking advantage of debugging and monitoring facilities provided by hardware circuitry [CMS98; AVF+01], and by minimizing changes in the target software [SFB+00]. Skarin, Barbosa, and Karlsson [SBK10] assessed the *metrological compatibility* (i.e., statistical equivalence) of results obtained from these alternative FI techniques. Kouwe, Giuffrida, and Tanenbaum [KGT14] evaluated the *distortion* of results due to injected faults that have no effect on the system and are under-represented. In [CLJ+04; CNR+13], the *precision* and the *repeatability* of FI is evaluated in the context of distributed systems, which are affected by non-determinism and by clock skew issues. Irrera et al. [IDM+13] assessed whether *virtualization environments* could be used for FI experiments without impacting on system metrics related to performance and resource usage: While their conclusions are positive, their experiments show that virtualization actually had a noticeable impact on some of the monitored metrics.

While some studies have advocated the potential benefits of parallelizing FI experiments [BKK+10; HBK+10; MEK+12; BC12], none of them investigated the impact of

parallelism on the validity of results (i.e., whether results from parallel experiments are metrologically compatible to sequential ones). In these studies, FI experiments were executed in separate virtual machines [BKK+10; HBK+10] and OS processes [BC12] to isolate the experiments. Memory protection mechanisms provided by virtualization and the OS can prevent data corruptions from propagating among experiments. For this reason, experiments are assumed to be independent from each other, and are treated as an "embarrassingly parallel" problem (i.e., experiments can be arbitrarily parallelized). Nevertheless, there are potentially adverse effects of parallelization: In fact, it is difficult to enforce perfect *performance isolation* among virtual environments [GCG+06; SC09], and performance interferences (e.g., the shortage of resources or the timing of events) can significantly change the *behavior* of a system, and even affect its security [HL13; NVN+13]. To account for such effects, this chapter investigates the interplay between parallelism and the experimental results in addition to its effects on experiment throughput.

6.3 Experimental Design

The following research questions (RQ) are addressed by experimentally investigating the feasibility of *increasing SFI experiments throughput by parallel execution without compromising the accuracy of the results.*

RQ 1 *Can parallel executions of FI experiments on the same machine increase the throughput of FI experiments?*

RQ 2 *Can parallel executions of FI experiments on the same machine change the results obtained from FI experiments?*

If the answer to RQ 1 is positive and to RQ 2 negative, then FI parallelization has no adverse effects and should be applied whenever parallel hardware is available. However, if RQ 1 is negative and RQ 2 positive, parallelization should be avoided. If both answers are negative, the decision whether to parallelize or not should be driven by other factors, such as hardware cost or complexity of the experiment setup. If both are positive, parallelism can be beneficial, but it can also potentially affect the accuracy of results. In this case, an additional question needs investigation.

RQ 3 *If RQ 1 and RQ 2 hold, can the parallelization of experiments be tuned to achieve both a (desirable) throughput increase and avoid the (undesirable) inaccuracy of results?*

6.3.1 System Model

The impact of parallelism is investigated in an experimental context similar to other contemporary FI studies. Deviating from the robustness assessments of the Windows CE kernel conducted in Chapters 4 and 5, the Linux-based Android OS kernel

[Gooa] is considered as the SUT in this chapter. Similar to the previous chapters, the SUT is exposed to external faults at its device driver interface. An important difference between the two SUTs is that Windows CE, in contrast to Android, is a real-time OS. Due to its suitability for real-time applications, Windows CE features a far more precise control over which components are loaded when, and how they are scheduled for execution, leaving much less room for non-deterministic preemption in task scheduling if configured properly. For Android, the absence of non-determinism of interleaving task executions is difficult to achieve without drastic changes in the Linux kernel, on which Android is based. This difference leads to non-deterministic activation sequences for the fault load, which in turn leads to non-deterministic experiment results. To achieve statistically stable results, each campaign has to be executed multiple times. Consequently, the number of required experiments is also increased by non-deterministic behavior of systems in addition to the aforementioned increase due to the adoption of higher order fault models.

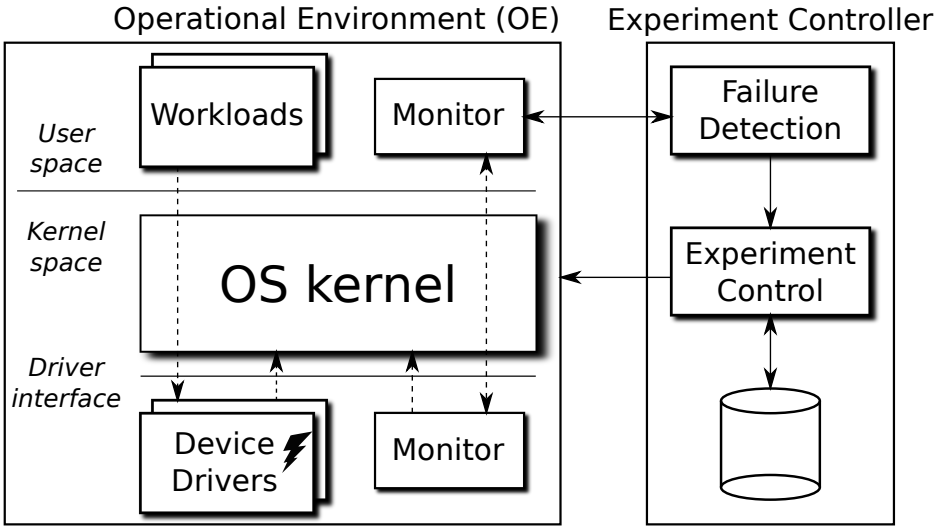

Figure 6.1: System model and experiment setup

In order to expose the SUT to external faults at its driver interface, faulty versions of driver code are created using code mutation (see Section 3.1.1). The mutated drivers are loaded as kernel modules and a workload is executed to trigger the altered code. Figure 6.1 details the experiment setup, which follows the abstract model introduced in Chapter 3. All experiment control logic is external to the target system, which is encapsulated in a virtual machine (VM) to prevent it from being corrupted by the effects of injected faults. To detect failures of the target system, *external failure detectors* are used that collect and analyze messages from the target system, while running in the experiment controller's separate virtual environment. Using these detectors, the following failure modes are reliably detected:

- **System crashes (SC)** by monitoring kernel messages emitted by the VM,
- severe **system errors (SE)** in a similar manner, and

- **workload failures (WF)** by monitoring application log information from the workloads forwarded to the system log that the VM emits.

The configured setup also detects stalls (so-called *hangs*) of the SUT, in which experiments neither make any progress nor produce any information about the failure. The SUT is assumed to hang if the applied workload does not terminate within a timeout interval, which is calculated by adding a generous safety margin to the maximum time needed to produce the correct result when no fault is present. These timeout-based external detectors *assume* hangs using these timeouts:

- During system initialization (**init hang assumed, IHA**),
- after system initialization (**system hang assumed, SHA**), or
- during workload execution (**workload hang assumed, WHA**).

As such timeout-based external detectors are known to be possibly imprecise or inefficient depending on their configuration policy [CNR09; BCC+11; ZLX+12], sophisticated hang detectors inspired by the approach of Zhu et al. [ZLX+12] are additionally used. Two such additional hang detectors are placed in the OE: A *light detector*, executing as a user space process, monitors basic system load statistics. If these statistics indicate lack of progress, the light detector triggers a *heavy detector* executing in kernel space. The heavy detector performs a more accurate (but also more complex and time consuming) analysis and triggers a controlled system crash if a hang is detected. The tests that the light and heavy detectors apply are identical to those suggested in [ZLX+12]. The SUT- and platform-specific threshold values for the experimental configuration in this chapter can be found in the source code at [DM15b]. This target-internal hang detection infrastructure provides two additional failure modes:

- **system hang detected (SHD)**
- **workload hang detected (WHD)**

6.3.2 The Higher Order Code Mutation Fault Model

In the experiments, *driver source code changes* are introduced to emulate residual defects of device drivers in a similar way to recent studies on OS software fault tolerance [NC01; SBL03b] and on dependability benchmarking [DM03b; VM03; DVM04]. The SAFE tool [NCD+13a] has been adopted to inject realistic code changes that were defined on the basis of actual faults found in commercial and open-source OSs [CC96; DM06]. The interested reader is referred to [NCD+13a] for a detailed exposition on fault types.

As faulty drivers are known to constitute a severe threat to system stability, the SUT and OE have to be executed in a strictly isolated environment that (a) prevents experiments from affecting the test bench and (b) enables subsequent experiments to start from a clean environment free from effects of previous experiments. These

requirements result in high experiment overheads for FI-based robustness assessments and thereby limit experiment throughput, which parallelism could compensate for, even for classical experiments that do not take higher order faults into account.

The main purpose for investigating throughput improvements in this thesis, however, is to mitigate overheads from higher numbers of experiments resulting from higher order fault models. In order to investigate the impact of parallelism on high volumes of FI experiments, faults are selected from both single and higher order injections into driver code. Higher order faults are injected by repeatedly invoking SAFE on previously mutated driver code in a similar way to emerging higher order mutation testing approaches [JH09]. The injection of multiple faults leads to a combinatorial explosion of the number of faulty drivers to test with, and allows experimentation with a high volume of injection runs.

6.3.3 Performance and Result Accuracy Measures

As stated in Section 6.1, there are two aspects of interest for the PAIN experiment executions. On the one hand, PAIN is intended to mitigate the experimentation overhead resulting from the usage of higher order fault models proposed in this thesis. On the other hand, the strategy is only viable if it does not affect result validity. I detail the corresponding measures applied in the evaluation of the PAIN approach in the following.

Performance measure: The performance metric considered for evaluating PAIN is *experiment throughput*, measured as the *average number of experiments per hour*. Besides the increase in experiment counts required by higher order FI, a speedup of FI experiments is generally desirable to achieve a better coverage of fault conditions to test with [BKK+10; HBK+10].

The accuracy of FI results needs to be defined in statistical terms, since the outcome of FI experiments is influenced by non-deterministic factors, as mentioned in the beginning of this section. This aspect may not be intuitive and requires a more detailed discussion. In order to observe the effects of injections, the injected code needs to be *activated* during experiment execution [GT07]. As the abstraction from hardware configuration details and the orchestration of access to hardware devices are among the core functions of OSs, most of them do not provide a direct interface to driver functions for programmers. As a consequence, a large and complex software layer interposes between device drivers and user programs (cf. Figure 6.1). Some driver functions (e.g., those related to power management) may even be entirely hidden from user programs and invoked by the OS upon (possibly non-deterministic) hardware events and task scheduling.

Accuracy measures: The measure for *result accuracy* has two aspects. First, it is important whether *result distributions of failure modes* (Section 6.3.1) change with increased parallelism. Second, the degree of result heterogeneity across repetitions

of identical campaigns is of relevance, as this indicates the reproducibility and representativeness of the results. While for the first case, a binary measure that indicates statistically significant deviations suffices, a comparative metric is required for the latter. A *Chi square test for independence* is conducted to decide whether or not result distributions for parallel experiments differ significantly from result distribution for sequential experiments. To measure the variance of distribution samples, the obtained distributions are interpreted as vectors and their *Euclidean distances from the mean* are calculated. The mean value of all such distances within a set of repetitions for the same setup is used and denoted as heterogeneity metric d.

6.3.4 Hypotheses

On the background of the introduced models and terminology, precise hypotheses can be derived from the research questions stated in the beginning of Section 6.3. In the following, only the null hypotheses to be tested are stated. The alternative hypotheses are the negation of the stated null hypotheses.

Hypothesis $\mathbf{H_0}$ 1 *If the number of parallel experiment instances running on the same physical machine is increased, the experiment throughput does not increase.*

Hypothesis $\mathbf{H_0}$ 2 *If the number of parallel experiment instances running on the same physical machine is increased, the obtained result distribution of failure modes is independent from that increase.*

Hypothesis $\mathbf{H_0}$ 3 *If the number of parallel experiment instances running on the same physical machine is increased, the heterogeneity among repeated injection campaigns does not increase.*

6.4 Experiment Setup and Execution

All software developed to conduct the experiments in this chapter (currently 14 159 physical SLOC[2]) is publicly available at github [DM15b] for the reproduction and cross-validation of the results.

6.4.1 Target System

The Android OS targeted as SUT is Android 4.4.2 "KitKat" with a 3.4 kernel from Google's repositories [Gooc]. The SUT and OE are running in the goldfish System-on-Chip emulator [Goob], which is based on the QEMU emulator/virtualizer [Bel] and ships with the Android SDK.

The MMC driver for the emulated SD card reader of the goldfish platform is targeted for injections. The driver has 435 physical source lines of code. Two different synthetic

[2]generated using David A. Wheeler's SLOCCount

workloads are applied to trigger injected faults in the MMC driver, both of which based on code from Roy Longbottom's Android benchmarks [Lon].

The first workload ("pure") performs file operations on the emulator's virtual SD card in order to (indirectly) exercise the MMC driver and faults injected there. The workload is based on code from the DriveSpeed benchmark app and configured to stress the SD card driver for approximately 30 seconds.

A "mixed" workload adds CPU and memory load to the pure workload. The goal is to create a more diverse utilization of system resources by the emulator to cover a wider range of possible interference between emulator instances competing for shared system resources. Besides DriveSpeed, code from the LinpackJava and RandMem benchmarks is used. All benchmarks are executed as parallel threads, leaving their scheduling to the Android OS.

An additional thread in the workload apps is used to perform WF failure detection, since application failures are signaled as exceptions within Android's Dalvik runtime and need to be explicitly forwarded to the external failure detector residing outside of the emulator.

6.4.2 Fault Load

Applying SAFE for creating faulty versions of the MMC driver results in 273 mutants to test with. The recursive application of SAFE to each of these mutants, as described in Section 6.3.2, yields 70 167 *second order mutants* and, hence, a total of 70 440 faulty driver variants to test with. This drastic increase in numbers from first to second order mutants illustrates the combinatorial explosion resulting from simultaneous fault injections and higher order mutation testing.

To test the hypotheses outlined in Section 6.3.4, the conducted campaigns are restricted to 400 randomly sampled mutants from the set of first and second order mutants. As the outcomes of experiments are subject to non-determinism and experiment repetitions are required to establish confidence in the obtained results, each campaign is repeated three times to account for non-determinism (Section 6.3.3).

6.4.3 Execution Environments

In order to conduct parallel tests on the same hardware, instances of the goldfish emulator, in which the target system is executing, are replicated on a single host machine. This parallelization by replication of emulator instances reflects the implicit assumption of non-interference at question, as emulation and virtualization form the basis of recent approaches to test parallelization [YZX+09; YTC+10; BKK+10; HBK+10; MEK+12].

In order to avoid effects from a single platform to bias the results, all experiments are executed on two different platforms:

- A *desktop* configuration with an AMD quad-core CPU, 8 GiB main memory, and a 500 GB hard drive operating at 7200 RPM.

- A *server* configuration with two Intel Xeon octa-core CPUs, 64 GiB main memory, and a 500 GB hard drive operating at 7200 RPM.

To avoid differing CPU frequencies from biasing results, frequency scaling has been disabled and all cores on both machines have been configured to constantly operate at 1.8 GHz (which was the only possible common configuration on both platforms). Hyper threading has been disabled on the Intel processors to achieve better comparability with the AMD processors, which do not provide this feature. The desktop configuration is running Ubuntu 13.10, the server configuration CentOS 6.5.

The number of parallel instances running on the same machine, which is the controlled variable in our experiments, is initially chosen as a) 1 (sequential) and b) 2N, where N is the total number of physical cores available in the machine. 2N is a common configuration to maximize hardware utilization. By launching more instances than actually available CPU cores, the cores are more likely to be utilized when some processes wait for I/O, without requiring frequent migration between cores, which would impair the effectiveness of L1 and L2 caches. These two basic settings suffice for a fundamental assessment of the hypotheses stated in Section 6.3.4. To answer RQ 3, the analysis is extended to further degrees of parallelism.

