simFI: From Single to Simultaneous Software Fault Injections

Stefan Winter, Michael Tretter, Benjamin Sattler, Neeraj Suri
DEEDS Group, Dept. of CS, TU Darmstadt, Germany
{sw, tretter, bsattler, suri}@cs.tu-darmstadt.de

***Abstract*—Software-implemented fault injection (SWIFI) is an established experimental technique to evaluate the robustness of software systems. While a large number of SWIFI frameworks exist, virtually all 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 [1] and current safety standards require the consideration of "multi-point faults" [2], the validity of this single-fault assumption is at question for contemporary software systems.**

To address the issue and support simultaneous SWIFI (simFI), we analyze how independent faults can manifest in a generic software composition model and extend an existing SWIFI tool to support some characteristic simultaneous fault types. We implement three simultaneous fault models and demonstrate their utility in evaluating the robustness of the Windows CE kernel. Our findings indicate that simultaneous fault injections prove highly efficient in triggering robustness vulnerabilities.

Keywords*—Software fault injections, fault models, robustness testing

I. Introduction

Software-implemented fault injection (SWIFI) is used to evaluate the resilience of software to perturbations in its operational environment. However, the conclusiveness of such evaluations fundamentally depends on the proper choice of the applied *fault models* [3], [4] that specify how faults manifest in the system spanning fault dimensions of timing, duration, and location [5], [4].

Despite the large number of fault models behind the various proposed SWIFI frameworks (e.g., [6], [7], [4], [8]; see [9] for an 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 [10], no error interactions occur (i.e., all errors resulting from different fault activations are independent or have a necessary and sufficient causal relationship) 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, although software systems containing more than a single fault are reported to be a common occurrence [1]. However, if the consideration of multiple coincident faults significantly affects the outcome of a robustness evaluation, it would be a fallacy to disregard them.

The ISO standard 26262 for the functional safety of road vehicles explicitly considers *multiple-point failures*, i.e., failures resulting from the combination of several independent faults [2]. In order to assess the degree to which systems are vulnerable to such multiple-fault conditions, new approaches are needed and this forms the focus of this paper.

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

(C1) Coincidence notion: We establish fault coincidence notions and discuss their applicability within a general model of SWIFI-based robustness assessments for software compositions (Sections II and IV).

(C2) Impact criteria: We identify and discuss SWIFI experimentation aspects affected by simultaneous faults. As these aspects have direct implications on the efficiency of SWIFI experiments, they form a set of criteria for assessing both the overhead and the benefits of simultaneous fault injections (Section III).

(C3) Simultaneous fault model design: The utility of coincident injections is shown via the design and implementation of three simultaneous fault models (Section V) that are instances of the abstract fault coincidence notion developed as (C1).

(C4) Quantitative evaluation of simFI: We quantitatively compare the efficiency of (a) three simultaneous and (b) three discrete fault models (Section V).

II. System & Basic Fault Models

We consider robustness evaluations of component-based software systems, where components interact via contractually specified interfaces [11]. The implementation of each component's functionality is accessible through sets of *services* offered by the component, termed *interfaces*. Compositions implement their functionality using services of their individual components. The set of services that a component provides is termed the component's *export interface*. The corresponding *import interface* spans the external services it depends on.

We assume the availability of functional service specifications for components as the minimum information required for composition. We do not assume source code access or other component implementation details.

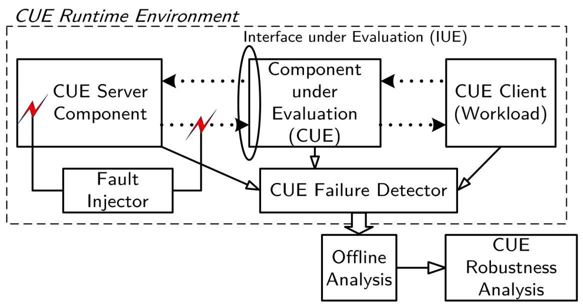

Fig. 1: System model and SWIFI experiment setup

SWIFI and Software Fault Models

Avižienis et al. [10] define service *failure* as the deviation of a service from its specification. The cause of a service failure is the impact of an unintended state (termed *error*) of the component providing that service. The cause of an error is the *activation* of a fault. Component-external events causing fault activation constitute *external faults*, whereas faults activated inside the component are called *internal faults*.

From the *robustness* notion in [12], our work considers the problem of detecting internal faults (*robustness vulnerabilities*) by exposing software components to external faults (*perturbations*). If an external fault is injected and a component failure is observed, then an internal fault exists, which has been triggered by the injection of the external fault. This detection mechanism is sound but incomplete, i.e., it never raises false alarms but existing internal faults may remain undetected.

Figure 1 shows our general system model for these experimental assessments. In a composition, the presence of internal robustness vulnerabilities in a *component under evaluation* (CUE) is assessed. The CUE is being used by one or more other components, which we term *CUE clients*, via its export interfaces. To provide services to these client components, the CUE depends on services from other components, termed *CUE servers*, whose services it imports. We inject external faults to the CUE by either mutating its servers or by corrupting data that the servers send to the CUE via both its exported and imported interfaces. The intended effect is that the servers' behavior differs from their specified behavior expected by the CUE's implementation. As for a given experiment setup only a fraction of all CUE interfaces may be used by the chosen server components, the used interface between the CUE and its servers is termed the *interface under evaluation* (IUE). When components interact only via contractually specified interfaces, component failures can only affect other components via these interfaces. Injections at the interface level are a sound method to accelerate fault injection experiments in these cases, as all erroneous behavior that can affect any other components must be observable at the faulty component's interfaces. We utilize failure detectors in the CUE's run-time environment to observe the CUE's response to injections. This data is evaluated offline for the identification of internal faults.

Despite choosing a specific system model and SWIFI setup, the number of possible external faults for an individually targeted service is extremely large. Faults are therefore categorized into fault models and in accordance with other SWIFI-related literature [5], [4], [13] we model faults according to their *location* (*where* to inject), *type* (*what* to inject), *trigger* (*when* to inject) and *latency* (for *how long* to inject).

Individual experiments are termed *injection runs*. Injection runs with the same system configuration (CUE clients, servers, IUEs) and the same fault model, but different fault model instances being injected, are called *injection campaigns*. Multiple injection campaigns for the same CUE, but with different fault models and different system configurations constitute an *evaluation* of the CUE.

III. Fault Simultaneity Aspects in SWIFI-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 simultaneous faults in SWIFI-based assessments may have a number of conceptual and technical implications on experimental assessments, which we discuss in the following.

A. 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 [14]: Fault interactions are so rare that they can be neglected during tests without significantly affecting the outcome in the common case.

However, in safety engineering, critical failures (*accidents*) are often considered to result from unlikely, rare, and unconsidered combinations of events, which is also reflected by the ISO 26262 safety standard [2], [15]. 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 [16], [17].

B. Experiment Acceleration vs. Counts

Single fault injections are capable of identifying one internal fault per injection run. If a simultaneous 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 simultaneous fault injection run may also lead to more ambiguous results.

If multiple faults are injected, one activated fault (termed *error* [10]) 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, 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 it would have caused the component to fail.

If the injection of simultaneous faults results in a system failure, a similar ambiguity can occur: It is unclear whether the activation of *all* injected faults is actually necessary to trigger the observed failure. This makes it difficult to spot the robustness vulnerability or vulnerabilities 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 single faults constituting the simultaneous fault condition. However, this may add considerably to the number of overall required experiments.

Even without the potential need for additional experiments to resolve ambiguities, the consideration of coincident 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 k coincident faults, assuming all possible combinations as feasible.

We experimentally investigate the impact of coincident faults on the evaluation efficiency in Section V.

C. Sequential vs. Concurrent Executions

Unlike classical sequential processing, modern processors provide higher performance via higher degrees of parallelism instead of higher execution rates. In order to exploit these performance gains, software applications are designed for the parallel execution of independent threads. This parallel execution of different instructions belonging to the same software component enables the simultaneous activation of independent internal faults within one component. With an increasing degree of parallelism, the rate of failures resulting from simultaneous fault activations naturally increases.

We develop notions of "temporal" and "spatial" fault coincidence that apply for both sequential and concurrent execution models in the following section.

IV. Modeling Simultaneous Faults

Attempting to define simultaneous fault models strictly on the fault notion and system model in Section II is problematic. The fault notion of Avižienis et al. [10] implies that a software component cannot be exposed to two independent external faults. As an external fault is defined as something that triggers an internal fault and an internal fault is defined as something possibly resulting in a component failure, there are two possibilities: Either the simultaneous injection of two independent alterations does not lead to a failure and, consequently, does not constitute a fault, or they lead to an observable component failure, in which case they are considered *one* external fault.

However, if we assume software components to be themselves composed of components, simultaneous faults can be defined via subcomponent faults. Consider component A to be composed of components B and C as shown in Figure 2. Assume B to have the internal fault f_B and C to have the internal fault f_C. If there exist external faults $\overline{f_B}$, $\overline{f_C}$, $\overline{f_{BC}}$, so that $\overline{f_B}$ triggers f_B, $\overline{f_C}$ triggers f_C, $\overline{f_{BC}}$ triggers both f_B and f_C, then we call $\overline{f_{BC}}$ a simultaneous fault if it is an external fault to A, i.e., if it results in a failure of A.

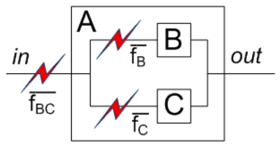

Fig. 2: Intuition for external fault simultaneity

The external faults $\overline{f_B}$, $\overline{f_C}$ are distinguishable, as they trigger different internal faults that constitute external faults to distinct subcomponents. While temporally coincident, they differ in location. There may also be different external faults triggering the same internal fault. These are perceived as distinct cases if they occur at different instances of time. Hence, we distinguish between temporally coincident faults (that differ in location) and spatially coincident faults (that differ in timing). While temporal fault coincidence is the more intuitive notion and less considered in existing work, we first discuss "spatially coincident but temporally spread" faults, as they facilitate the accurate definition of temporal fault coincidence.

A. Spatial Coincidence and Temporal Resolution

Definition 1: An external fault is termed *spatially coincident but temporally spread* if it triggers the same internal fault at different instances of time.

The precise timing and even an approximate timing with respect to real execution times can be difficult to measure if the component's implementation is not accessible for instrumentation. Therefore, we relax the notion for black-box components and only require observability at their interfaces during different *usage sequences*.

Definition 2: An external fault of a black-box component is termed *spatially coincident but temporally spread* if it triggers the same internal fault within more than one temporal usage sequence of the functional interface. *Temporal usage sequence* may refer to an evaluation, an injection campaign, a single injection run, or an individual service invocation during an injection run. The choice of the usage sequence is referred to as *temporal resolution*.

For simplicity we write "temporally spread" to mean "spatially coincident but temporally spread". 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. It is important to note that the external fault is only considered temporally spread if it targets the same internal fault in multiple evaluations. 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.

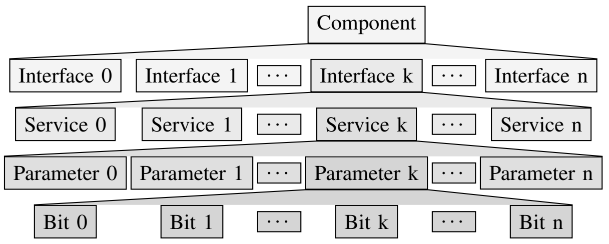

Fig. 3: Relationship of spatial entities

Temporally spread fault injections are more widely applied than temporally coincident fault injections, e.g., for the the injection of intermittent or permanent faults. They are also applied to investigate fault effect accumulation, often referred to as "software aging". While we do not develop temporally spread fault models in this paper, we discuss existing work addressing temporally spread injections in Section VI.

B. Temporal Coincidence and Spatial Resolution

Definition 3: An external fault of a component is termed *temporally coincident* if it triggers internal faults in more than one sub-component.