The different factors outlined above yield 8 different configurations (2 workloads, 2 hardware platforms, 2 degrees of parallelism). For each of these, campaigns of 400 experiments are executed three times (cf. Section 6.4.2), resulting in 24 campaigns with 400 experiments each to test Hypotheses 1 to 3. The results of these experiments are reported in the following section, along with a set of additional experiments for an in-depth analysis of the initial result set.

6.5 Experimental Results and Data Analysis

6.5.1 Initial Results

Table 6.1 shows the results of the 24 campaigns to test Hypotheses 1 and 2. Each row lists the results for a different setup. The HW column specifies the hardware platform, the # column the number of parallel instances of the emulator performing the experiments, and the WL column the used workload. The next ten columns contain the average number of experiments that resulted in the corresponding failure mode. In addition to the eight failure modes specified in Section 6.3.1, there are two additional columns for possible experiment results: An **Invalid** class of experiment results for cases in which the experiment control logic had to abort the experiment due to unforeseen failures within the control logic, and a **No Failure (NF)** class for experiments that finished execution without any failure detection. Unforeseen failures include (for instance) cases when the host OS cannot fulfill requests for resource allocations. The last three columns list the throughput in experiments per hour, the average experiment duration in seconds, and the average Euclidean distance of obtained samples from the mean values as introduced in Section 6.3.3.

Table 6.1: Mean failure mode distributions, performance and accuracy measures from 24 experiment campaigns

Setup			Failure Modes										Performance and Accuracy Measures		
HW	#	WL	Invalid	NF	SC	SE	WF	SHD	WHD	SHA	WHA	IHA	Throughput (exp./h)	Experiment Duration (s)	d
Desktop	1	pure	0.00	108.67	97.00	0	182.67	6.33	0	0	0.00	5.33	16.4	219.21	0.89
Desktop	1	mixed	0.00	108.00	97.00	0	182.00	0.00	0	0	6.33	6.67	12.5	286.97	2.02
Server	1	pure	0.00	101.33	97.00	0	183.00	18.67	0	0	0.00	0.00	14.6	246.38	1.26
Server	1	mixed	0.00	114.67	97.00	0	183.00	5.33	0	0	0.00	0.00	12.2	295.36	0.63
Desktop	8	pure	0.00	42.00	97.00	0	181.33	25.67	0	0	48.67	5.33	67.1	416.23	7.34
Desktop	8	mixed	0.00	1.00	96.67	0	6.33	10.00	0	1	281.67	3.33	56.1	493.77	3.50
Server	32	pure	0.33	95.00	97.00	0	182.67	22.00	0	0	0.00	3.00	146.3	496.98	3.37
Server	32	mixed	0.00	65.00	97.00	0	179.00	5.00	0	0	48.00	6.00	115.6	616.19	6.35

From the obtained experiment throughput data, **Hypothesis 1 is clearly rejected**: In the parallel case the average experiment throughput is considerably higher than in the sequential case. An average speedup between 4 and 4.5 is achieved for an 8-fold increase of instances on the desktop machines and an average speedup between 9.4 and 10 for a 32-fold increase on the servers. The throughput calculation is not only based on experiment execution times, but also includes the processing overhead of the experiment control logic. In the conducted experiments, however, this overhead was small (less than one second per experiment) compared to the experiment duration.

To test Hypothesis 2, a Chi-square test for independence has been conducted to assess if the observed result distributions are statistically independent from the degree of parallelism. As multiple tests are performed simultaneously on the same population, the p-values have been adjusted according to the Benjamini-Hochberg procedure [BH95] to account for the risk of false discoveries (i.e., incorrect rejections of the null hypothesis) in such cases. These adjusted p-values (p) and the corresponding test results are shown in Table 6.2, along with the normalized Pearson coefficients (r). The normalized Pearson coefficient indicates the relative "strength" of correlation and can be used to compare the degree of correlation between parallel/sequential execution and the different result distributions. The coefficient also indicates positive or negative correlation. For the Chi square tests the absolute numbers from the distributions have been used rather than the mean values reported in Table 6.1. The results in Table 6.2 indicate that there is no independence of the result distributions from parallelism in three out of four cases, and **Hypothesis 2 is, therefore, rejected**.

Table 6.2: Chi-square test of independence for parallelism and initial result distributions

HW	WL	p	r	Result
Desktop	pure	3.2×10^{-55}	0.45	reject
Desktop	mixed	0	0.90	reject
Server	pure	2.1×10^{-1}	0.1	do not reject
Server	mixed	4.3×10^{-41}	0.40	reject

To test Hypothesis 3, the Euclidean distance of the mean distributions shown in Table 6.1 to each of the three distribution samples they were derived from has been calculated. The mean values of these distance measures are shown in the d column of Table 6.1. The heterogeneity of result distributions from parallel experiments is between 1.7 and 10.1 times higher than the heterogeneity of sequential results and, therefore, **Hypothesis 3 is rejected**.

From the presented results, both RQs 1 and 2 are positively answered and, consequently, RQ 3 is addressed in the following.

6.5.2 Influence of Timeout-Values on the Result Distribution

In order to better understand the trade-off between experiment throughput and result accuracy, a closer look at the observed changes in the result distributions is required. While the numbers for the result classes Invalid, SC, SE, WHD, and SHA only marginally differ across the different setups, we see major deviations in SHD and WHA failure modes, especially illustrated by the drastic change in the distribution for 8 instances on the desktop setup with the mixed workload (cf. Table 6.1). As both failure modes are related to hang detectors and these depend on timeouts for detection, the increased rates for parallel experiments are suspected false positives of these detectors and that their timeouts need adjustment in the parallel case. Indeed, the experiments exhibit longer execution times in the parallel case, as the *Experiment Duration* column of Table 6.1 shows. Compared to sequential experiments, the execution times are roughly doubled. To prevent the longer execution times from affecting results, the timeout values for the WHA, SHA, and IHA detectors are tripled for the re-execution of the experiments. Higher timeout values lead to unnecessarily long wait times in the case of an actual hang failure and Section 6.5.3 discusses better strategies. In the results from the sequential experiments of the conducted study, there is a relatively small fraction of hang failures and the longer detection time for those few cases are considered a reasonable cost for more accurate results in the parallel case.

After eliminating this potential source of deviations in the result distributions, the parallel campaigns were re-executed with the modified setup. All further analyses presented in this chapter focus on experiments with the mixed workload, as these exhibit higher correlation coefficients in both setups. Table 6.3 shows the obtained results, which are closer to the distributions obtained for sequential runs. However, while the heterogeneity of results has decreased for the server setup, it has increased for the desktop setup and this divergence of result accuracy is also reflected by the Chi square test results in the first two rows of Table 6.5: While the result distribution for the desktop setup still significantly correlates with the degree of parallelism, it is statistically independent for the server setup. The distribution differences that lead to this indication of diverging results are mostly due to differing SHD and WHA failure counts. As a consequence, appropriate timeout selection strategies for the corresponding detectors on this platform are investigated in Section 6.5.3.

The improved accuracy of experimental results on the server platform raises the question, whether similar systematic causes of result divergence can be identified. In order to provoke a stronger impact of such potential effects on the result distributions, additional campaigns on the server platform with 36, 40, 44, and 48 instances have been conducted. The results are shown in Table 6.4. The comparatively large improvement in terms of throughput is probably due to a minor change in the experiment logic to regularly clean parts of the host file system from temporary files used by the experiment controller. As the Chi square tests for independence in Table 6.5 show, the result distributions for 36, 40, and 44 instances do not significantly differ from the results obtained from the sequential campaigns, whereas the distributions for 48 instances do. We also observe a strong increase of the heterogeneity measure

Table 6.3: Mean failure mode distributions, performance and accuracy measures from repeated parallel experiments

Setup			Failure Modes										Performance and Accuracy Measures		
HW	#	WL	Invalid	NF	SC	SE	WF	SHD	WHD	SHA	WHA	IHA	Throughput (exp./h)	Experiment Duration (s)	d
Desktop	8	mixed	0	104	97	0	181.67	5.00	0	0.67	11.33	0.33	47.0	587.25	5.41
Server	32	mixed	0	114	97	0	181.67	6.67	0	0.67	0.00	0.00	118.1	619.48	1.99

Table 6.4: Mean failure mode distributions, performance and accuracy measures from highly parallel experiments

Setup			Failure Modes										Performance and Accuracy Measures		
HW	#	WL	Invalid	NF	SC	SE	WF	SHD	WHD	SHA	WHA	IHA	Throughput (exp./h)	Experiment Duration (s)	d
Server	36	mixed	0.00	113.67	97	0	181.67	7.00	0	0.33	0.33	0.00	157.1	712.11	2.16
Server	40	mixed	0.67	113.00	97	0	180.00	8.00	0	0.67	0.67	0.00	154.1	834.14	1.98
Server	44	mixed	0.00	112.00	97	0	180.33	6.67	0	1.33	2.33	0.33	143.0	951.52	3.54
Server	48	mixed	0.67	104.67	96	0	177.67	11.00	0	2.00	5.00	3.00	102.5	1069.03	6.81

d for the latter. Although the results for the setups with 36, 40, and 44 instances are still more heterogeneous than the results obtained from sequential experiments, it is deemed acceptable and results are expected to further stabilize with moderately higher numbers of campaign repetitions.

Table 6.5: Chi-square test of independence for parallelism and additional result distributions

HW	#	WL	p	r	Result
Desktop	8	mixed	6.7×10^{-7}	0.18	reject
Server	32	mixed	1.0	0.05	do not reject
Server	36	mixed	1.0	0.05	do not reject
Server	40	mixed	7.8×10^{-1}	0.08	do not reject
Server	44	mixed	2.1×10^{-1}	0.10	do not reject
Server	48	mixed	1.3×10^{-4}	0.17	reject

6.5.3 Selection of Timeout Values for PAIN Experiments

As the results of the conducted experiments indicate, the selection of suitable timeout values for the employed hang detectors is crucial to avoid false positive hang detections, and good estimates become challenging for PAIN experiments. The timeout values for the initial experiments had been based on experience with previous sequential experiments to which a generous safety margin had been added. This turned out to be insufficient for parallel executions, as is evident from the drastically increased WHA detections in Table 6.1. A naïve approach to avoid these false positives would be to upscale the timeout values with the number of parallel instances. However, as the degree by which execution times increase with the numbers of instances is generally unknown and, according to the presented results, also depends on the execution platform (hardware and OS), this entails an iterative process of trial and error until suitable timeout values are found.

A better strategy is to estimate values based on observations made during so-called *golden runs* on the intended execution platform *without injections*. Such runs should be performed for each targeted level of parallelism and relevant timing data be recorded as a baseline to derive suitable timeout values. To assess the suitability of the approach for estimating accurate timeout values, a number of such calibration runs have been performed for the parallel desktop setup with the mixed workload using a modified version of the experiment controller to record additional data. During calibration runs the times needed for system initialization (*sysinit*) and for workload completion are measured, as these two time values are relevant for the IHA and WHA detectors. As the recorded times appear to be normally distributed, the 99.99 percentiles of a fitted normal distribution are considered as timeout values, i.e., 99.99 % of experiments

should execute without false hang detections if the fitted distribution matches the actual distribution of run times.

However, the repetition of the original experiments for the same setup using these timeout values yielded unsatisfactory results with 106 WHA and 3 IHA detections in average. While the number of IHA detections is in the expected range for the 99.99 percentile, the number of WHA detections, which is considerably lower than with our original timeout values (cf. Table 6.1), is still 10 times higher than with our tripled timeouts (cf. Table 6.3). The difference in the quality of the estimated timeout values indicates that the chosen calibration approach is suitable for estimating system initialization timeouts, but not workload timeouts. Since a modified version of the experiment controller was used to perform the calibration runs, the workload behavior for real experiments is suspected to differ from the observations in the calibration runs.

To further investigate the timing behavior for real experiments, the original experiments for the desktop setup with the mixed workload have been repeated with stepwise increased degrees of parallelism. To avoid interference from any spurious detections, extremely high timeout values were configured for the hang detectors. The system initialization and workload times were inferred by parsing experiment logs. In order to exclude outliers in the collected data, only time values between the 0.5 and 99.5 percentiles have been included in the analysis. The graph in Figure 6.2 visualizes the minimum and maximum times observed. With increasing parallelism, the difference between the observed minimum and maximum times also increases, with a maximum difference of about 618 seconds for the system initialization and about 323 seconds for the workload times. Moreover, the maximum times increase with increasing parallelism whereas the minimum times are almost constant.

A comparison of the observed time distributions for our calibration runs and the repeated experiment runs shows that the workload execution completes significantly faster in the calibration setup: With 8 parallel instances, for instance, the workload completes after about 98 seconds on average in the calibration, but needs about 304 seconds on average in the experiment setup.

As these results show that a dedicated calibration setup may exhibit different timing behavior than real experiments, the choice of suitable timeout values should be based on timing data from real experiments rather than calibration runs. Timeout values based on the 99.99 percentile of a distribution fitted to the timing data from actual experiments yields hang detection counts comparable to the original experiments with tripled timeout values (cf. Table 6.3), confirming the suitability of the approach. The new timeout value for the IHA detector is about 300 seconds shorter than the original tripled value, whereas the new WHA detector timeout is about 30 seconds longer.

With timeout values of comparable or even better accuracy than the previous trial and error approach, the proposed systematic approach to calculating timeout values is preferable if it results in acceptable overhead. Timing data from 3 complete experiment campaigns were used for the systematic timeout estimation, summing up to 1200 experiments. While this overhead is acceptable for large FI campaigns, fewer experiments would be desirable, if they provide similarly accurate estimates. To improve the performance of our timeout value assessment by finding a subset of the available data

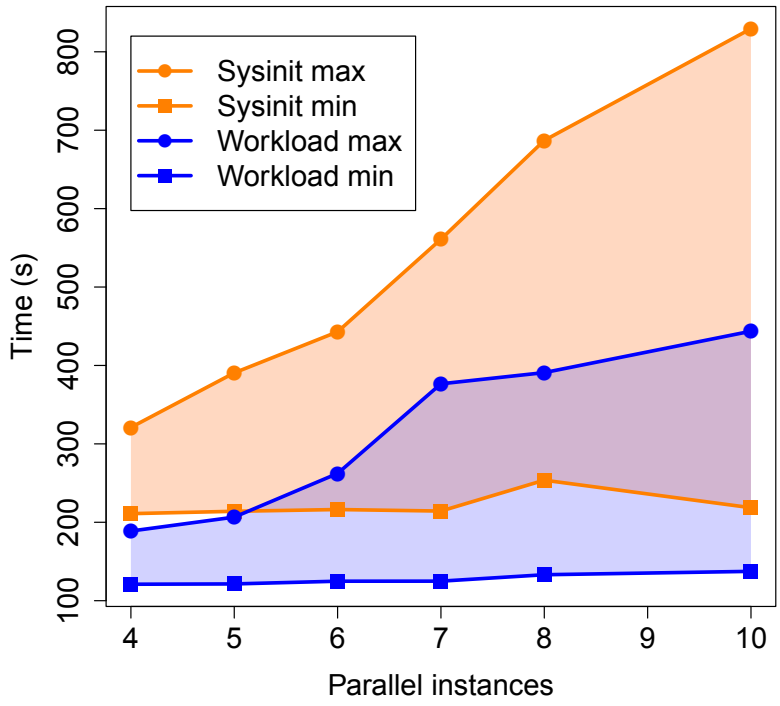

Figure 6.2: Minimum and maximum times for system initialization and mixed workload execution observed during experiment runs with increasing levels of parallelism for the desktop setup.

samples that provides sufficiently accurate estimates, timing data subsets of varying sizes are selected and used in the estimation process outlined above. The resulting timeout values for 10 % steps in sample sizes are illustrated in Figure 6.3. 1.0 in the figure corresponds to 873 samples for the sysinit times and 298 samples for the workload times. This is a subset of the total experiment count, as only timing data from experiments, which did not finish their execution prematurely, could be used. For instance, if in an experiment the system crashed during initialization, no valid sysinit time for that test case exists. Since the workload times have a narrower distribution (cf. Figure 6.2), less overall samples are needed to obtain a robust estimation than for the sysinit times. Starting from a sample size of 0.5 to 0.6, the estimated timeout values are very close to the value estimated using all available samples. Thus, the execution of only 2 experiment campaigns would have sufficed for the estimation of suitable timeout values.

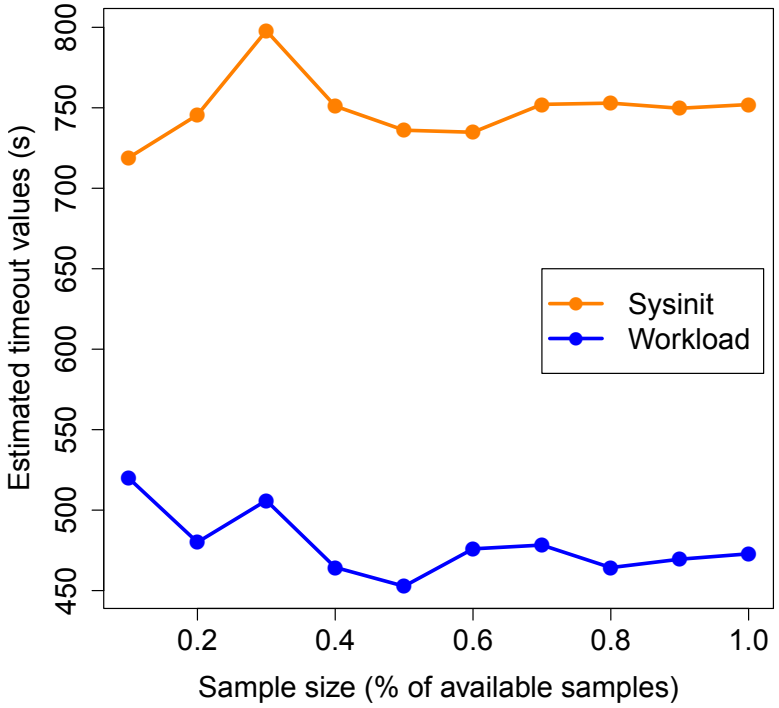

Figure 6.3: Estimated timeout values for different samples sizes of observed real experiment times.

6.6 Discussion

The experiments performed in this study provide interesting insights about the result accuracy and performance of FI experiments. The main lessons that can be drawn from them are summarized in the following. A subtle aspect that turned out to be more important than initially expected was:

> *The correct setup and tuning of parallel experiments may prove difficult and requires special care.*

The parallel execution of experiments had a significant impact on the duration and the timing behavior of individual experiments. Besides changing the mean experiment duration, parallelism also led to incorrect failure detections in the first round of experiments. Even though failure detector timeouts were calibrated on the basis of preliminary fault-free parallel runs, they turned out to be inadequate to account for non-deterministic execution delays caused by unexpected interactions among failing and non-failing VMs that were not observed in fault-free runs.

Moreover, the technical implementation of the PAIN setup, being concurrent and complex by itself, required significant efforts for testing and debugging. To achieve a reliable setup, a number of transient and subtle issues had to be found and fixed that were related to

- portability across different platforms (e.g., the desktop and the server platforms had different limits on the maximum number of processes and on other resources, causing experiment failures),
- resource leaks of the OE (in the case of the Android emulator, temporary files), and
- communication between the experiment controller and the OE, which in some cases went out of sync due to unexpected timing behavior and the loss of messages.

On the basis of this experience, it is advisable for researchers and practitioners, who design parallel FI setups, to pay attention to these aspects. As the PAIN framework presented in this chapter has matured to avoid a number of such issues, I hope that it provides a useful basis for future parallel experiments and experimentation frameworks. PAIN is based on GRINDER, a versatile fault injection framework designed for reuse in a wide range of FI-based assessment scenarios with different SUTs and OEs. Both PAIN and GRINDER, which is described in more detail in Chapter 7, are provided on github [DM15b; DM15a] under the AGPL v3 license.

Going back to the initial research questions, the conducted experiments provide evidence that parallelism can influence both experiment performance and results. Executing experiments in parallel can significantly RQ 1 increase experiment throughput with a speedup factor of up to 10. Thus, FI experiments can be included among the computer applications that can benefit from parallelism. Nevertheless, it was also observed that:

> *The parallel execution of FI experiments can improve experiment throughput, but can also affect the accuracy of results.*

The statistically significant differences observed in result distributions demonstrate that there are cases in which parallelism can actually change the results of experiments RQ 2, both in terms of failure mode distributions and of result stability across repetitions. Such effects, ultimately, affect the conclusions about fault tolerance properties of the target system. In the presented study, performance interferences between parallel instances changed the failure mode of a subset of experiments (that were originally not failing or failing differently) to hang failures of the OS and the workload. Moreover, these hang failures were not easily reproducible and led to unstable failure distributions.

Such changes in the results were observed for very high degrees of parallelism, e.g., up to three times the number of cores (48 parallel experiments) in the case of the server

setup. Nevertheless, running parallel experiments at a lower degree of parallelism (e.g., 32 parallel experiments for the server setup with appropriate timeouts) was found to not cause statistically significant variations from the results of sequential executions. This observation indicates that parallelism does not necessarily harm result accuracy RQ 3, if the degree of parallelism is not too high.

Therefore, to maximize the experiment throughput while preserving result accuracy, it is important to properly tune the degree of parallelism. For instance, the best throughput in the server setup with mixed workload was achieved when running 36 parallel instances (157.1 experiments per hour on average). Further increasing the number of parallel instances to 48 gradually reduced the throughput (down to 102.5 experiments per hour) and caused statistically significant inaccuracies. This behavior illustrates that result inaccuracies occur when the number of parallel instances exceeds the system's capacity:

> *Negative effects on result accuracy can be avoided, if the degree of parallelism is carefully tuned for best throughput.*

Consequently, testers have to determine the appropriate degree of parallelism for their considered setup by running parallel experiments. For instance, performing preliminary parallel runs with an increasing number of instances is a viable method for finding a suitable degree of parallelism. The timeout value analysis presented in Section 6.5.3 indicates that such runs should not use modified or even simplified versions of the intended experiment workload, as these may yield significantly different loads that, for instance, lead to different timing behavior and inaccurate assessments of resource utilization. While the analyses presented in this chapter focus on FI experiments and are influenced by the way these tests are conducted, the obtained insights also apply to other forms of testing (e.g., unit testing frameworks using fixed timeouts per test case are affected by longer execution times in the parallel case) and systems that exhibit non-deterministic behavior (e.g., concurrent software systems with non-deterministic interleavings are vulnerable to the discussed test result deviations).

6.7 Threats to Validity

For any empirical study, care must be taken when interpreting the results and drawing conclusions. The main threats to validity are the choices for the SUT, the fault model, and the workload. Although the SUT, fault model, and workload differ from those in the previous chapters, the drawn conclusions on experiment throughput and result stability equally apply for those, as detailed below.

SUT selection: The Android kernel was used as the SUT in the presented study. Android is representative of embedded systems software, like Windows CE, and as such among the most relevant targets for FI. In contrast to Windows CE, Android

features more sources for non-deterministic behavior, including concurrency in both user and kernel space, I/O interactions, and the use of non-deterministic heuristics adopted in many parts of the OS, such as in task and I/O scheduling and page reclaim algorithms. As a consequence, experiment result distributions are expected to be more stable for Windows CE and the effectiveness of parallel experiment execution observed for Android should, thus, be even higher for Windows CE. Therefore, the effectiveness conclusion drawn from the study in this chapter also applies for the SUT used in the previous chapters and a wider range of SUTs.

Fault model selection: For the conducted experiments, a higher order fault model based on code mutations was used instead of the interface injection based models introduced in Chapter 5. The model is based on mutations, whose representativeness is supported by extensive analyses of real faults and mutants, and which are an accepted practice for operational FI [ABL05; DR06; DM06; NCD+13b]. In order to investigate the utility of parallelization for coping with the high number of experiments that result from multiple FIs, both 1st and 2nd order mutants are created and used in the experiments. As discussed in Section 3.1.1, interface injections are often preferable for debug FI, as representativeness of injected faults is of little concern and interface injections usually provide higher injection efficiency. This higher injection efficiency could lead to a greater influence of timeout values, as hang failures are expected to occur more often. If the proposed mechanism for timeout value selections presented in Section 6.5.3 is followed, higher hang failure rates from interface injections should be reflected in the chosen values and should not pose a threat to the validity of the conclusions drawn from the study in this chapter. While the achievable throughput for experiments with high hang rates is expected to be lower, a higher injection efficiency also entails that less experiments are required for the detection of robustness issues, which likely compensates for the higher waiting times for hang failures.

Workload selection: As workload, existing performance benchmarks for the Android platform were used to create load for *specific hardware elements*, rather than manually crafted workloads targeting a *specific driver implementation* in the previous chapters. Although benchmarks may not be representative of specific user scenarios, they are the most typical workloads in FI experiments, especially in the context of dependability benchmarks [NC01; VM03; KS08]. Moreover, performance benchmarks are stressful workloads and increase the likelihood of fault activation during experiments [THZ+99]. The new workloads were chosen, as the Windows CE and Android APIs significantly differ and required a re-implementation for the workload. Despite the difference, the type of load that the pure workload in this chapter imposes on the targeted driver is similar to the workload created by the Windows CE workloads. As the obtained results show greater interference for the mixed workload, which creates CPU and memory load in addition to the driver load, this aspect should not threaten the conclusion on throughput improvements by parallel execution, as interference is likely lower for the workload in the Windows CE setup.

6.8 Conclusion

As software systems become more complex and, at the same time, tend to exhibit sensitivity to higher order faults, the number of relevant fault conditions to test against also increases drastically. The simultaneous execution of similar tests on parallel hardware has been advocated as a viable strategy to cope with rapidly increasing test counts for correctness testing. In this chapter I have addressed the question whether such strategies can be applied to speed up FI experiments by performing PArallel INjections (PAIN). Besides assessing the speedup of experiment throughput, the question whether PAIN affects the accuracy of FI results was also addressed. Two measures of "metrological compatibility" (i.e., *how accurately results from PAIN reflect results from sequential experiments*) for FI results are defined and applied in an analysis of higher order fault injection experiments on the Android OS.

The results show that while PAIN significantly improves the throughput, it also impairs the metrological compatibility of the results. The observed result inaccuracy is related both to the degree of parallelism and to the choice of timeout values for failure detection, due to resource contention and timing of events that influence the experiments. To mitigate this interference across PAIN experiments, timeout selection strategies are assessed and corresponding guidelines to tune experiments using data from preliminary experiment executions given, in order to achieve high experimental throughput while preserving result accuracy.

7 GRINDER: A Generic Fault Injection Tool

As stated in Chapter 3, FI tests are usually highly automated for efficiency and to prevent human error from affecting result reliability. While an existing tool was adjusted for the studies presented in Chapters 4 and 5, the tool was tightly coupled with the targeted Windows CE SUT and its OE, and was not reusable for the Android SUT used for the assessment of PAIN experiments presented in Chapter 6. In fact, most existing FI automation tools have been built for a specific application domain, i.e., a certain system under test (SUT) and fault types to test the SUT against, which significantly restricts their reusability.

To improve reusability, few *generalist* FI tools have been proposed in the literature to decouple SUT-independent functionality from SUT-specific code. Unfortunately, existing generalist tools often embed subtle and implicit assumptions about the target system that affect their reusability. Moreover, at the time the tool implementation for the Android robustness assessments in Chapter 6 started, these generalist tools were not openly available to even assess their applicability for the Android SUT.

In order to avoid future re-implementations of the common fault injection functionality discussed in Section 3.1, the PAIN experiment framework for Android has been based on GRINDER, a generalist FI tool that provides an abstract interface for SUT interactions. In this chapter, which is based on a workshop paper accepted for presentation at AST 2015 [WPS+15], I present GRINDER and its application to the Android SUT. To illustrate the reusability aspect, I quote numbers from an extension of GRINDER for a different SUT, an AUTOSAR-based automotive system. This adaptation has been conducted by Thorsten Piper and constitutes no contribution of my thesis.

GRINDER is made available under the AGPL v3 license on github [DM15a].

7.1 Introduction & Related Work

Although a variety of FI tools exist (cf. [HTI97; SQP+11; ZAV04; Nat11]), most of them are tailored for a specific FI mechanism and the targeted SUT. As a result, testers tend to develop custom tools for new SUTs, either because no FI tool exists for that SUT or the effort to identify it among the large number of existing tools (and adjust it) exceeds the effort for re-implementation.

Schirmeier et al. [SHK+12] distinguish between *generalist* and *specialist* tools. Specialists (e.g., Xception [CMS98], FERRARI [KKA95], Ballista [KD99], LFI [MC09],

SAFE [NCD+13a]) support specific SUT classes and FI mechanisms and are highly customized for their intended use case. On the positive side, such specialist tools can provide very efficient implementations with a very low run time overhead, as they can exploit SUT-specific assumptions. Xception, for instance, makes use of special CPU debug registers to perform injections with minimal intrusiveness. On the downside, such high specialization causes a strong coupling between the FI tool and the SUT, which limits the reusability of the tool for experiments with other SUTs. In the case of Xception, the applicability of minimally intrusive injections relies on the presence of the required debug registers and functions in the targeted processor and cannot be used on hardware platforms that do not feature these mechanisms.

Generalist tools (e.g., GOOFI [AVF+01], NFTAPE [SFB+00], FAIL* [SHK+12]) are adaptable to various SUTs and FI mechanisms. They provide means to reuse common modules and avoid the overhead for their re-implementation. Generalists are built around an extensible architecture with interfaces for extensions and customization to tailor them for specific use cases.

Even though a number of generalist tools have been proposed, none of them has matured to a point where it could easily be applied to new classes of SUTs unforeseen by the original authors of the tools. In fact, even if generalist tools are designed for high portability and reusability, they often embed subtle and implicit assumptions about the SUT (e.g., assumptions about the OE and the failure modes of the SUT) that hamper their applicability for other SUTs. As a result, there is no generalist fault injection tool that is widely adopted outside the context of the research projects in which it was developed. Unfortunately, no previous experience report on the practical issues that arise when applying a generalist fault injection tool to a new SUT exists, thus providing little guidance for prospective users and researchers.