As two internal faults are triggered by one event occurring at one instance of time, their *causation* is considered temporally coincident. While this definition works well for cases where the internal structure of a component under evaluation is known, it cannot be directly applied to black-box components for which the internal structure is not known to the evaluator. We, therefore, approximate the internal structure of black-box components by reference to their accessible interfaces.

Definition 4: An external fault of a black-box component is termed *temporally coincident* if it triggers internal faults through different spatial entities in the component'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 3 depicts the relationship between the considered spatial entities. An example for a temporally coincident fault of a black-box component 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.

Note that the redefinition of temporal coincidence for black-box components requires the previously introduced notion of temporal usage sequences. We recognize that *the temporal and spatial dimensions of fault models are not independent* for black-box components. 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 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. We develop fault models for these two cases in the following section.

V. EXPERIMENTAL EVALUATION

We define three simultaneous fault models according to the temporal coincidence notion developed in Section IV (C3) and investigate the impact of these models on the evaluation efficiency, taking masking and amplification effects into account (C4). We conclude that the evaluation effort resulting from the higher numbers of experiments for simultaneous fault injections is well justified in this case study (C2). Before going into the details of the developed models and their performance, we briefly describe the experimental setup for our case study.

A. Experimental Setup

We apply SWIFI to the Windows CE 4.2[1] operating system (OS), specifically targeting device drivers. Device drivers are known to be error prone and constitute a common cause for OS failures [18], [19], [20], [21]. We therefore choose to investigate the effects that device driver failures have on the overall system by injecting faults at the OS's driver interface and observing the results at its application interface. The approach is similar to previous studies [22], [6] and aims at the identification of vulnerable OS services, so that runtime fault-tolerance mechanisms (e.g., [23]) can be selectively applied if required.

Windows CE loads device drivers as dynamically linked library (DLL) binaries. We use a serial port driver and an Ethernet driver for our case study. Both drivers export a Windows stream driver interface [24]. The serial port driver imports the CEDDK [25] and COREDLL [26] kernel interfaces. The Ethernet driver imports COREDLL and NDIS [27].

We inject external faults 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 classically known outcomes of an injection run [28]. 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.

B. Fault Models: From Discrete to Simultaneous

Our injection framework supports a large number of fault models with a variety of injection triggers and latencies. While it supports various injection locations, i.e., injections

[1]The choice of Windows CE is deliberate for (a) its structure being representative for a large class of component-based software systems and (b) its wide research usage providing an extensive published basis to objectively compare results for a variety of single-fault models.

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 [3], [29]. 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): A passed parameter value is replaced by a data type-specific boundary value, e.g., MAXINT, MININT, -1, 0, 1 for the integer data type.

We use BF and FZ models as a basis for constructing three temporally simultaneous fault models (C3), i.e., models of faults that are activated within the same temporal usage sequence in different spatial locations. We choose individual service invocations as the considered temporal usage sequence. Simultaneous faults can then manifest themselves as faults in distinct parameters of the same service call or as distinct faults within one single parameter of the targeted service call, depending on the spatial resolution.

1) The Simultaneous FuzzFuzz Model: While the fuzzing 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. Note that fuzzing is only usefully applicable for spatial resolutions lower than the individual parameter level: If fuzzing was simultaneously 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). Consider a service invocation `foo(a,b)` to a component under evaluation. Fuzzing mandates to replace either parameter value `a` or `b` by a random value. If fuzzing was applied to *the same parameter* twice, the result would be indistinguishable from a single parameter fuzzing. We therefore consider simultaneously fuzzing *different* parameters with the FuzzFuzz model, thereby implicitly restricting its applicability to services that are taking at least two parameters as inputs.

For the example `foo(a,b)`, $2^{bitlength(a)} + 2^{bitlength(b)}$ possible FZ injections and $2^{bitlength(a)+bitlength(b)}$ possible FuzzFuzz injections exist. Even for the FZ model an exhaustive evaluation 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 et al. [4], who report error propagation measures to stabilize for the considered CUE after approximately 10 injections.

2) The Simultaneous FuzzBF Model: Johansson et al. report the fuzzing and bit flip models to perform well in robustness evaluations of the considered CUE [4]. We combine these models for injection experiments in a simultaneous model: 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 invocation `foo(a,b)` there exist $2^{bitlength(a)} + 2^{bitlength(b)}$ possible test cases for individual fuzzing injections and $bitlength(a) + bitlength(b)$ possible individual bit flips. For the FuzzBF model the amount of possible test cases expands to $(2^{bitlength(a)} - 1) * bitlength(b) + (2^{bitlength(b)} - 1) * bitlength(a)$. This considers that all single bit flip cases for a parameter are contained in the number of possible Fuzzing values.

3) The Simultaneous SimBF Model: As opposed to the FuzzFuzz and FuzzBF models, the SimBF model has a spatial resolution of individual data elements of which a parameter value in a service invocation is composed: We are flipping two bits in the binary value of the same parameter.

For a service invocation `foo(a,b)` there exist $bitlength(a) + bitlength(b)$ possible single bit flips and $\binom{bitlength(a)}{2} + \binom{bitlength(b)}{2}$ possible SimBF test cases. This may still result in large (polynomial) test case numbers, but exhaustive testing is feasible. Although multi-bit fault models have been applied in previous work before (e.g., [30]), they have not been discussed as simultaneous single-bit faults and their efficiency has, therefore, not been assessed comparatively.

From the differing applicability of the fuzzing and bit flip models to different spatial resolutions, we observe that *the fault type and the fault location are not independent from each other*. These dimensions have often been considered orthogonal in the literature. 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.

C. Evaluation Criteria

To evaluate the efficiency of the simultaneous fault models, we utilize four previously proposed metrics [29] and add a discussion of simultaneous fault interactions in terms of masking and amplification compared to single faults.

Coverage and unique coverage: We define the *coverage* of a fault model as the fraction of services in the interface under evaluation that the fault model identifies as vulnerable or faulty. In order to distinguish between distinct sets of identified services that comprise equal numbers of services, *unique coverage* is defined as the fraction of services that no other model identifies as vulnerable or faulty among the set of compared models.

Injection efficiency: The fraction of experiments that resulted in a failure of a specific class (AE, AH, SC).

Average execution time: The amount of time required for one injection run.