Chapter contribution: To fill this gap, this chapter reports on the architecture and usage of GRINDER, a generalist FI tool that was implemented to conduct FI experiments for different SUTs and that forms the basis of the PAIN experiment framework used to analyze experiment parallelization effects in Chapter 6. GRINDER is published under an open source license with the goal of lowering the bar for the adoption of fault injection by researchers and testers [DM15a]. GRINDER achieves reusability across diverse SUTs by introducing an abstract interface for SUT and OE control described in Section 7.3that requires SUT-specific implementations for each targeted SUT. To provide guidance on GRINDER's usage I briefly discuss the adaptation of GRINDER to the Android SUT in Section 7.4. To demonstrate the reusability of GRINDER I present physical SLOC counts for the reusable and SUT-specific parts of the FI framework implementations for four different FI scenarios with two different SUTs in Section 7.5.The numbers indicate that a large part of GRINDER is reusable across the different application scenarios (up to 72 %). In order to improve the performance of fault injection experiments and reduce interference across parallel experiments in the case of Android, the SUT-specific part of the framework had to provide alternative implementations for some of GRINDER's functions, thereby devi-

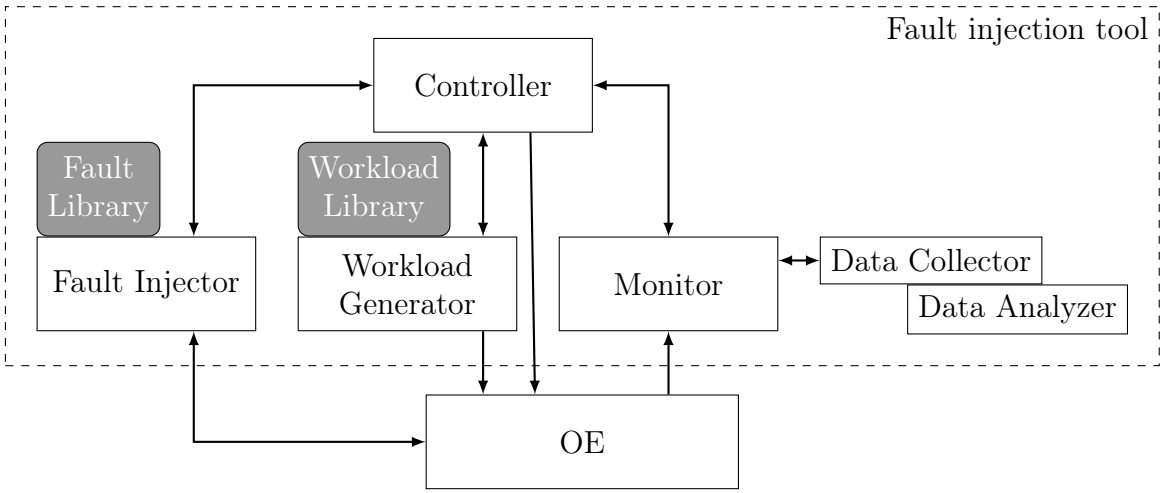

Figure 7.1: Common modules of a fault injection tool. Adopted from Hsueh, Tsai, and Iyer [HTI97] with modifications.

ating from the intended reuse. Nevertheless, even in this case the amount of reused code accounted for more than 54 % of the overall code for conducting the tests.

7.2 Reusability of Fault Injection Tools

Figure 7.1 shows the generic FI setup introduced in Section 3.1. As a different SUT often implies a different OE[1] (for the relationship between SUT and OE, see Section 3.1), all components of the FI tool that directly interact with the OE and contain at least some OE-dependent code usually cannot be reused if the SUT changes. As the OE contains the SUT, I collectively refer to OE dependence in the following, although changes in the OE that affect FI tool components mostly result from a changed SUT. OE-independent code, like reusable fault types or workloads, can be implemented in corresponding libraries. Components that have no direct OE-dependency (i.e., the controller, data collection, and data analysis) can be reused, unless implicit couplings with the OE exist, e.g., data analyses on SUT-specific data. Specifically, the controller and test data processing components bear a high potential for code reuse, whereas the fault injector, the workload generator, and the monitor bear a lesser potential for code reuse, as their direct interaction with the OE implies SUT-dependence to a certain degree.

[1]For instance, the Windows CE workloads and drivers from Chapters 4 and 5 could not be used with the Android kernel in Chapter 6

7.3 The GRINDER Test Tool

Based on experience building specialist FI tools for various SUTs [HJS02; JSM07b] and the frequently encountered need for reimplementing previously developed functionality, GRINDER (GeneRic fault INjection tool for DEpendability and Robustness assessments) has been developed as a generalist tool. Previous specialist tools were not reusable for new SUTs, such as the Android OS targeted in Chapter 6 or the AUTOSAR platform targeted by colleagues [PWM+12], and existing generalist tools were not publically available.

FAIL* [SHK+12] has been made openly available under an open source license after the work on GRINDER and PAIN had been mostly completed. However, although FAIL* is a generalist tool, it has been developed with the primary intention to emulate hardware failures and assumes the SUT to execute in an emulator, where the emulated hardware constitutes the OE, whose malfunction is simulated. This underlying assumption on the SUT interface via an emulator renders FAIL* inapplicable for testing systems that require execution on actual hardware platforms, such as real-time systems or systems for which no appropriate emulator is available, as was the case for the targeted AUTOSAR system. Circumventing this constraint would have required major changes to FAIL*'s architecture, making the development of a new generalist tool even retrospectively the better decision.

GRINDER is written in Java and follows the general FI tool architecture introduced in Figure 3.4. Like other state-of-the-art generalists, GRINDER implements the FI experiment flow independently from a specific OE and injection method. GRINDER provides two interfaces for OE interactions.

- Test stimuli (faultload and workload), OE configuration data, test logs, etc. are transmitted via a *communication channel*. Different communication channels can be utilized for different OEs. For all experiments with GRINDER to date TCP was used.

- To control OE components (e.g., start/halt[2], initiate and respond to interactions via the communication channel), GRINDER requires OE-specific code. To supply GRINDER with the required functions, testers implement a *TargetAbstraction* interface defined by GRINDER.

Both interfaces are discussed in the following.

7.3.1 Communication between GRINDER and the SUT

The end points of GRINDER's communication channels within the OE are interceptors (cf. Section 3.1). These are probes that can be used to expose the SUT to external faults at run time or monitor information about the SUT's state, including failure detection. This separation of mechanism (interception) from functionality (injection,

[2]for common operations on the SUT and OE components see Section 3.1

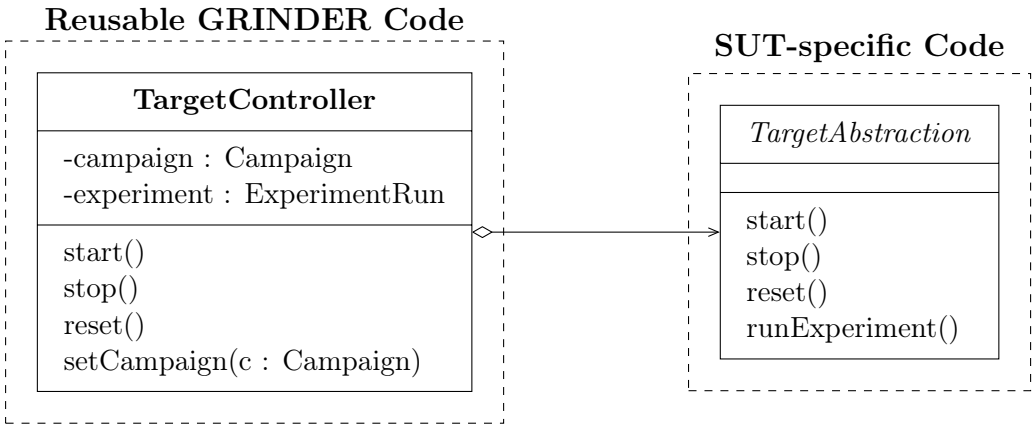

Figure 7.2: The TargetController class and the TargetAbstraction interface

monitoring) makes injectors and monitors reusable at different interceptor locations in the OE. If, for instance, debugger probes are used for interception, each such probe can be utilized to perform injections (e.g., by altering data) or to monitor the results of injections (e.g., by logging data).

Communication channels between interceptors and the FI framework are highly OE-dependent, at least on the lower layers of the protocol stack. If the OE is running in a virtual machine, the communication link may be a virtual network or a hypervisor-specific interface. If the OE is an embedded control system, network or debug connectors may be used. To support diverse implementations of the communication channel in GRINDER, testers can provide corresponding implementations of interceptors and the TargetAbstraction.

The process of placing interceptors in the OE, commonly referred to as *instrumentation*, is usually performed automatically by a separate tool that is not part of FI tool chain, due to the high dependency on implementation details of the OE. Debugging probes, for instance, can be inserted by compilers.

7.3.2 TargetAbstraction

The TargetAbstraction interface specifies a simple set of functions that must be implemented for an OE to be controllable by GRINDER. The specification of the TargetAbstraction was driven by the observation that, on an abstract level, the progression of FI experiments across different tools and SUTs is the same: SUT initialization, workload invocation, fault injection, and data collection, as outlined in Chapter 3. Consequently, the TargetAbstraction comprises these basic control functions that GRINDER requires to automatically run FI tests. The choice of these functions is based on a survey of FI-related literature (cf. [HTI97; SQP+11; ZAV04; Nat11]) and previous experience with FI experimentation.

Figure 7.2 shows the TargetAbstraction class that testers have to implement and GRINDER's TargetController, which directly maps the functions that the TargetAb-

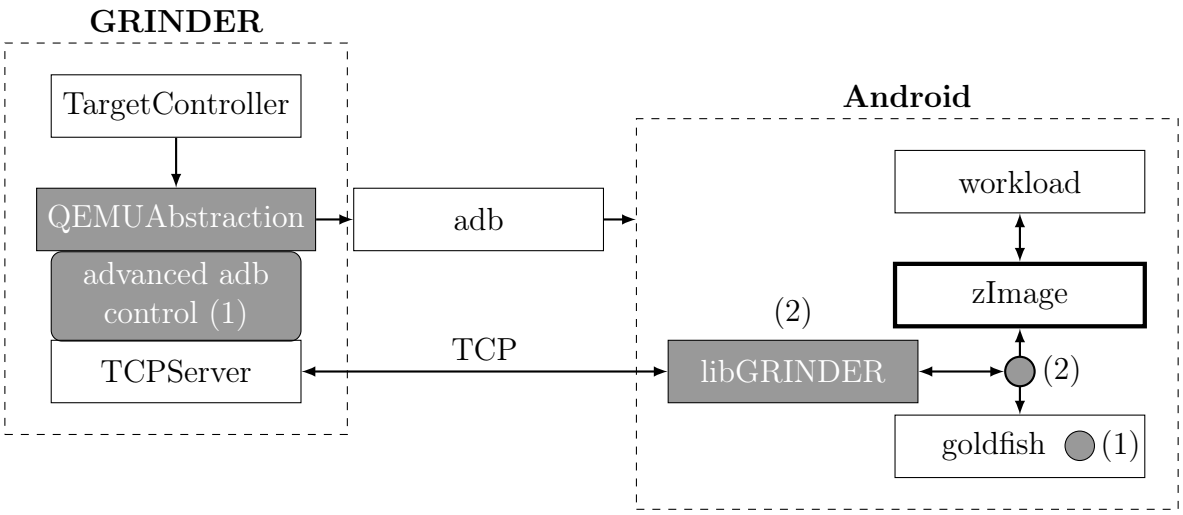

Figure 7.3: Adapting GRINDER to an Android SUT

straction provides to its control functions for conducting automated FI campaigns: GRINDER *starts* the SUT, *runs* an experiment, and stores data for offline analysis. The SUT is then *reset* to a known stable state to avoid the impact of undetected residual injection effects on subsequent runs. These steps are repeated for each run of the chosen campaign from GRINDER's campaign database for the targeted OE. After all tests from the database have been performed, the OE is *stopped* and exchanged or reconfigured, if this is required for subsequent campaigns. The framework automatically stops execution after the last specified test case. GRINDER's start operation includes (re-)setting the OE to its initial state. The run operation includes the workload execution. Injections and monitoring are performed upon activation of interceptors during workload execution.

7.4 GRINDER Adaptation to Android

GRINDER has been used to implement the PAIN framework (see Chapter 6) for Android [Gooa], which is a popular Linux-based operating system for mobile devices. Android's development tools ship with a QEMU-based emulator for convenient development without access to a mobile hardware platform. GRINDER runs on the same host as the emulator, controls the emulator process, and interacts with the Android system via the Android Debugging Bridge (adb).

Figure 7.3 gives an overview of the Android OE. The Android kernel (*zImage*) is the SUT. The *goldfish* SD card driver's misbehavior is simulated by FI. The *workload* component is an Android app that triggers kernel/driver interactions to activate injections. System instability is detected by monitoring existing Linux kernel mechanisms that signal *kernel oops/panic* (possibly triggered by the more sophisticated light/heavy detectors discussed in Section 6.3.1) failures via adb. The TargetAbstraction (*QEMUAbstraction*) is implemented by using QEMU's functions to start/stop/reset the

OE.

Two versions of the Android experiment setup with GRINDER have been implemented.

In Scenario 1 code mutations are used to emulate representative residual software defects [NCD+13a] in the driver. Injections are performed upfront on the driver source code. The OE is started, the corresponding mutant loaded, and the workload started to trigger the fault. If an error occurs, a corresponding message (workload error, kernel oops or panic) is parsed from the adb output and the test ends. If the workload finishes without error, this is also signaled via adb. If none of these events occur, a timeout triggers and signals a test hang. Upon any of these events, the emulator is shut down and the corresponding test result stored in GRINDER's test database. This setup has been used for the study presented in Chapter 6.

In Scenario 2 faults are injected by corrupting function call parameters in the interface between the driver and the kernel, similar to the studies presented in Chapters 4 and 5. The interceptor is implemented as an interface wrapper between these components. The injection logic is contained in libGRINDER, which is built as a Linux kernel module. GRINDER first starts the emulator, then loads the interceptor after boot-up, which in turn loads the driver. GRINDER then starts the workload, which eventually triggers kernel/driver interactions, which are altered for injections by the interceptor. The detection mechanisms are identical to Scenario 1.

7.5 Software Reuse

Besides Android, GRINDER has been used for automated FI assessments of an adaptive cruise control system based on AUTOSAR [AUT], the de facto standard for automotive systems. On AUTOSAR timing protection mechanisms were assessed in two scenarios against (Scenario 3) timing errors in applications and (Scenario 4) timing errors in the AUTOSAR OS kernel.

Table 7.1: Implementation effort in physical SLOC for five different FI assessment setups

Case Study	Reusable SLOC (GRINDER)	SUT/Scenario specific SLOC	Total
Android (Scenario 1)	2429	2028	4457
Android (Scenario 2)	2429	1121	3550
AUTOSAR (Scenario 3 & 4)	2429	523	2952
Android (specialist)	–	2464	2464

Table 7.1 illustrates the implementation effort for all four scenarios. All SLOC counts were generated using David A. Wheeler's SLOCCount. In Scenario 1 the

scenario-specific implementation effort accounts for 45.5 % of the overall code. With a corresponding reusable fraction of only 54.5 %, this scenario required the highest adaptation effort. The reason for this increase lies in performance optimizations that were applied to decrease the run time per test. To avoid the time-consuming transmission of mutated drivers from the host system to the emulator during the execution of each test, all mutants have been included in the emulator's data image. This change, however, required a number of extensions to the QEMUAbstraction, e.g., to detect when the system was available for loading the mutant. With a per-test transmission, this detection is implicit: if the system is able to receive TCP data, it is also ready to load modules. Moreover, the failure detection logic became significantly more complex due to longer run times in parallel execution of several tests (see Chapter 6).

The Android assessments with interface injections (Scenario 2) only require lightweight data transmission to configure injection runs, as only information on where and when which change should be applied to a parameter value is required. As a consequence, the scenario-specific code accounts for for only 1121 SLOC. More than 68 % of the overall code is reused from GRINDER's code base.

For both test scenarios on the AUTOSAR SUT, the SUT-specific implementations of the TargetAbstraction, the interceptors, and the communication channel accounted for 523 SLOC, which is less than 18 % of the overall framework implementation for these case studies. The main reason for the significantly lower implementation overhead of the injection logic is the diversity of the intercepted interfaces. While the Android interceptor covers a complex driver/kernel interface with 22 injection-relevant functions for a single driver, the interceptor logic was simple and mostly reusable across different experiments for the AUTOSAR scenarios.

The last line of the table shows the implementation overhead for a previously developed FI specialist tool that targeted the same SUT and injection scenario as Scenario 2 and was written in the same language as GRINDER and its extensions (Java). The scenario-specific implementation for the same scenario (Scenario 2) with GRINDER is smaller than 46 % of the specialist tool SLOC, indicating that the generalist tool implementation indeed saves re-implementation efforts.

7.6 Conclusion

In order to avoid re-implementation efforts for common functionality across fault injection setups for different SUTs, several *generalist* tools have been proposed. While the proposed tools have been applied for diverse FI scenarios by their authors, the corresponding efforts to adapt these generalists for each application scenario have not been assessed. In particular, testers cannot assess if the reuse of such a generalist would outweigh the implementation effort for a specialized tool from scratch. Moreover, as most generalists are not freely available, testers cannot conduct such assessments themselves.

To address this issue this chapter introduces GRINDER, a generalist FI tool that is freely available under an open source license. GRINDER features a generic SUT

interface that makes it portable to a large variety of SUTs, as demonstrated by its successful adoption in four different FI scenarios for two different SUTs. GRINDER's code reuse across these scenarios ranges between 54.5 % and 72 %.

8 Summary and Conclusion

The pervasiveness of software-controlled computing systems and their interconnection has increased massively over the past decade and, according to estimates [Cis15], will continue to increase similarly in the future. With the increasing impact these systems have on our daily lives, our reliance on their dependable operation, i.e., the absence of unacceptable failures, also grows. To ensure the absence of unacceptable failures, computing systems need to fulfill two properties. They need to behave according to their specifications (*correctness*) and must not behave dangerously in the case on unspecified operational conditions (*robustness*). Similar to testing approaches for assessing and improving the correctness of software, fault injection (FI) is an experimental technique for assessing and improving its robustness by exposing software to external faults and monitoring its response. Depending on the assessment purpose, FI-based assessments fall in two classes. For one class, termed *operational FI* in this thesis, the purpose is to assess the software's behavior for a known application context. The focus of my thesis is on the second class, termed *debug FI*, for which the purpose is to identify as many robustness vulnerabilities with as few experiments as possible.

Although a large number of frameworks and fault models for debug FI exist, virtually all of them are based on a *single fault assumption*, i.e., the system under test (SUT) is only exposed to a single fault during each experiment. While this assumption has practical advantages for the interpretation of FI experiment results, as observed SUT behavior can always be attributed to the single considered fault condition it was exposed to, the exclusion of SUT sensitivity to coincident external faults *by assumption* bears the risk of residual robustness vulnerabilities that escape detection in the FI assessment if the assumption does not hold. On this background, my thesis addresses the following research questions.

Research Question (RQ1): How can the effectiveness (and efficiency) of fault models for debug FI be assessed?

In order to investigate if the single fault assumption commonly found in debug FI assessments is justified, corresponding fault models that violate the assumption have to be developed and compared against the classically applied models. The assumption is not justified if its violation improves the effectiveness of debug FI assessments or their performance or both. In order to investigate their relative effectiveness and performance, a corresponding set of metrics for comparative fault model evaluations is required. Inspired by related work on test adequacy criteria for correctness testing, Chapter 4 makes the following contribution.

Contribution (C1): A set of metrics to quantify fault model effectiveness and cost

Although fault models have been compared in existing literature, the applied criteria either targeted representativeness [MBD+06], which is of lesser importance for debug FI, or propagation measures [JSM07b], which only capture a fraction of debug FI effectiveness.

I propose a set of four metrics to capture the efficiency, i.e., the effectiveness and performance, of fault models for debug FI, spanning interface coverage, injection efficiency, experiment execution time, and implementation complexity. Except for the implementation complexity of fault models, for which existing measurement tools are referenced, the proposed metrics do not require additional measurements beyond those required for the actual robustness assessments. All metrics provide ratio scale measures, which makes them suitable for combination in derived metrics. To overcome the impossibility to state a normalized coverage metric due to the unknown number of vulnerable services in a SUT, I adopt ideas from correctness test adequacy criteria and relate observed vulnerabilities to the number of services provided by the SUT's interfaces.

I demonstrate the utility of the proposed metrics by comparatively evaluating four classical fault models according to their efficiency in assessing the robustness of Windows CE's driver interface.

Research Question (RQ2): Is debug FI with complex fault conditions more effective than the classical fault models?

The question directly targets the single fault assumption. If complex fault conditions prove effective for revealing robustness issues, the single fault assumption should be reconsidered. Using the previously developed metrics for the efficiency of debug FI fault models, complex and classical fault models can be comparatively evaluated.

Contribution (C2): A taxonomy for fault coincidence and framework for fault model combination

In order to investigate the validity of the single fault assumption, corresponding fault models that constitute the intended complex fault conditions are required. In Chapter 5, I develop a notion of fault coincidence by stating external fault properties of temporal and spatial spread that faults need to fulfill in order to violate the single fault assumption. Based on this coincidence notion, I derive three temporally coincident and spatially spread *higher order fault models* that combine classically applied fuzzing and bit flip fault models. I use these higher order models to conduct robustness assessments on Windows CE's driver interface, evaluate their efficiency according to the previously identified metrics, and compare them against the classical fault models they are composed of. The results show significant differences in coverage, indicating that the single fault assumption prevents the identification of non-robust services in Windows CE's driver interface and should, consequently, be dropped for this SUT. The

observed violation of the single fault assumption counterproves its universal validity and suggests a re-assessment of the assumption for other SUTs.

Although the developed higher order models identify robustness issues that classical models do not detect, they also miss a number of service vulnerabilities that the other fault models detect. Furthermore, a conducted analysis of fault masking and amplification effects of higher order fault effects indicates that additional experiments with classical models are required to derive accurate cause-effect relations, e.g., for the efficient placement of fault containment wrappers. Due to the observed masking and amplification effects, it is unclear whether the cause of an observed failure is caused by a higher order fault or by a subset of the individual faults that a higher order fault is composed of. Both the incomplete coverage of higher order fault models and the ambiguity they introduce necessitate additional experiments with classical models, which leads to an increased number of relevant faults to test with if higher order models are used.

However, due to their achieved coverage of previously undetected robustness issues, higher order fault models cannot be disregarded, either. In order to cope with the expected overhead their adoption implies, I investigate the following research question.

Research Question (RQ3): Can parallel hardware help to increase the efficiency of debug FI to mitigate the overhead that higher order fault models entail?

To mitigate the increased experiment counts resulting from the adoption of higher order fault models in debug FI assessments, I propose to exploit parallel hardware by concurrent experiment executions. Although parallel hardware has been successfully exploited to speed up correctness testing, its adoption for FI-based assessments is not trivial. In contrast to unit and integration tests, for which test parallelization has been mostly applied, FI-based assessments are system level tests of longer run times and higher hardware utilization. In addition, FI-based assessments require a complete re-initialization of the SUT and OE between experiments to prevent dormant faults and errors from affecting subsequent experiments. In the case of OSs, this entails a complete OS reboot, which adds to the experiment run time and hardware utilization. For these reasons, FI experiments and other system level tests likely cause higher interference on shared resources when executing on parallel computing platforms, on which often only processing units are replicated. If these interferences lead to deviations of the experiment results, throughput improvements only imply that wrong results are obtained faster. The applicability of parallelization to speed up experiment executions, therefore, depends on both the throughput and the integrity of results.

Contribution (C3): A framework for parallel FI experiments

In Chapter 6, I propose PAIN, a framework for PArallel fault INjections to exploit the potential of parallel hardware for improved experiment throughput. To ensure

that PAIN has no adverse effects on experiment results, the *metrological compatibility*, i.e., result equivalence, of parallel and sequential test executions is assessed along with performance measures.

The conducted robustness assessment of the Android OS's driver interface reveals that while considerable performance improvements are achieved, the results obtained from parallel experiment executions significantly differ from the sequential case. A more detailed analysis identifies timeout configurations in the applied hang failure detectors as the cause behind the observed result deviations. The respective timeouts need to be long enough to avoid false positives. However, if they are too long, the detection of hang failures is delayed and long execution times result for the corresponding experiments, thereby at least partially negating the throughput gains from parallel execution.

To overcome the issue, a systematic timeout identification approach is proposed that relies on time measurements from a series of parallel experiment executions with the intended degree of parallelism and on the targeted hardware platform. For the conducted study, reliable timeout estimates are achieved from a campaign of around 800 experiments, which entail a moderate calibration overhead, as they run in parallel and benefit from the according throughput improvements.

Contribution (C4): A generic FI tool

The choice of a different SUT, e.g., Android instead of Windows CE, requires the re-implementation the applied FI tool, as FI tools are usually highly coupled with their SUT. However, as most of these tools share common functionality, a modular framework would allow for code reuse if changes to the SUT only had to be reflected by few interfacing modules. With this motivation, a number of FI *generalist* tools have been proposed in the literature. Unfortunately, none of the proposed tools was openly accessible at the time the development of the PAIN framework began. In Chapter 7 I introduce GRINDER (GeneRic fault INjection tool for DEpendability and Robustness assessments), a generalist FI tool that specifies an abstract interface for SUT control. In order to adapt GRINDER for experiments with a new SUT, only this interface needs to be implemented in addition to SUT-specific instrumentation, which is not part of the FI framework and needs to be performed for every targeted SUT individually. The adaptation of GRINDER for Android and the AUTOSAR automotive platform shows that large parts of GRINDER can be reused for both SUTs if the standard interface control functions provided by GRINDER are used. In the case of PAIN, the interface between GRINDER and the SUT had to be extended in order to facilitate experiment control at run time and reduce experiment execution time, which led to a larger fraction of SUT-specific code. However, even in that case the reused code fraction from GRINDER accounted for 54.5 %

Both GRINDER and PAIN are made available on github [DM15a; DM15b] under the AGPL v3 license.

Bibliography

[AAA+90] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. "Fault injection for dependability validation: a methodology and some applications". In: *Software Engineering, IEEE Transactions on* 16.2 (Feb. 1990), pp. 166–182. ISSN: 0098-5589.

[AAF04] A. Albinet, J. Arlat, and J.-C. Fabre. "Characterization of the impact of faulty drivers on the robustness of the Linux kernel". In: *Dependable Systems and Networks, 2004 International Conference on*. June 2004, pp. 867–876. DOI: 10.1109/DSN.2004.1311957.

[ABL05] J. Andrews, L. Briand, and Y. Labiche. "Is mutation an appropriate tool for testing experiments?" In: *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. May 2005, pp. 402–411. DOI: 10.1109/ICSE.2005.1553583.

[ACD+14] K. Androutsopoulos, D. Clark, H. Dan, R. M. Hierons, and M. Harman. "An Analysis of the Relationship Between Conditional Entropy and Failed Error Propagation in Software Testing". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 573–583. ISBN: 978-1-4503-2756-5.

[Aer12] R. T. C. for Aeronautics. *DO-178C – Software Considerations in Airborne Systems and Equipment Certification*. English. Radio Technical Commission for Aeronautics, 2012.

[Aer92] R. T. C. for Aeronautics. *DO-178B*. English. Radio Technical Commission for Aeronautics, 1992.

[AFR02] J. Arlat, J.-C. Fabre, and M. Rodriguez. "Dependability of COTS microkernel-based systems". In: *Computers, IEEE Transactions on* 51.2 (Feb. 2002), pp. 138–163. ISSN: 0018-9340. DOI: 10.1109/12.980005.

[AK88] P. Ammann and J. Knight. "Data diversity: an approach to software fault tolerance". In: *Computers, IEEE Transactions on* 37.4 (Apr. 1988), pp. 418–425. ISSN: 0018-9340. DOI: 10.1109/12.2185.

[ALR+04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *Dependable and Secure Computing, IEEE Transactions on* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971.

[App15] Apple Support Communities Forum. *iOS 8 Calendar on iPhone showing GMT times for new events.* 2015. URL: https://discussions.apple.com/message/26652110 (visited on 03/09/2015).

[Arm72] D. Armstrong. "A Deductive Method for Simulating Faults in Logic Circuits". In: *Computers, IEEE Transactions on* C-21.5 (May 1972), pp. 464–471. ISSN: 0018-9340.

[Ash95] N. Ashley. *Measurement as a powerful software management tool.* The McGraw-Hill International Software Quality Assurance Series. McGraw-Hill, 1995. ISBN: 9780077079024.

[AUT] AUTOSAR development cooperation. *Official AUTOSAR Website.* http://www.autosar.org. (Visited on 03/09/2015).

[AVF+01] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. "GOOFI: generic object-oriented fault injection tool". In: *Dependable Systems and Networks, 2001. DSN 2001. International Conference on.* July 2001, pp. 83–88. DOI: 10.1109/DSN.2001.941394.

[Avi78] A. Avizienis. "Fault-tolerance: The survival attribute of digital systems". In: *Proceedings of the IEEE* 66.10 (Oct. 1978), pp. 1109–1125. ISSN: 0018-9219.

[BBC+06] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. "Thorough Static Analysis of Device Drivers". In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006.* EuroSys '06. Leuven, Belgium: ACM, 2006, pp. 73–85. ISBN: 1-59593-322-0.

[BC12] R. Banabic and G. Candea. "Fast Black-box Testing of System Recovery Code". In: *Proceedings of the 7th ACM European Conference on Computer Systems.* EuroSys '12. Bern, Switzerland: ACM, 2012, pp. 281–294. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168865. URL: http://doi.acm.org/10.1145/2168836.2168865.

[BCC+11] A. Bovenzi, M. Cinque, D. Cotroneo, R. Natella, and G. Carrozza. "OS-level hang detection in complex software systems". In: *International Journal of Critical Computer-Based Systems* 2.3 (Jan. 2011), pp. 352–377. DOI: 10.1504/IJCCBS.2011.042333. URL: http://dx.doi.org/10.1504/IJCCBS.2011.042333.

[BCF+07] A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi. "Foundations of Measurement Theory Applied to the Evaluation of Dependability Attributes". In: *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on.* June 2007, pp. 522–533. DOI: 10.1109/DSN.2007.52.

[BCH+00] B. W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, R. Madachy, and B. Steece. *Software Cost Estimation with Cocomo II.* 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000. ISBN: 0130266922.

[BCL+04] T. Ball, B. Cook, V. Levin, and S. Rajamani. "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft". English. In: *Integrated Formal Methods*. Ed. by E. Boiten, J. Derrick, and G. Smith. Vol. 2999. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 1–20. ISBN: 978-3-540-21377-2.

[BCS+90] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek. "Fault injection experiments using FIAT". In: *Computers, IEEE Transactions on* 39.4 (Apr. 1990), pp. 575–582. ISSN: 0018-9340. DOI: 10.1109/12.54853.

[Bel] F. Bellard. *QEMU*. URL: http://wiki.qemu.org/Main_Page (visited on 03/09/2015).

[Bel89] C. M. Belcastro. *Laboratory test methodology for evaluating the effects of electromagnetic disturbances on fault-tolerant control systems*. Tech. rep. NASA-TM-101665. NASA Langley Research Center; Hampton, VA, United States, 1989.

[BH95] Y. Benjamini and Y. Hochberg. "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 57.1 (1995), pp. 289–300.

[BKK+10] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato. "D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology". In: *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 631–636. ISBN: 978-0-7695-4039-9. DOI: 10.1109/CCGRID.2010.72. URL: http://dx.doi.org/10.1109/CCGRID.2010.72.

[Bor05] S. Borkar. "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation". In: *IEEE Micro* 25.6 (Nov. 2005), pp. 10–16. ISSN: 0272-1732.

[BSS13] C. Borchert, H. Schirmeier, and O. Spinczyk. "Generative Software-based Memory Error Detection and Correction for Operating System Data Structures". In: *Proceedings of the 43nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*. (Budapest, Hungary). IEEE Computer Society Press, June 2013. ISBN: 978-1-4673-6471-3.

[CB89] R. Chillarege and N. Bowen. "Understanding large system failures-a fault injection experiment". In: *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*. June 1989, pp. 356–363.