Implementation complexity: The effort required to implement the model measured as delivered source instructions (DSI; the number of physical source lines of code excluding comments and test code) and the accumulated cyclomatic complexity [31] of all code specific to the model.

In order to analyze masking and amplification effects of simultaneously injected faults relative to single fault injections, we investigate where the combination of two individual faults leads to an outcome that differs from the outcomes for the individual injections (C4).

Masking and amplification: To measure the amount of masking and amplification due to simultaneous injections, we investigate the failure class distributions for each spatial entity, for which simFI experiments are conducted. We compare the obtained distributions against the failure class distributions of experiments with the single faults, from which the simultaneous faults have been composed. If fewer failures are observed for the simultaneous cases than for any of the corresponding single faults, masking has taken place. If we observe higher failure rates in the simultaneous cases, amplification has taken place.

In order to obtain reliable results for the masking and amplification analyses of models that involve fuzzing, we additionally perform single injection experiments for each random value used in a simultaneous fuzzing experiment and compare the outcomes. This is necessary, as the application of fuzzing may result in different random values for the single and simultaneous injections, which would lead to incomparable results and, hence, render the proposed approximations of masking and amplification invalid.

D. Experimental Results: Exploring Simultaneous Injections

We compare the effects of all fault models targeting interactions of a serial port driver and the OS kernel. We have performed a number of additional experiments using an Ethernet driver, evaluating all fault models except for the SimBF model. As previously mentioned, we performed exhaustive testing with this model. However, this resulted in more than 34000 individual injection runs taking up two months of experiment run-time for only this model and for only the serial driver. Thus, we have restricted our experiments with this model to the serial port driver. Despite this restriction, we are only able to discuss a fraction of our extensive experimental results to illustrate our findings due to space limitations. Supplemental experimental results are made available on our web page [32].

1) Coverage and unique coverage: Figures 4 to 6 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. As return values are considered as a single parameter, FuzzFuzz and FuzzBF cannot be applied for injections in the export interface.

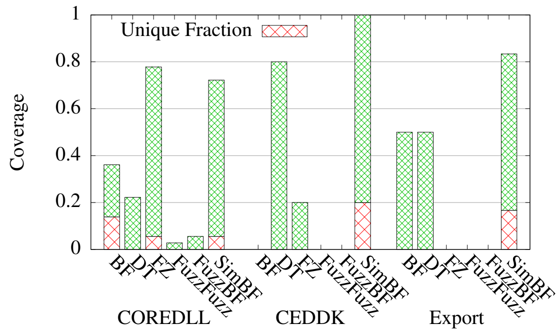

Fig. 4: (Unique) coverage of AE vulnerabilities using the serial driver

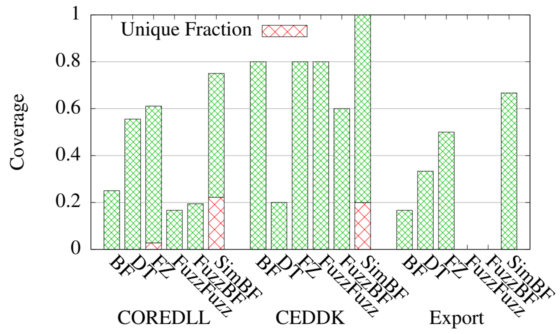

Fig. 5: (Unique) coverage of AH vulnerabilities using the serial driver

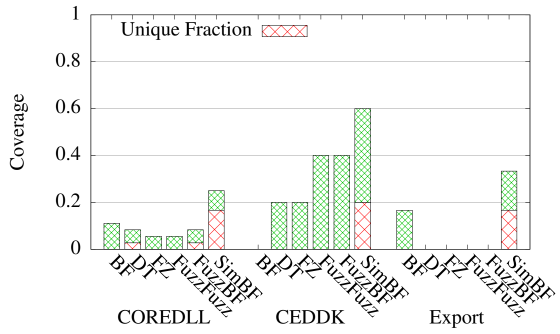

Fig. 6: (Unique) coverage of SC vulnerabilities using the serial driver

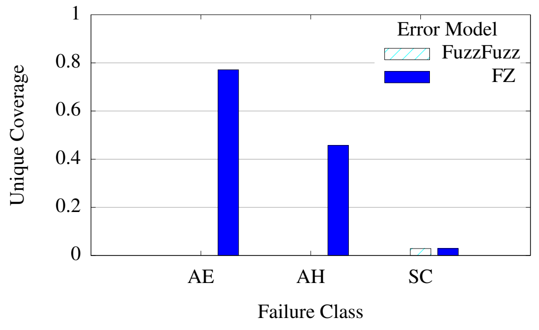

Fig. 7: Unique coverages of FuzzFuzz & FZ evaluating the COREDLL interface using the serial driver

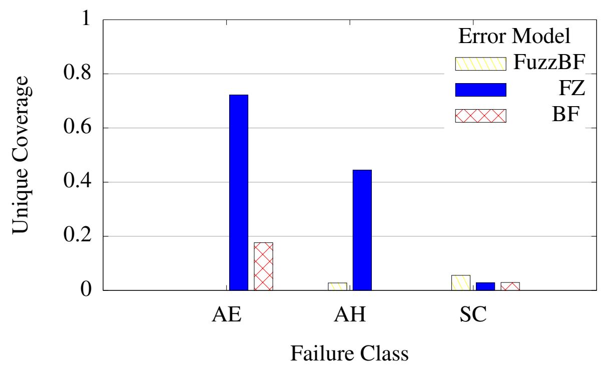

Fig. 8: Unique coverages of FuzzBF, FZ & BF evaluating the COREDLL interface using the serial driver

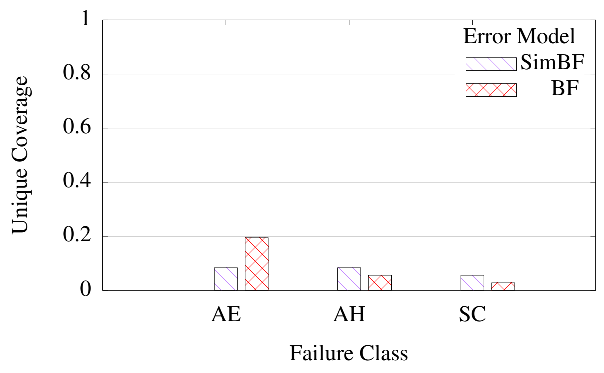

Fig. 9: Unique coverages of SimBF & BF evaluating the COREDLL interface using the serial driver