[CBC+92] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M.-Y. Wong. "Orthogonal defect classification-a concept for in-process measurements". In: *Software Engineering, IEEE Transactions on* 18.11 (Nov. 1992), pp. 943–956. ISSN: 0098-5589.

[CBZ10] G. Candea, S. Bucur, and C. Zamfir. "Automated Software Testing As a Service". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 155–160. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807153. URL: http://doi.acm.org/10.1145/1807128.1807153.

[CC96] J. Christmansson and R. Chillarege. "Generation of an error set that emulates software faults based on field data". In: *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*. June 1996, pp. 304–313. DOI: 10.1109/FTCS.1996.534615.

[CCM+09] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. "Fast Byte-granularity Software Fault Isolation". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 45–58. ISBN: 978-1-60558-752-3.

[CDC+03] G. Candea, M. Delgado, M. Chen, and A. Fox. "Automatic failure-path inference: a generic introspection technique for Internet applications". In: *Internet Applications. WIAPP 2003. Proceedings. The Third IEEE Workshop on*. June 2003, pp. 132–141. DOI: 10.1109/WIAPP.2003.1210298.

[CGN+13] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. Trivedi. "Fault triggers in open-source software: An experience report". In: *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. Nov. 2013, pp. 178–187. DOI: 10.1109/ISSRE.2013.6698917.

[Cha09] R. N. Charette. "This car runs on code". In: *IEEE Spectrum* 46.3 (2009), p. 3.

[CHR82] L. Clarke, J. Hassell, and D. Richardson. "A Close Look at Domain Testing". In: *Software Engineering, IEEE Transactions on* SE-8.4 (July 1982), pp. 380–390. ISSN: 0098-5589.

[Cis15] Cisco. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2014–2019*. 2015. URL: http://cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html (visited on 02/11/2015).

[CKC91] R. Chillarege, W.-L. Kao, and R. G. Condit. "Defect Type and Its Impact on the Growth Curve". In: *Proceedings of the 13th International Conference on Software Engineering*. ICSE '91. Austin, Texas, USA: IEEE Computer Society Press, 1991, pp. 246–255. ISBN: 0-89791-391-4.

[CLJ+04] R. Chandra, R. Lefever, K. Joshi, M. Cukier, and W. Sanders. "A global-state-triggered fault injector for distributed system evaluation". In: *Parallel and Distributed Systems, IEEE Transactions on* 15.7 (July 2004), pp. 593–605. ISSN: 1045-9219. DOI: 10.1109/TPDS.2004.14.

[CLN+12] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa. "Experimental Analysis of Binary-Level Software Fault Injection in Complex Software". In: *Dependable Computing Conference (EDCC), 2012 Ninth European*. May 2012, pp. 162–172. DOI: 10.1109/EDCC.2012.12.

[CLZ+10] J. Chen, Q. Li, J. Zhao, and X. Li. "Test Adequacy Criterion Based on Coincidental Correctness Probability". In: *Proceedings of the Second Asia-Pacific Symposium on Internetware*. Internetware '10. Suzhou, China: ACM, 2010, 20:1–20:4. ISBN: 978-1-4503-0694-2.

[CM13] D. Cotroneo and H. Madeira. "Introduction to Software Fault Injection". English. In: *Innovative Technologies for Dependable OTS-Based Critical Systems*. Ed. by D. Cotroneo. Springer Milan, 2013, pp. 1–15. ISBN: 978-88-470-2771-8.

[CMR13] M. Carbin, S. Misailovic, and M. C. Rinard. "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. '13. New York, NY, USA: ACM, 2013, pp. 33–52. ISBN: 978-1-4503-2374-1. (Visited on 02/11/2015).

[CMS98] J. Carreira, H. Madeira, and J. G. Silva. "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers". In: *IEEE Trans. Softw. Eng.* 24.2 (Feb. 1998), pp. 125–136. ISSN: 0098-5589.

[CN13] D. Cotroneo and R. Natella. "Fault Injection for Software Certification". In: *Security Privacy, IEEE* 11.4 (July 2013), pp. 38–45. ISSN: 1540-7993. DOI: 10.1109/MSP.2013.54.

[CNR+13] D. Cotroneo, R. Natella, S. Russo, and F. Scippacercola. "State-Driven Testing of Distributed Systems". English. In: *Principles of Distributed Systems*. Ed. by R. Baldoni, N. Nisse, and M. van Steen. Vol. 8304. Lecture Notes in Computer Science. Springer International Publishing, 2013, pp. 114–128. ISBN: 978-3-319-03849-0. DOI: 10.1007/978-3-319-03850-6_9. URL: http://dx.doi.org/10.1007/978-3-319-03850-6_9.

[CNR09] D. Cotroneo, R. Natella, and S. Russo. "Assessment and Improvement of Hang Detection in the Linux Operating System". In: *Reliable Distributed Systems, 2009. SRDS '09. 28th IEEE International Symposium on*. Sept. 2009, pp. 288–294. DOI: 10.1109/SRDS.2009.26.

[Coc70] J. Cocke. "Global Common Subexpression Elimination". In: *SIGPLAN Not.* 5.7 (July 1970), pp. 20–24. ISSN: 0362-1340.

[CPY07] R.-Y. Chang, A. Podgurski, and J. Yang. "Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software". In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA '07. London, United Kingdom: ACM, 2007, pp. 163–173. ISBN: 978-1-59593-734-6.

[CVM+03] P. Costa, M. Vieira, H. Madeira, and J. Silva. "Plug and Play Fault Injector for Dependability Benchmarking". English. In: *Dependable Computing*. Ed. by R. de Lemos, T. Weber, and J. Camargo JoãoBatista. Vol. 2847. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 8–22. ISBN: 978-3-540-20224-0. DOI: 10.1007/978-3-540-45214-0_4. URL: http://dx.doi.org/10.1007/978-3-540-45214-0_4.

[CYC+01] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. "An Empirical Study of Operating Systems Errors". In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP '01. Banff, Alberta, Canada: ACM, 2001, pp. 73–88. ISBN: 1-58113-389-8. DOI: 10.1145/502034.502042. URL: http://doi.acm.org/10.1145/502034.502042.

[CZB+10] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. "Cloud9: A Software Testing Service". In: *SIGOPS Oper. Syst. Rev.* 43.4 (Jan. 2010), pp. 5–10. ISSN: 0163-5980. DOI: 10.1145/1713254.1713257. URL: http://doi.acm.org/10.1145/1713254.1713257.

[Dal00] D. Dalcher. "Smooth Seas - Rough Sailing: The Case of the Lame Ship". In: *Engineering of Computer Based Systems, 2000. (ECBS 2000) Proceedings. Seventh IEEE International Conference and Workshopon the*. 2000, pp. 393–394.

[DAS+12] D. Di Leo, F. Ayatolahi, B. Sangchoolie, J. Karlsson, and R. Johansson. "On the Impact of Hardware Faults – An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions". English. In: *Computer Safety, Reliability, and Security*. Ed. by F. Ortmeier and P. Daniel. Vol. 7612. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 198–209. ISBN: 978-3-642-33677-5. DOI: 10.1007/978-3-642-33678-2_17. URL: http://dx.doi.org/10.1007/978-3-642-33678-2_17.

[DCB+05] A. Duarte, W. Cirne, F. Brasileiro, and P. Machado. "Using the Computational Grid to Speed up Software Testing". In: *Proceedings of 19th Brazilian Symposium on Software Engineering*. 2005.

[DCB+06] A. Duarte, W. Cirne, F. Brasileiro, and P. Machado. "GridUnit: Software Testing on the Grid". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, 2006, pp. 779–782. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134410. URL: http://doi.acm.org/10.1145/1134285.1134410.

[DEE] DEEDS Group. *Robustness Evaluation of Windows CE using Simultaneous Fault Injections*. http://www.deeds.informatik.tu-darmstadt.de/simFI.

[Dij68] E. W. Dijkstra. "The Structure of the THE-multiprogramming System". In: *Commun. ACM* 11.5 (May 1968), pp. 341–346. ISSN: 0001-0782.

[Dij72] E. W. Dijkstra. "Structured Programming". In: ed. by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. London, UK, UK: Academic Press Ltd., 1972. Chap. Chapter I: Notes on Structured Programming, pp. 1–82. ISBN: 0-12-200550-3.

[DM02] J. Duraes and H. Madeira. "Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation". In: *Dependable Computing, 2002. Proceedings. 2002 Pacific Rim International Symposium on*. Dec. 2002, pp. 201–209. DOI: 10.1109/PRDC.2002.1185639.

[DM03a] J. Duraes and H. Madeira. "Definition of software fault emulation operators: a field data study". In: *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*. June 2003, pp. 105–114. DOI: 10.1109/DSN.2003.1209922.

[DM03b] J. Durães and H. Madeira. "Multidimensional characterization of the impact of faulty drivers on the operating systems behavior". In: *IEICE Transactions on Information and Systems* 86.12 (2003), pp. 2563–2570.

[DM06] J. Duraes and H. Madeira. "Emulation of Software Faults: A Field Data Study and a Practical Approach". In: *Software Engineering, IEEE Transactions on* 32.11 (Nov. 2006), pp. 849–867. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.113.

[DM15a] DEEDS/TUD and Mobilab/UniNa. *GRINDER – GeneRic fault INjection tool for DEpendability and Robustness assessments*. https://github.com/DEEDS-TUD/GRINDER.git. 2015. (Visited on 03/09/2015).

[DM15b] DEEDS/TUD and Mobilab/UniNa. *PAIN Software Framework*. https://github.com/DEEDS-TUD/PAIN.git. 2015. (Visited on 03/09/2015).

[DMP+01] M. Delamaro, J. Maldonado, A. Pasquini, and A. Mathur. "Interface Mutation Test Adequacy Criterion: An Empirical Evaluation". English. In: *Empirical Software Engineering* 6.2 (2001), pp. 111–142. ISSN: 1382-3256. DOI: 10.1023/A:1011429104252. URL: http://dx.doi.org/10.1023/A:1011429104252.

[DR06] H. Do and G. Rothermel. "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques". In: *Software Engineering, IEEE Transactions on* 32.9 (Sept. 2006), pp. 733–752. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.92.

[Dri10] K. R. Driscoll. "Murphy Was an Optimist". In: *Proceedings of the 29th International Conference on Computer Safety, Reliability, and Security*. SAFECOMP'10. Vienna, Austria: Springer-Verlag, 2010, pp. 481–482.

[DVM04] J. Durães, M. Vieira, and H. Madeira. "Dependability Benchmarking of Web-Servers". English. In: *Computer Safety, Reliability, and Security*. Ed. by M. Heisel, P. Liggesmeyer, and S. Wittmann. Vol. 3219. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 297–310. ISBN: 978-3-540-23176-9. DOI: 10.1007/978-3-540-30138-7_25. URL: http://dx.doi.org/10.1007/978-3-540-30138-7_25.

[DWB+06] A. Duarte, G. Wagner, F. Brasileiro, and W. Cirne. "Multi-environment Software Testing on the Grid". In: *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*. PADTAD '06. Portland, Maine, USA: ACM, 2006, pp. 61–68. ISBN: 1-59593-414-6. DOI: 10.1145/1147403.1147415. URL: http://doi.acm.org/10.1145/1147403.1147415.

[FGA+10] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. "Shoestring: Probabilistic Soft Error Reliability on the Cheap". In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. XV. New York, NY, USA: ACM, 2010, pp. 385–396. ISBN: 978-1-60558-839-1. (Visited on 02/11/2015).

[FHL+98] P. Frankl, R. Hamlet, B. Littlewood, and L. Strigini. "Evaluating testing methods by delivered reliability". In: *Software Engineering, IEEE Transactions on* 24.8 (Aug. 1998), pp. 586–601. ISSN: 0098-5589.

[FSM+99] J.-C. Fabre, F. Salles, M. Moreno, and J. Arlat. "Assessment of COTS microkernels by fault injection". In: *Dependable Computing for Critical Applications 7, 1999*. Jan. 1999, pp. 25–44.

[FX02] C. Fetzer and Z. Xiao. "An automated approach to increasing the robustness of C libraries". In: *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. 2002, pp. 155–164. DOI: 10.1109/DSN.2002.1028896.

[GCG+06] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. "Enforcing Performance Isolation Across Virtual Machines in Xen". English. In: *Middleware 2006*. Ed. by M. van Steen and M. Henning. Vol. 4290. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 342–362. ISBN: 978-3-540-49023-4. DOI: 10.1007/11925071_18. URL: http://dx.doi.org/10.1007/11925071_18.

[GDJ+11] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. "FATE and DESTINI: A Framework for Cloud Recovery Testing". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Imple-*

mentation. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 238–252. URL: http://dl.acm.org/citation.cfm?id=1972457.1972482.

[Ger03] A. German. "Software static code analysis lessons learned". In: *Crosstalk* 16.11 (2003).

[GGP06] A. Ganapathi, V. Ganapathi, and D. Patterson. "Windows XP Kernel Crash Analysis". In: *Proceedings of the 20th Conference on Large Installation System Administration*. LISA '06. Washington, DC: USENIX Association, 2006, pp. 12–12. URL: http://dl.acm.org/citation.cfm?id=1267793.1267805.

[GKT13] C. Giuffrida, A. Kuijsten, and A. Tanenbaum. "EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments". In: *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*. Dec. 2013, pp. 31–40.

[GKT89] U. Gunneflo, J. Karlsson, and J. Torin. "Evaluation of error detection schemes using fault injection by heavy-ion radiation". In: *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*. June 1989, pp. 340–347.

[Gooa] Google Inc. *Android*. http://www.android.com. (Visited on 03/09/2015).

[Goob] Google Inc. *Android Emulator*. URL: http://developer.android.com/tools/help/emulator.html (visited on 03/09/2015).

[Gooc] Google Inc. *android Git repositories*. https://android.googlesource.com/. (Visited on 03/09/2015).

[Gra86] J. Gray. *Why do computers stop and what can be done about it?* Tech. rep. TR-85.7. Tandem Computers, 1986.

[Gro+04] N. S. Group et al. "Technical analysis of the August 14, 2003, blackout: What happened, why, and what did we learn". In: *report to the NERC Board of Trustees* (2004).

[GSS98] A. Ghosh, M. Schmid, and V. Shah. "Testing the robustness of Windows NT software". In: *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*. Nov. 1998, pp. 231–235. DOI: 10.1109/ISSRE.1998.730886.

[GT07] M. Grottke and K. Trivedi. "Fighting bugs: remove, retry, replicate, and rejuvenate". In: *Computer* 40.2 (Feb. 2007), pp. 107–109. ISSN: 0018-9162. DOI: 10.1109/MC.2007.55.

[HBG+09] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum. "Fault isolation for device drivers". In: *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*. June 2009, pp. 33–42.

[HBK+10] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato. "Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems". In: *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*. Apr. 2010, pp. 428–433. DOI: 10.1109/ICSTW.2010.59.

[Hec93] H. Hecht. "Rare conditions-an important cause of failures". In: *Computer Assurance, 1993. COMPASS '93, Practical Paths to Assurance. Proceedings of the Eighth Annual Conference on*. June 1993, pp. 81–85. DOI: 10.1109/CMPASS.1993.288855.

[HGW04] M. Heimdahl, D. George, and R. Weber. "Specification test coverage adequacy criteria = specification test generation inadequacy criteria". In: *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*. Mar. 2004, pp. 178–186.

[HJS01] M. Hiller, A. Jhumka, and N. Suri. "An approach for analysing the propagation of data errors in software". In: *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. July 2001, pp. 161–170. DOI: 10.1109/DSN.2001.941402.

[HJS02] M. Hiller, A. Jhumka, and N. Suri. "PROPANE: An Environment for Examining the Propagation of Errors in Software". In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '02. Roma, Italy: ACM, 2002, pp. 81–85. ISBN: 1-58113-562-9. DOI: 10.1145/566172.566184. URL: http://doi.acm.org/10.1145/566172.566184.

[HL13] Q. Huang and P. P. Lee. "An Experimental Study of Cascading Performance Interference in a Virtualized Environment". In: *SIGMETRICS Perform. Eval. Rev.* 40.4 (Apr. 2013), pp. 43–52. ISSN: 0163-5999. DOI: 10.1145/2479942.2479948. URL: http://doi.acm.org/10.1145/2479942.2479948.

[HS67] F. H. Hardie and R. J. Suhocki. "Design and Use of Fault Simulation for Saturn Computer Design". In: *Electronic Computers, IEEE Transactions on* EC-16.4 (Aug. 1967), pp. 412–429. ISSN: 0367-7508.

[HSR95] S. Han, K. Shin, and H. Rosenberg. "DOCTOR: an integrated software fault injection environment for distributed real-time systems". In: *Computer Performance and Dependability Symposium, 1995. Proceedings., International*. Apr. 1995, pp. 204–213.

[HTI97] M.-C. Hsueh, T. Tsai, and R. Iyer. "Fault injection techniques and tools". In: *Computer* 30.4 (Apr. 1997), pp. 75–82. ISSN: 0018-9162. DOI: 10.1109/2.585157.

[Hua75] J. C. Huang. "An Approach to Program Testing". In: *ACM Comput. Surv.* 7.3 (Sept. 1975), pp. 113–128. ISSN: 0360-0300. DOI: 10.1145/356651.356652. URL: http://doi.acm.org/10.1145/356651.356652.

[IDM+13] I. Irrera, J. Duraes, H. Madeira, and M. Vieira. "Assessing the Impact of Virtualization on the Generation of Failure Prediction Data". In: *Dependable Computing (LADC), 2013 Sixth Latin-American Symposium on*. Apr. 2013, pp. 92–97. DOI: 10.1109/LADC.2013.24.

[IEC99] IEC. *IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems*. English. International Electrotechnical Commission: 1999.

[ISO03a] ISO/IEC 19761:2003. *Software Engineering – COSMIC-FFP – A Functional Size Measurement Method*. 2003.

[ISO03b] ISO/IEC 20926:2003. *Software Engineering – IFPUG 4.1 Unadjusted Functional Size Measurement Method – Counting Practices Manual*. 2003.

[ISO11a] ISO 26262-1:2011. *Road vehicles – Functional safety – Part 1: Vocabulary*. ISO, Geneva, Switzerland, 2011.

[ISO11b] ISO 26262-5:2011. *Road vehicles – Functional safety – Part 5: Product development at the hardware level*. ISO, Geneva, Switzerland, 2011.

[ISO11c] ISO 26262-6:2011. *Road vehicles – Functional safety – Part 6: Product development at the software level*. ISO, Geneva, Switzerland, 2011.

[JGS11] P. Joshi, H. S. Gunawi, and K. Sen. "PREFAIL: A Programmable Tool for Multiple-failure Injection". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 171–188. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048082. URL: http://doi.acm.org/10.1145/2048066.2048082.

[JH08] Y. Jia and M. Harman. "Constructing Subtle Faults Using Higher Order Mutation Testing". In: *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*. Sept. 2008, pp. 249–258. DOI: 10.1109/SCAM.2008.36.

[JH09] Y. Jia and M. Harman. "Higher Order Mutation Testing". In: *Information and Software Technology* 51.10 (2009). Source Code Analysis and Manipulation, {SCAM} 2008, pp. 1379–1393. ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/j.infsof.2009.04.016. URL: http://www.sciencedirect.com/science/article/pii/S0950584909000688.

[JH11] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *Software Engineering, IEEE Transactions on* 37.5 (Sept. 2011), pp. 649–678. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.62.

[JKJ+10] H. Jo, H. Kim, J.-W. Jang, J. Lee, and S. Maeng. "Transparent Fault Tolerance of Device Drivers for Virtual Machines". In: *Computers, IEEE Transactions on* 59.11 (Nov. 2010), pp. 1466–1479. ISSN: 0018-9340.

[Joh08] A. Johansson. "Robustness Evaluation of Operating Systems". PhD thesis. TU Darmstadt, 2008.

[JS05] A. Johansson and N. Suri. "Error propagation profiling of operating systems". In: *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. June 2005, pp. 86–95. DOI: 10.1109/DSN.2005.45.

[JSJ+05] A. Johansson, A. Sârbu, A. Jhumka, and N. Suri. "On Enhancing the Robustness of Commercial Operating Systems". English. In: *Service Availability*. Ed. by M. Malek, M. Reitenspieß, and J. Kaiser. Vol. 3335. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 148–159. ISBN: 978-3-540-24420-2. DOI: 10.1007/978-3-540-30225-4_11. URL: http://dx.doi.org/10.1007/978-3-540-30225-4_11.

[JSM07a] A. Johansson, N. Suri, and B. Murphy. "On the Impact of Injection Triggers for OS Robustness Evaluation". In: *Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on*. Nov. 2007, pp. 127–126. DOI: 10.1109/ISSRE.2007.23.

[JSM07b] A. Johansson, N. Suri, and B. Murphy. "On the Selection of Error Model(s) for OS Robustness Evaluation". In: *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*. June 2007, pp. 502–511. DOI: 10.1109/DSN.2007.71.

[Kap01] G. M. Kapfhammer. "Automatically and transparently distributing the execution of regression test suites". In: *Proceedings of the 18th International Conference on Testing Computer Software*. 2001.

[KCK+05] K. Kanoun, Y. Crouzet, A. Kalakech, A.-E. Rugina, and P. Rumeau. "Benchmarking the dependability of Windows and Linux using PostMark/spl trade/ workloads". In: *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*. Nov. 2005, pp. 10–20.

[KD00] P. Koopman and J. DeVale. "The exception handling effectiveness of POSIX operating systems". In: *Software Engineering, IEEE Transactions on* 26.9 (Sept. 2000), pp. 837–848. ISSN: 0098-5589. DOI: 10.1109/32.877845.

[KD99] P. Koopman and J. DeVale. "Comparing the robustness of POSIX operating systems". In: *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*. June 1999, pp. 30–37. DOI: 10.1109/FTCS.1999.781031.

[KDD08] P. Koopman, K. Devale, and J. Devale. "Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project". In: *Dependability Benchmarking for Computer Systems*. John Wiley & Sons, Inc., 2008, pp. 201–226. ISBN: 9780470370506.

[KGT14] E. van der Kouwe, C. Giuffrida, and A. Tanenbaum. "Evaluating Distortion in Fault Injection Experiments". In: *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*. Jan. 2014, pp. 25–32. DOI: 10.1109/HASE.2014.13.

[KK07] K. Koster and D. C. Kao. "State Coverage: A Structural Test Adequacy Criterion for Behavior Checking". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. Dubrovnik, Croatia: ACM, 2007, pp. 541–544. ISBN: 978-1-59593-811-4.

[KKA92] G. Kanawati, N. Kanawati, and J. Abraham. "FERRARI: a tool for the validation of system dependability properties". In: *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*. July 1992, pp. 336–344.

[KKA95] G. Kanawati, N. Kanawati, and J. Abraham. "FERRARI: a flexible software-based fault and error injection system". In: *Computers, IEEE Transactions on* 44.2 (Feb. 1995), pp. 248–260. ISSN: 0018-9340. DOI: 10.1109/12.364536.

[KKC+04] A. Kalakech, K. Kanoun, Y. Crouzet, and J. Arlat. "Benchmarking the dependability of Windows NT4, 2000 and XP". In: *Dependable Systems and Networks, 2004 International Conference on*. June 2004, pp. 681–686. DOI: 10.1109/DSN.2004.1311938.

[KKS98] N. Kropp, P. Koopman, and D. Siewiorek. "Automated robustness testing of off-the-shelf software components". In: *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*. June 1998, pp. 230–239. DOI: 10.1109/FTCS.1998.689474.

[KLB00] M. Kaaniche, J.-C. Laprie, and J.-P. Blanquart. "Dependability engineering of complex computing systems". In: *Engineering of Complex Computer Systems, 2000. ICECCS 2000. Proceedings. Sixth IEEE International Conference on*. 2000, pp. 36–46. DOI: 10.1109/ICECCS.2000.873926.

[KM02] P. Koopman and H. Madeira. "Workshop on Dependability Benchmarking". In: *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*. IEEE Computer Society, 2002, pp. 790–791. ISBN: 0-7695-1597-5. URL: http://computer.org/proceedings/dsn/1597/15970790.pdf.

[KM06] T. Kelly and J. McDermid. "Software in Safety Critical Systems: Achievement and Prediction". In: *Nuclear Future* (2006).

[KNS10] M. de Kruijf, S. Nomura, and K. Sankaralingam. "Relax: An Architectural Framework for Software Recovery of Hardware Faults". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. '10. New York, NY, USA: ACM, 2010, pp. 497–508. ISBN: 978-1-4503-0053-7. (Visited on 02/11/2015).

[Koo] P. Koopman. *Ballista FAQ – Robustness Failure*. URL: `http://www.cs.cmu.edu/afs/cs/project/edrc-ballista/www/faq.html#failure` (visited on 02/28/2015).

[KRK+98] M. Kaâniche, L. Romano, Z. Kalbarczyk, R. Iyer, and R. Karcich. "A hierarchical approach for dependability analysis of a commercial cache-based RAID storage architecture". In: *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*. June 1998, pp. 6–15. DOI: `10.1109/FTCS.1998.689450`.

[KS08] K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society, 2008.

[KS09] M. D. Kruijf and K. Sankaralingam. "Exploring the synergy of emerging workloads and silicon reliability trends". In: *Proc. SELSE '09*. 2009.

[KSD+97] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. "Comparing operating systems using robustness benchmarks". In: *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on*. Oct. 1997, pp. 72–79. DOI: `10.1109/RELDIS.1997.632800`.

[KV10] J. Kinder and H. Veith. "Precise Static Analysis of Untrusted Driver Binaries". In: *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*. FMCAD '10. Lugano, Switzerland: FMCAD Inc, 2010, pp. 43–50.

[KWG04] D. Kuhn, D. Wallace, and J. Gallo A.M. "Software fault interactions and implications for software testing". In: *Software Engineering, IEEE Transactions on* 30.6 (June 2004), pp. 418–421. ISSN: 0098-5589. DOI: `10.1109/TSE.2004.24`.

[KZC12a] B. Kasikci, C. Zamfir, and G. Candea. "CORD: A Collaborative Framework for Distributed Data Race Detection". In: *Presented as part of the Eighth Workshop on Hot Topics in System Dependability*. Hollywood, CA: USENIX, 2012. URL: `https://www.usenix.org/conference/hotdep12/workshop-program/presentation/Kasikci`.

[KZC12b] B. Kasikci, C. Zamfir, and G. Candea. "Data Races vs. Data Race Bugs: Telling the Difference with Portend". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. London, England, UK: ACM, 2012, pp. 185–198. ISBN: 978-1-4503-0759-8.

[Las05] A. Lastovetsky. "Parallel testing of distributed software". In: *Information and Software Technology* 47.10 (2005), pp. 657–662. ISSN: 0950-5849. DOI: http://dx.doi.org/10.1016/j.infsof.2004.11.006. URL: http://www.sciencedirect.com/science/article/pii/S0950584904001752.

[LDZ+06] Y. Li, T. Dong, X. Zhang, Y.-d. Song, and X. Yuan. "Large-scale software unit testing on the grid". In: *Granular Computing, 2006 IEEE International Conference on*. May 2006, pp. 596–599. DOI: 10.1109/GRC.2006.1635873.

[LM03] R. R. Lutz and I. C. Mikulski. "Operational anomalies as a cause of safety-critical requirements evolution". In: *Journal of Systems and Software* 65.2 (2003), pp. 155–161. ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/S0164-1212(02)00057-2. URL: http://www.sciencedirect.com/science/article/pii/S0164121202000572.

[LNW+14] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri. "An Empirical Study of Injected Versus Actual Interface Errors". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. 2014, pp. 397–408.

[Lon] R. Longbottom. *Roy Longbottom's Android Benchmark Apps*. URL: http://www.roylongbottom.org.uk/android%20benchmarks.htm (visited on 03/09/2015).

[LPG+14] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers. "SDCTune: A Model for Predicting the SDC Proneness of an Application for Configurable Protection". In: *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES '14. New Delhi, India: ACM, 2014, 23:1–23:10. ISBN: 978-1-4503-3050-3.

[LUS+] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. "[Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines". In:

[LVT+11] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh. "Pervasive Detection of Process Races in Deployed Systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 353–367. ISBN: 978-1-4503-0977-6.

[MBD+06] 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: *Dependable Computing Conference, 2006. EDCC '06. Sixth European*. Oct. 2006, pp. 53–64. DOI: 10.1109/EDCC.2006.16.

[MC09] P. Marinescu and G. Candea. "LFI: A practical and general library-level fault injector". In: *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*. June 2009, pp. 379–388. DOI: 10.1109/DSN.2009.5270313.

[MC11] P. D. Marinescu and G. Candea. "Efficient Testing of Recovery Code Using Fault Injection". In: *ACM Trans. Comput. Syst.* 29.4 (Dec. 2011), 11:1–11:38. ISSN: 0734-2071. DOI: 10.1145/2063509.2063511. URL: http://doi.acm.org/10.1145/2063509.2063511.

[McC76] T. McCabe. "A Complexity Measure". In: *Software Engineering, IEEE Transactions on* SE-2.4 (Dec. 1976), pp. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[MEK+12] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. "A whitebox approach for automated security testing of Android applications on the cloud". In: *Automation of Software Test (AST), 2012 7th International Workshop on*. June 2012, pp. 22–28. DOI: 10.1109/IWAST.2012.6228986.

[Mil94] S. P. Miller. "Applicability of modified condition/decision coverage to software testing". English. In: *Software Engineering Journal* 9 (5 Sept. 1994), 193–200(7). ISSN: 0268-6961. URL: http://digital-library.theiet.org/content/journals/10.1049/sej.1994.0025.

[MK01] H. Madeira and P. Koopman. "Dependability Benchmarking: making choices in an n-dimensional problem space". In: *Proceedings of the first Workshop on Evaluating and Architecting System Dependability*. 2001.

[MN07] M. Mendonca and N. Neves. "Robustness Testing of the Windows DDK". In: *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*. June 2007, pp. 554–564. DOI: 10.1109/DSN.2007.85.

[Mog06] J. C. Mogul. "Emergent (Mis)Behavior vs. Complex Software Systems". In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. EuroSys '06. Leuven, Belgium: ACM, 2006, pp. 293–304. ISBN: 1-59593-322-0.

[MSDa] MSDN. *coredll Module*. http://msdn.microsoft.com/en-us/library/aa448387.aspx. (Visited on 03/09/2015).

[MSDb] MSDN. *Implementing CEDDK.dll*. http://msdn.microsoft.com/en-us/library/ms898217.aspx. (Visited on 03/09/2015).

[MSDc] MSDN. *Network Driver Functions*. http://msdn.microsoft.com/en-us/library/ms895631.aspx. (Visited on 03/09/2015).

[MSDd] MSDN. *Stream Interface Driver Implementation (Windows CE .NET 4.2)*. http://msdn.microsoft.com/en-us/library/ms895309.aspx. (Visited on 03/09/2015).

[Mus96] J. Musa. "Software-reliability-engineered testing". In: *Computer* 29.11 (Nov. 1996), pp. 61–68. ISSN: 0018-9162.

[Nat11] R. Natella. "Achieving Representative Faultloads in Software Fault Injection". PhD thesis. Università di Napoli Federico II, 2011.

[NC01] W. Ng and P. Chen. "The design and verification of the Rio file cache". In: *Computers, IEEE Transactions on* 50.4 (Apr. 2001), pp. 322–337. ISSN: 0018-9340. DOI: 10.1109/12.919278.