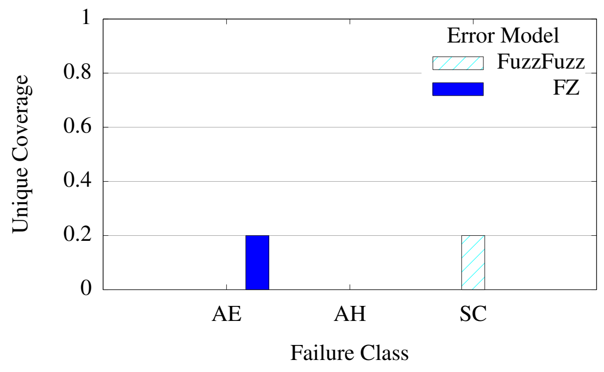

Fig. 10: Unique coverages of FuzzFuzz & FZ evaluating the CEDDK interface using the serial driver

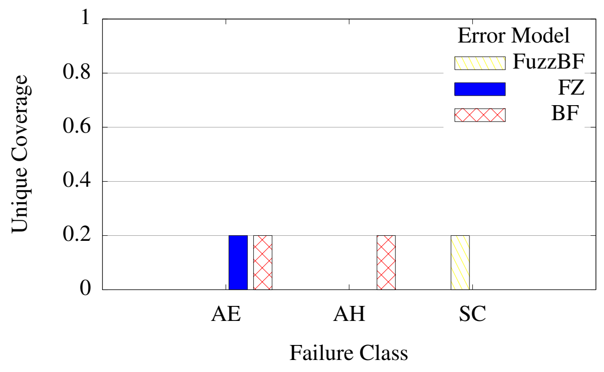

Fig. 11: Unique coverages of FuzzBF, FZ & BF evaluating the CEDDK interface using the serial driver

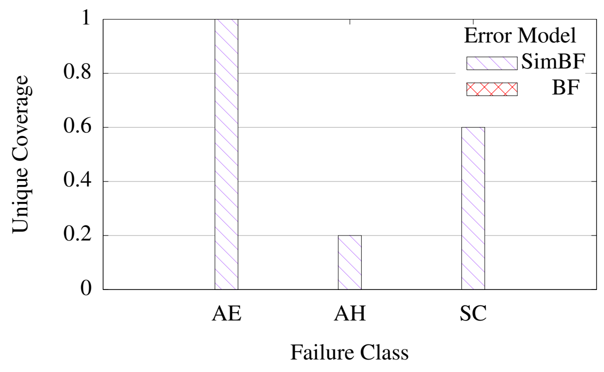

Fig. 12: Unique coverages of SimBF & BF evaluating the CEDDK interface using the serial driver

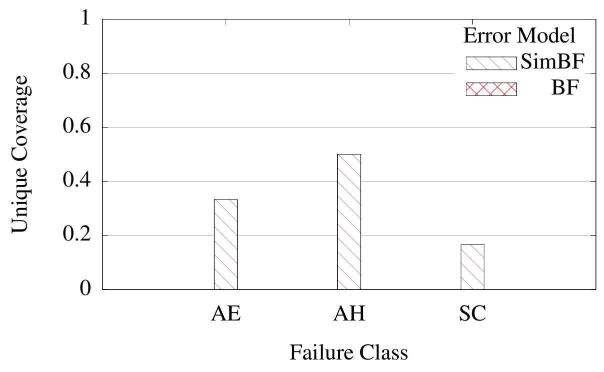

Fig. 13: Unique coverages of SimBF & BF evaluating the export interface of the serial driver