[NCD+10] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira. "Representativeness analysis of injected software faults in complex software". In: *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. June 2010, pp. 437–446. DOI: 10.1109/DSN.2010.5544282.

[NCD+13a] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira. "On Fault Representativeness of Software Fault Injection". In: *IEEE Transactions on Software Engineering* 39.1 (Jan. 2013), pp. 80–96. ISSN: 0098-5589.

[NCD+13b] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira. "On Fault Representativeness of Software Fault Injection". In: *Software Engineering, IEEE Transactions on* 39.1 (Jan. 2013), pp. 80–96. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.124.

[NVN+13] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. "DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. San Jose, CA: USENIX Association, 2013, pp. 219–230. URL: http://dl.acm.org/citation.cfm?id=2535461.2535489.

[OnL] On-Line Applications Research (OAR) Corporation. *RTEMS.com | An Open Real-Time Operating System*. URL: http://www.rtems.com/ (visited on 02/28/2015).

[OU10] M. Oriol and F. Ullah. "YETI on the Cloud". In: *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*. Apr. 2010, pp. 434–437. DOI: 10.1109/ICSTW.2010.68.

[Pan99] J. Pan. "The Dimensionality of Failures—A Fault Model for Characterizing Software Robustness". In: *Proc. Int'l Symp. Fault-Tolerant Computing*. 1999.

[PM10] M. Papadakis and N. Malevris. "An Empirical Evaluation of the First and Second Order Mutation Testing Strategies". In: *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*. Apr. 2010, pp. 90–99. DOI: 10.1109/ICSTW.2010.50.

[PTD+09] T. Parveen, S. Tilley, N. Daley, and P. Morales. "Towards a distributed execution framework for JUnit test cases". In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. Sept. 2009, pp. 425–428. DOI: 10.1109/ICSM.2009.5306292.

[PTS+11] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. "Faults in Linux: Ten Years Later". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 305–318. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950401. URL: http://doi.acm.org/10.1145/1950365.1950401.

[PWM+12] T. Piper, S. Winter, P. Manns, and N. Suri. "Instrumenting AUTOSAR for dependability assessment: A guidance framework". In: *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2012, pp. 1–12.

[Ray04] E. Raymond, ed. *Jargon File: nasal demons*. 2004. URL: http://catb.org/jargon/html/N/nasal-demons.html (visited on 02/12/2015).

[ROT89] D. Richardson, O. O'Malley, and C. Tittle. "Approaches to Specification-based Testing". In: *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*. TAV3. Key West, Florida, USA: ACM, 1989, pp. 86–96. ISBN: 0-89791-342-6.

[RSF+99] M. Rodríguez, F. Salles, J.-C. Fabre, and J. Arlat. "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid". English. In: *Dependable Computing — EDCC-3*. Ed. by J. Hlavička, E. Maehle, and A. Pataricza. Vol. 1667. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 143–160. ISBN: 978-3-540-66483-3.

[RWH08] A. Rajan, M. Whalen, and M. Heimdahl. "The effect of program and model structure on mc/dc test adequacy coverage". In: *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*. May 2008, pp. 161–170.

[Ryd79] B. Ryder. "Constructing the Call Graph of a Program". In: *Software Engineering, IEEE Transactions on* SE-5.3 (May 1979), pp. 216–226. ISSN: 0098-5589.

[SAM08] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. "Sufficient Mutation Operators for Measuring Test Effectiveness". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 351–360. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368136. URL: http://doi.acm.org/10.1145/1368088.1368136.

[SBK10] D. Skarin, R. Barbosa, and J. Karlsson. "Comparing and Validating Measurements of Dependability Attributes". In: *Dependable Computing Conference (EDCC), 2010 European*. Apr. 2010, pp. 3–12. DOI: 10.1109/EDCC.2010.11.

[SBL03a] M. M. Swift, B. N. Bershad, and H. M. Levy. "Improving the Reliability of Commodity Operating Systems". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 207–222. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945466. URL: http://doi.acm.org/10.1145/945445.945466.

[SBL03b] M. M. Swift, B. N. Bershad, and H. M. Levy. "Improving the Reliability of Commodity Operating Systems". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 207–222. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945466. URL: http://doi.acm.org/10.1145/945445.945466.

[SBM+09] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors. "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures". In: *Dependable and Secure Computing, IEEE Transactions on* 6.2 (Apr. 2009), pp. 135–148. ISSN: 1545-5971. DOI: 10.1109/TDSC.2008.62.

[SC09] G. Somani and S. Chaudhary. "Application Performance Isolation in Virtualization". In: *Cloud Computing, 2009. CLOUD '09. IEEE International Conference on*. Sept. 2009, pp. 41–48. DOI: 10.1109/CLOUD.2009.78.

[SC91] M. Sullivan and R. Chillarege. "Software defects and their impact on system availability-a study of field failures in operating systems". In: *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*. June 1991, pp. 2–9.

[SFB+00] D. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. Iyer. "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors". In: *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International*. 2000, pp. 91–100. DOI: 10.1109/IPDS.2000.839467.

[SHK+12] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk. "Fail*: Towards a versatile fault-injection experiment framework". In: *ARCS Workshops (ARCS), 2012*. Feb. 2012, pp. 1–5.

[Sim03] D. Simpson. *Windows XP Embedded with Service Pack 1 Reliability*. http://msdn.microsoft.com/en-us/library/ms838661(WinEmbedded.5).aspx. 2003. (Visited on 03/09/2015).

[SJF+06] C. Sârbu, A. Johansson, F. Fraikin, and N. Suri. "Improving Robustness Testing of COTS OS Extensions". English. In: *Service Availability*. Ed. by D. Penkler, M. Reitenspiess, and F. Tam. Vol. 4328. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 120–139. ISBN: 978-3-540-68724-5.

[SJS+10] C. Sârbu, A. Johansson, N. Suri, and N. Nagappan. "Profiling the operational behavior of OS device drivers". English. In: *Empirical Software Engineering* 15.4 (2010), pp. 380–422. ISSN: 1382-3256.

[SKD00] C. Shelton, P. Koopman, and K. Devale. "Robustness testing of the Microsoft Win32 API". In: *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*. 2000, pp. 261–270. DOI: 10.1109/ICDSN.2000.857548.

[SKK+02] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. "Modeling the effect of technology trends on the soft error rate of combinational logic". In: *International Conference on Dependable Systems and Networks, 2002. DSN 2002. Proceedings*. 2002, pp. 389–398.

[Sof] C. Software. *SourceMonitor Version 2.5*. http://www.campwoodsw.com/sourcemonitor.html. (Visited on 03/09/2015).

[SP10] M. Staats and C. Pasareanu. "Parallel Symbolic Execution for Structural Test Generation". In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA '10. Trento, Italy: ACM, 2010, pp. 183–194. ISBN: 978-1-60558-823-0. DOI: 10.1145/1831708.1831732. URL: http://doi.acm.org/10.1145/1831708.1831732.

[SQP+11] N. Song, J. Qin, X. Pan, and Y. Deng. "Fault injection methodology and tools". In: *Electronics and Optoelectronics (ICEOE), 2011 International Conference on*. Vol. 1. July 2011, pp. V1-47–V1-50. DOI: 10.1109/ICEOE.2011.6013043.

[SRF+99] F. Salles, M. Rodriguez, J.-C. Fabre, and J. Arlat. "MetaKernels and fault containment wrappers". In: *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*. June 1999, pp. 22–29. DOI: 10.1109/FTCS.1999.781030.

[Sta00] E. Starkloff. "Designing a parallel, distributed test system". In: *AUTOTESTCON Proceedings, 2000 IEEE*. 2000, pp. 564–567. DOI: 10.1109/AUTEST.2000.885641.

[SVS+88] 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: *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*. June 1988, pp. 102–107. DOI: 10.1109/FTCS.1988.5306.

[SWN+09] C. Sarbu, S. Winter, N. Nagappan, and N. Suri. "OS Driver Test Effort Reduction via Operational Profiling". In: *Supplemental Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. 2009.

[TCF+07] L. Tan, E. Chan, R. Farivar, N. Mallick, J. Carlyle, F. David, and R. Campbell. "iKernel: Isolating Buggy and Malicious Device Drivers Using Hardware Virtualization Support". In: *Dependable, Autonomic and Secure Computing, 2007. DASC 2007. Third IEEE International Symposium on*. Sept. 2007, pp. 134–144.

[TDB13] P. Tsankov, M. T. Dashti, and D. Basin. "Semi-valid Input Coverage for Fuzz Testing". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA 2013. Lugano, Switzerland: ACM, 2013, pp. 56–66. ISBN: 978-1-4503-2159-4.

[THZ+99] T. Tsai, M.-C. Hsueh, H. Zhao, Z. Kalbarczyk, and R. Iyer. "Stress-based and path-based fault injection". In: *Computers, IEEE Transactions on* 48.11 (Nov. 1999), pp. 1183–1201. ISSN: 0018-9340. DOI: 10.1109/12.811108.

[TI95] T. K. Tsai and R. K. Iyer. "Measuring Fault Tolerance with the FTAPE fault injection tool". English. In: *Quantitative Evaluation of Computing and Communication Systems*. Ed. by H. Beilner and F. Bause. Vol. 977. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, pp. 26–40. ISBN: 978-3-540-60300-9.

[TIJ96] T. Tsai, R. Iyer, and D. Jewitt. "An approach towards benchmarking of fault-tolerant commercial systems". In: *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*. June 1996, pp. 314–323.

[Tip95] F. Tip. "A survey of program slicing techniques". In: *J. Prog. Lang.* 3.3 (1995).

[UBW72] E. G. Ulrich, T. Baker, and L. R. Williams. "Fault-test Analysis Techniques Based on Logic Simulation". In: *Proceedings of the 9th Design Automation Workshop*. DAC '72. New York, NY, USA: ACM, 1972, pp. 111–115.

[VCM+97] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. "Predicting how badly ldquo;good rdquo; software can behave". In: *Software, IEEE* 14.4 (July 1997), pp. 73–83. ISSN: 0740-7459. DOI: 10.1109/52.595959.

[VLM07] M. Vieira, N. Laranjeiro, and H. Madeira. "Assessing Robustness of Web-Services Infrastructures". In: *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*. June 2007, pp. 131–136. DOI: 10.1109/DSN.2007.16.

[VM03] M. Vieira and H. Madeira. "A Dependability Benchmark for OLTP Application Environments". In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, pp. 742–753. ISBN: 0-12-722442-4. URL: http://dl.acm.org/citation.cfm?id=1315451.1315515.

[VM95] J. Voas and K. Miller. "Predicting software's minimum-time-to-hazard and mean-time-to-hazard for rare input events". In: *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*. Oct. 1995, pp. 229–238.

[VM97] J. M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. New York, NY, USA: John Wiley & Sons, Inc., 1997. ISBN: 0-471-18381-4.

[Voa97] J. Voas. *A Defensive Approach to Certifying COTS Software*. 1997.

[Voa98] J. Voas. "Mitigating the Potential for Damage Caused by COTS and Third-Party Software Failures". In: *IN PROC. OF AQUIS'98*. Citeseer. 1998.

[WC80] L. White and E. Cohen. "A Domain Strategy for Computer Program Testing". In: *Software Engineering, IEEE Transactions on* SE-6.3 (May 1980), pp. 247–257. ISSN: 0098-5589.

[WCC+09] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. "Taming Coincidental Correctness: Coverage Refinement with Context Patterns to Improve Fault Localization". In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 45–55. ISBN: 978-1-4244-3453-4.

[Wey86] E. Weyuker. "Axiomatizing software test data adequacy". In: *Software Engineering, IEEE Transactions on* SE-12.12 (Dec. 1986), pp. 1128–1138. ISSN: 0098-5589.

[WGG+12] S. Winter, D. Germanus, H. Ghani, T. Piper, A. Khelil, and N. Suri. "Trustworthiness Evaluation Of Critical Information Infrastructures". In: *Critical Infrastructure Security: Assessment, Prevention, Detection, Response*. Ed. by F. Flammini. WIT Press, 2012.

[WGY+13] M. Whalen, G. Gay, D. You, M. P. E. Heimdahl, and M. Staats. "Observable Modified Condition/Decision Coverage". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 102–111. ISBN: 978-1-4673-3076-3.

[Whe] D. A. Wheeler. *SLOCCount*. http://www.dwheeler.com/sloccount/. (Visited on 03/09/2015).

[WNL+08] T. Wang, P. Narayanan, M. Leuchtenburg, and C. Moritz. ": A nanoscale fabric for nanoscale microprocessors". In: *Nanoelectronics Conference, 2008. INEC 2008. 2nd IEEE International*. Mar. 2008, pp. 989–994.

[WPS+15] S. Winter, T. Piper, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo. "GRINDER: On Reusability of Fault Injection Tools". In: *Proceedings of the IEEE/ACM International Workshop on Automation of Software Test (AST) (to appear)*. 2015.

[WRW+08] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. "Device Driver Safety Through a Reference Validation Mechanism". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 241–254.

[WSJ+09] S. Winter, C. Sarbu, A. Johansson, and N. Suri. "Impact of Error Models on OS Robustness Evaluations". In: *Supplemental Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. 2009.

[WSN+15] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo. "No PAIN, No Gain? The utility of PArallel fault INjections". In: *Proceedings of the International Conference on Software Engineering (ICSE) (to appear)*. 2015.

[WSS+11] S. Winter, C. Sârbu, N. Suri, and B. Murphy. "The Impact of Fault Models on Software Robustness Evaluations". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. 2011, pp. 51–60.

[WT04] J. Whittaker and H. Thompson. *How to Break Software Security: Effective Techniques for Security Testing*. Pearson/Addison Wesley, 2004. ISBN: 9780321194336.

[WTS+13] S. Winter, M. Tretter, B. Sattler, and N. Suri. "simFI: From single to simultaneous software fault injections". In: *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2013, pp. 1–12.

[XPZ+10] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. "Ad Hoc Synchronization Considered Harmful". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–8.

[YTC+10] L. Yu, W.-T. Tsai, X. Chen, L. Liu, Y. Zhao, L. Tang, and W. Zhao. "Testing as a Service over Cloud". In: *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*. June 2010, pp. 181–188. DOI: 10.1109/SOSE.2010.36.

[YZX+09] L. Yu, L. Zhang, H. Xiang, Y. Su, W. Zhao, and J. Zhu. "A Framework of Testing as a Service". In: *Management and Service Science, 2009. MASS '09. International Conference on*. Sept. 2009, pp. 1–4. DOI: 10.1109/ICMSS.2009.5302717.

[ZAV04] H. Ziade, R. Ayoubi, and R. Velazco. "A survey on fault injection techniques". In: *International Arab Journal of Information Technology* Vol. 1, No. 2, July (2004), pp. 171–186. URL: https://hal.archives-ouvertes.fr/hal-00105562.

[ZCA+06] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. "SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 45–60. ISBN: 1-931971-47-1.

[ZHM97] H. Zhu, P. A. V. Hall, and J. H. R. May. "Software Unit Test Coverage and Adequacy". In: *ACM Comput. Surv.* 29.4 (Dec. 1997), pp. 366–427. ISSN: 0360-0300.

[ZLX+12] Y. Zhu, Y. Li, J. Xue, T. Tan, J. Shi, Y. Shen, and C. Ma. "What Is System Hang and How to Handle It". In: *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. Nov. 2012, pp. 141–150. DOI: 10.1109/ISSRE.2012.12.

[ZXZ14] X. Zhou, L. W. and Xuandong Li, and J. Zhao. "An Empirical Study on the Test Adequacy Criterion Based on Coincidental Correctness Probability". In: *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013*. Ed. by M. Reformat. Knowledge Systems Institute Graduate School, 2014, pp. 632–635.