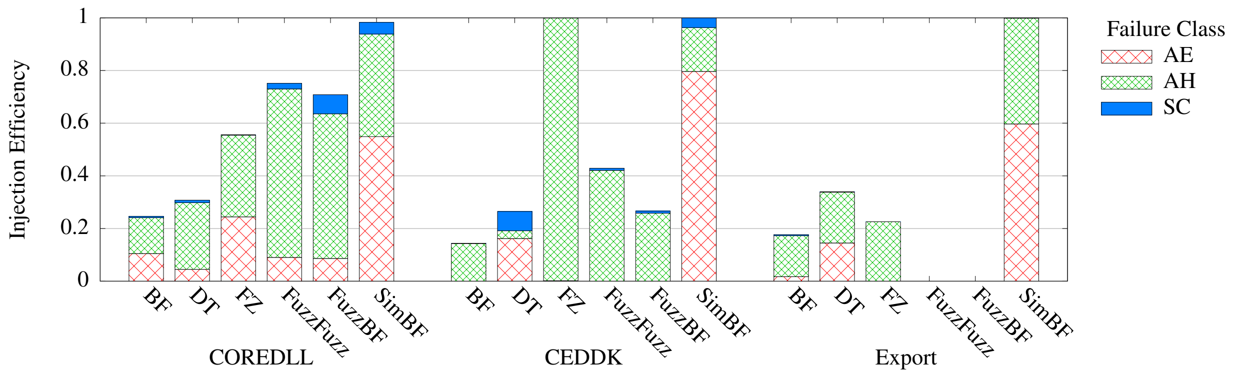

Fig. 14: Injection efficiencies of experiments using the serial driver

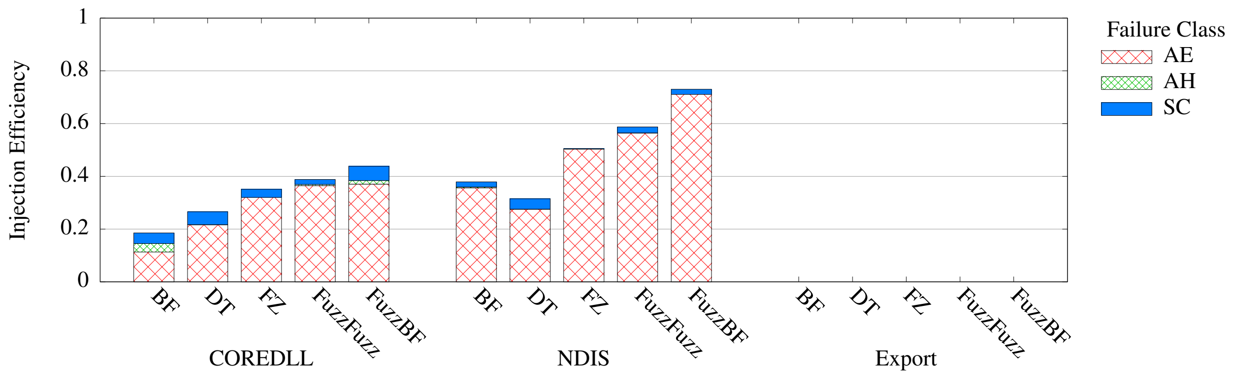

Fig. 15: Injection efficiencies of experiments using the Ethernet driver

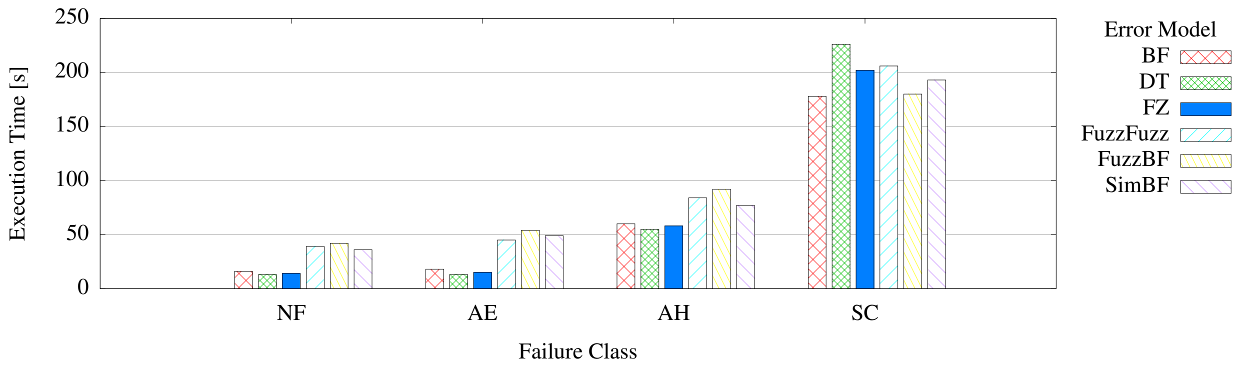

Fig. 16: Average experiment execution times using the serial driver

SimBF in contrast outperforms all other models in most of the cases, showing coverages of up to 100% for the CEDDK interface in Figures 4 and 5, i.e., it identifies a robustness vulnerability 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 7 to 9. For the COREDLL interface we see substantially higher unique coverages for vulnerabilities that result in AE or AH failures (Figures 7 and 8), when the discrete FZ model is applied. However, Figures 7 to 9 also show that the simultaneous models cannot be neglected when testing for defects that lead to system crashes (SC), as all three simultaneous 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 10 to 13).

2) Injection efficiency: Figures 14 and 15 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 discrete 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 simultaneous models. Given these expected benefits, we investigate the costs for these additional experiment runs and the simultaneous fault model implementations.

3) Average execution time: Figure 16 shows the average execution time for injection runs using the serial driver grouped by failure classes. The simultaneous models always take the longest execution times. As the actual injection overhead for our implementations of the simultaneous models is negligible, we assume that the observed delays 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 disorder 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.

4) Implementation complexity: The implementation complexities in terms of delivered source instructions and accumulated cyclomatic complexity are presented in Table I. Being the least complex simultaneous model among the assessed models, SimBF also is the second-least complex model overall and only outperformed by the BF model. The DSI 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 simultaneous injections into different parameters of the same service invocation accounts for only 105 DSIs and an accumulated cyclomatic complexity of 18.

TABLE I: Implementation complexities

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

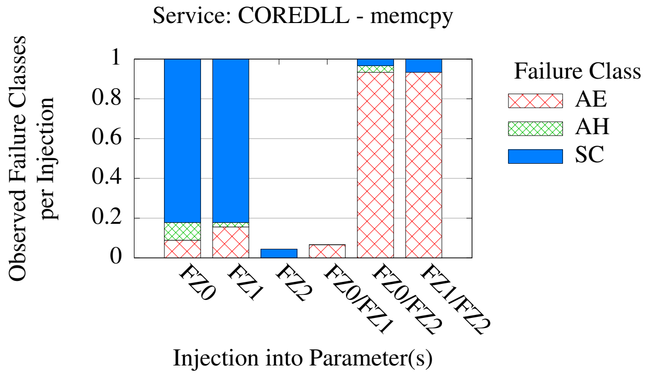

Fig. 17: Example for masking effects of the FuzzFuzz model

We conclude that simultaneous fault injections entail moderate overhead for implementation and execution time.

5) Masking and amplification: We have found masking and amplification effects to depend highly 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 due to space limitations we restrict ourselves to only discussing a few illustrative examples. Detailed plots for all tested services are available on our web site [32].

Figure 17 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 18 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 simultaneous injections into parameters 0 and 2 result in higher SC failure rates. To avoid sampling effects in the comparisons, we have backed up our

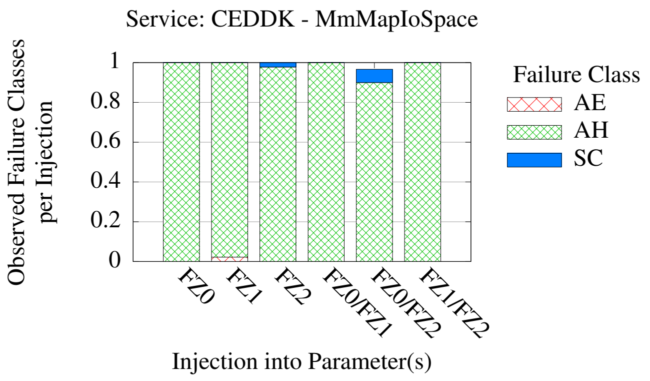

Fig. 18: Example for amplification effects of the FuzzFuzz model

results by additional experiments, as discussed in Section V-C.

E. Discussion

The simultaneous models add value to the performed evaluation, as they detect service faults that are not covered by other models. On efficiency considerations, the SimBF model outperforms discrete fault models in most cases. The implementation overhead of all considered simultaneous models is moderate compared to the discrete models. Especially for critical applications the consideration of coincident faults threatening dependability may well be worth the effort.

Caveats: As we have discussed previously, exhaustive tests are feasible for the BF and SimBF models, while they are not for the FZ-based models. Consequently, we were able to exhaustively inject BF and SimBF faults and had to select a fixed number of test cases for the FZ-based models. Interestingly, the BF and SimBF faults that we injected are also among the injection candidates of the FZ model. However, we see significantly different results for the (simultaneous) bit flips, indicating that there are significant 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 paper and left as subject to future research.

We observed differing performance of the different fault models for different drivers. Furthermore some interfaces are only used by some drivers and not by others. We therefore assume that the efficiency of an evaluation depends on an interplay of fault models and the precise system configuration, comprising fault loads and applied workloads.

VI. Related Work

While basic simultaneous fault injections have previously been conducted, they have not been detailed for either spatial or temporal resolutions as in our work. We discuss related approaches to put our contributions in context.

A. Simultaneous Injections into Function Call Parameters

In the Ballista project, simultaneous injections into multiple parameters of a function call have been performed with a data-type fault model [33]. The authors do not differentiate between valid and invalid parameters that are combined it the tests and do not quantitatively analyze the impact of simultaneous injections, i.e., the combination of invalid inputs. However, they report that simultaneous injections have not had a significant impact on the detection probability of robustness vulnerabilities [34], [28], which contradicts our results. We see three possible reasons for this deviation, which we are planning to investigate in more depth in the future: (1) Ballista and our work target different interfaces of the OS (driver interface vs. API), (2) the quoted Ballista results were obtained more than a decade ago and may not hold for newer systems, and (3) the results may significantly differ due to the different fault models. We plan to investigate these possible causes in future work.

Although not discussed explicitly, automated fault injections applied for the determination of the robust argument type for n-ary functions in HEALERS [35] probably also use simultaneous injections into different parameters: If the robust types of multiple arguments depend on each other, this dependency can only be reflected by simultaneous injections. If there is no such dependency, simultaneous tests for robust types of independent arguments can reduce the test effort.

B. Validating software-implemented hardware fault-tolerance

Software replication (e.g., N-copy programming [36], or process-level redundancy [37]) is a common technique to handle the effects of non-deterministic hardware errors on software. 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 et al. [38] 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 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. 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 [39]. The work in this paper differs from these approaches, as we do not restrict ourselves to the emulation of hardware-induced software errors. We do not make any assumption about a fault, apart from constituting an external fault [10] to the component under evaluation.

C. Higher order mutation testing

Mutation testing [40] 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 introduced higher order mutants [41] resulting from consecutive applications of multiple traditional first-order mutation operators, thereby possibly injecting both temporally coincident and temporally spread faults according to our notions. They classify higher order mutants according to their impact in test suite evaluations, as opposed to our classification according to where and at what granularity injections are applied.

While mutation order and interface fault coincidence have similar intent, our proposed coincidence notion is more general, as we provide definitions for both white-box and black-box assessment scenarios. We also consider coincidence at different spatial and temporal resolutions instead of restricting ourselves to a specific spatial or temporal scope, such as individual programs.

D. 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., [8], [42]) tries to avoid accumulating effects for methodological reasons, as these complicate the identification of effects that an individual fault activation has on the system (cf. Section III-B). The second class of work (e.g., [43], [44]) considers accumulating fault effects and either accepts that no direct causal relation between an individual fault activation can be established or performs additional experiments to establish these relations. Our work falls in the latter category. While existing work has partially considered temporal spread in terms of software aging, our work also explores temporal coincidence in both our definitions and experiments.

Moreover, our definitions for temporally spread faults provide a general template for defining fault models to emulate software aging effects, which can be used to assess and optimize the effectivness of software rejuvenation strategies, as mandated by experimental analyses of complex systems code [45], [46].

VII. Conclusion

Surprisingly, the interaction of coincident fault effects is often neglected in software robustness analyses. In this paper we have investigated the benefits and costs that come with a consideration of coincident faults in terms of evaluation efficiency. Our paper makes the following contributions.

(C1) establishes coincidence notions for SWIFI covering both the temporal and spatial domains.

(C2) investigates conceptual and technical arguments that affect the effectiveness of coincident fault injections in software evaluations.

(C3) details the design and implementation of three simultaneous, i.e., temporally coincident, fault models for SWIFI experimentation.

(C4) assesses the simultaneous models' performance quantitatively in an actual robustness evaluation.

The proposed coincidence notions enable a large number of novel fault models for SWIFI experiments and although we have limited ourselves to the design and assessment of only three models, our experimental results indicate that *coincident fault models have significant effects on the coverage in a software robustness evaluation*. Depending on the temporal and spatial granularity, 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.

Our 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 [14] 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.

From our work we, hence, conclude both the need and the practicability of simultaneous fault injections for software robustness evaluations.

VIII. Acknowledgements

The authors would like to thank Thorsten Piper and Holger Rother for comments on a previous draft of the paper. The work in this paper has been supported by CASED (http://www.cased.de).

References

[1] G. Candea, M. Delgado, M. Chen, and A. Fox, "Automatic Failure-Path Inference: A Generic Introspection Technique for Internet Applications," in *Proc. WIAPP 2003*, 2003, pp. 132 – 141.

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

[3] 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 *Proc. EDCC*, 2006, pp. 53–64.

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

[5] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," in *Proc. FTCS*, 1996, pp. 304–313.

[6] J.-C. Fabre, M. Rodriguez, J. Arlat, and J.-M. Sizun, "Building dependable COTS microkernel-based systems using MAFALDA ," in *Proc. PRDC*, 2000, pp. 85 –92.

[7] 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 *Proc. IPDS*, 2000, pp. 91 –100.

[8] P. Costa, M. Vieira, H. Madeira, and J. Silva, "Plug and Play Fault Injector for Dependability Benchmarking," in *Dependable Computing*, ser. LNCS. Springer, 2003, vol. 2847, pp. 8–22.

[9] R. Natella, "Achieving Representative Faultloads in Software Fault Injection," Ph.D. dissertation, Università di Napoli Federico II, 2011.

[10] A. Avižienis, J. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, 2004.

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

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

[13] P. D. Marinescu and G. Candea, “Efficient Testing of Recovery Code Using Fault Injection,” *ACM Trans. Comput. Syst.*, vol. 29, no. 4, pp. 11:1–11:38, Dec. 2011.

[14] D. Kuhn, D. Wallace, and J. Gallo, A.M., “Software fault interactions and implications for software testing,” *Software Engineering, IEEE Transactions on*, vol. 30, no. 6, pp. 418–421, 2004.

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

[16] H. Hecht, “Rare Conditions – An Important Cause of Failures,” in *Proc. COMPASS*, 1993, pp. 81–85.

[17] R. R. Lutz and I. C. Mikulski, “Operational Anomalies as a Cause of Safety-Critical Requirements Evolution,” *JSS*, vol. 65, no. 2, pp. 155 – 161, 2003.

[18] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An Empirical Study of Operating Systems Errors,” in *Proc. SOSP*, 2001, pp. 73–88.

[19] D. Simpson, “Windows XP Embedded with Service Pack 1 Reliability,” http://msdn.microsoft.com/en-us/library/ms838661(WinEmbedded.5).aspx.

[20] A. Ganapathi, V. Ganapathi, and D. Patterson, “Windows XP Kernel Crash Analysis,” in *Proc. LISA*, 2006, pp. 12–22.

[21] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, “Faults in linux: ten years later,” in *Proc. ASPLOS*, 2011, pp. 305–318.

[22] A. Albinet, J. Arlat, and J.-C. Fabre, “Characterization of the impact of faulty drivers on the robustness of the Linux kernel,” in *Proc. DSN*, 2004, pp. 867 – 876.

[23] M. M. Swift, B. N. Bershad, and H. M. Levy, “Improving the reliability of commodity operating systems,” in *Proc. SOSP*. ACM, 2003, pp. 207–222.

[24] MSDN, “Stream Interface Driver Implementation (Windows CE .NET 4.2),” http://msdn.microsoft.com/en-us/library/ms895309.aspx.

[25] ——, “Implementing CEDDK.dll,” http://msdn.microsoft.com/en-us/library/ms898217.aspx.

[26] ——, “coredll Module,” http://msdn.microsoft.com/en-us/library/aa448387.aspx.

[27] ——, “Network Driver Functions,” http://msdn.microsoft.com/en-us/library/ms895631.aspx.

[28] P. Koopman, K. Devale, and J. Devale, “Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project,” in *Dependability Benchmarking for Computer Systems*, K. Kanoun and L. Spainhower, Eds., 2008, pp. 201–226.

[29] S. Winter, C. Sârbu, N. Suri, and B. Murphy, “The impact of fault models on software robustness evaluations,” in *Proc. ICSE*, 2011, pp. 51–60.

[30] T. Liu, Z. Kalbarcyzk, and R. K. Iyer, “Software-Embedded Multilevel Faul Injection Mechanism for Evaluation of a High-Speed Network System under Windows NT,” in *Proc. IOLTW*, 1999.

[31] T. McCabe, “A Complexity Measure,” *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976.

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

[33] N. Kropp, P. Koopman, and D. Siewiorek, “Automated robustness testing of off-the-shelf software components,” in *Proc. FTCS*, 1998, pp. 230 –239.

[34] J. Pan, “The Dimensionality of Failures – A Fault Model for Characterizing Software Robustness,” in *Proc. FTCS*, 1999.

[35] C. Fetzer and Z. Xiao, “An automated approach to increasing the robustness of C libraries,” in *Proc. DSN*, 2002, pp. 155 – 164.

[36] P. E. Ammann and J. C. Knight, “Data diversity: an approach to software fault tolerance,” *IEEE Trans. Comput.*, vol. 37, no. 4, pp. 418–425, 1988.

[37] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, “PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures,” *IEEE Trans. Dependable Secure Comput.*, vol. 6, no. 2, pp. 135–148, 2009.

[38] P. Joshi, H. S. Gunawi, and K. Sen, “PREFAIL: a programmable tool for multiple-failure injection,” in *Proc. OOPSLA*, 2011, pp. 171–188.

[39] M. de Kruijf and K. Sankaralingam, “Exploring the synergy of emerging workloads and silicon reliability trends,” in *SELSE*, 2009.

[40] Y. Jia and M. Harman, “An Analysis and Survey of the Development of Mutation Testing,” *IEEE TSE*, vol. 37, no. 5, pp. 649 –678, 2011.

[41] ——, “Higher Order Mutation Testing,” *Information and Software Technology*, vol. 51, no. 10, pp. 1379 – 1393, 2009.

[42] M. Vieira, N. Laranjeiro, and H. Madeira, “Assessing Robustness of Web-Services Infrastructures,” in *Proc. DSN*, 2007, pp. 131–136.

[43] M. Kaaniche, L. Romano, Z. Kalbarczyk, R. Iyer, and R. Karcich, “A hierarchical approach for dependability analysis of a commercial cache-based RAID storage architecture,” in *Proc. FTCS*, 1998, pp. 6–15.

[44] M. Kaaniche, J.-C. Laprie, and J.-P. Blanquart, “Dependability engineering of complex computing systems,” in *Proc. ICECCS*, 2000, pp. 36–46.

[45] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo, “Memory leak analysis of mission-critical middleware,” *Journal of Systems and Software*, vol. 83, no. 9, pp. 1556 – 1567, 2010.

[46] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, “Software aging analysis of the linux operating system,” in *Proc. ISSRE*, 2010, pp. 71–80.