SABRINE: State-Based Robustness Testing of Operating Systems

Domenico Cotroneo*†, Domenico Di Leo*†, Francesco Fucci*, Roberto Natella*†
*DIETI department, Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Napoli, Italy
†Critiware S.r.l., Incipit, via Cinthia, Complesso Univ. Monte S. Angelo, 80126, Napoli, Italy
{cotroneo, francesco.fucci, roberto.natella}@unina.it, domenico.dileo@critiware.com

***Abstract*—The assessment of operating systems robustness with respect to unexpected or anomalous events is a fundamental requirement for mission-critical systems. Robustness can be tested by deliberately exposing the system to erroneous events during its execution, and then analyzing the OS behavior to evaluate its ability to gracefully handle these events. Since OSs are complex and stateful systems, robustness testing needs to account for the timing of erroneous events, in order to evaluate the robust behavior of the OS under different *states*. This paper presents SABRINE (StAte-Based Robustness testIng of operatiNg systEms), an approach for state-aware robustness testing of OSs. SABRINE automatically extracts state models from execution traces, and generates a set of test cases that cover different OS states. We evaluate the approach on a Linux-based Real-Time Operating System adopted in the avionic domain. Experimental results show that SABRINE can automatically identify relevant OS states, and find robustness vulnerabilities while keeping low the number of test cases.**

***Index Terms*—Robustness Testing; Fault Injection; Operating Systems; Linux kernel; Fault Tolerance; Dependability Benchmarking**

I. Introduction

Operating Systems (OSs) are the core of a wide range of software systems with which we interact in our everyday life, ranging from embedded systems to large critical infrastructures. OS failures are a major concern, as they can potentially compromise all the applications running on it and the mission of the overall system [44], [11], [29], [28], [4], [37]. Given that an OS can execute under a large variety of scenarios, facing unexpected and erroneous conditions that can arise from its working environment (including the hardware, applications, users and other systems), it is expected that the OS operates *robustly* even in such conditions. Therefore, it is of paramount importance, especially in the context of mission-critical systems, to assess the robustness of OSs with respect to such unexpected conditions (e.g., an invalid input), that is, its ability to gracefully handle these anomalous events and to avoid catastrophic consequences, e.g., an OS crash [20].

Robustness testing consists in executing a software system in the presence of unexpected events (e.g., invalid or untimely inputs) that are deliberately introduced ("injected") during the test. Several approaches have been proposed, but robustness testing of OSs is still an open challenge. Most of robustness testing approaches rely on a formal description of the software (e.g., Timed Input/Output Automata) for identifying unexpected events to inject, the states in which to inject and the expected responses of the system to events [14], [2], [30], but these approaches do not scale well for OSs, given that OSs can be very large and complex (up to millions of lines of code), and that their source code and/or expertise on their internals may not be available. Alternative proposals are instead based on black-box approaches [28], [1], [24], in which OS robustness test cases are devised from the analysis of the input domain of OS interfaces (e.g., the system call interface). However, these approaches neglect *the state of the OS* at which an anomalous event is injected, by injecting at random times, or require extensive manual analysis in order to define when to perform the injection. These limitations affect the efficiency of robustness testing, since they can reduce the number of *robustness vulnerabilities* that are found, and increase the number of experiments to perform.

In this paper, we propose an approach for state-aware robustness testing of OSs, namely *SABRINE* (*StAte-Based Robustness testIng of operatiNg systEms*). SABRINE improves the efficiency of robustness testing, by performing exactly one robustness test for distinct states of the OS, thus avoiding superfluous experiments (i.e., experiments that inject a fault in the same OS state), and increasing the likelihood to find robustness vulnerabilities (by covering more OS states than a random approach). SABRINE does so by *automatically* extracting behavioral models from execution traces of the target system, and by generating a distinct robustness test case for each state of the behavioral model of the system. We evaluate the approach on a Linux-based Real-Time Operating System adopted in the avionic domain. Experimental results show that SABRINE can automatically identify relevant OS states, and find robustness vulnerabilities while keeping low the number of test cases.

The paper is structured as follows. Section II provides an overview of studies on robustness testing of OSs. Section III presents the SABRINE approach. Section IV describes the case study on which we applied the proposed approach, and Section V reports and discusses experimental results. Section VI concludes the paper.

II. Related work

Several studies approached the problem of robustness testing of operating systems, from different points of view. One of the earliest study has been presented in [34], which evaluated the robustness of UNIX utilities in the presence of random

inputs ("fuzzing"). Two tools, respectively *Fuzz* and *ptyjig*, were proposed to submit a random stream of data to the target through the standard input and through the terminal device. The study found that a significant number of utility programs on three UNIX systems (between 24% and 33%) is vulnerable to invalid inputs, causing process crashes or stalls. A subsequent experiment [35] found that the same utilities were still sensible to a significant part of faults found in [34] 5 years later. These studies highlighted that robustness can be a serious concern even for mature, widely-adopted software.

Even if the fuzzing approach is simple to implement and can reveal robustness problems, its efficiency was questioned by some studies, since it relies on many trials and "good luck". In [17], it is pointed out that most of unstructured random tests only test the input parsing code of the program, and do not stress other software functions. *RIDDLE* [17], a tool for robustness testing of Windows NT utilities, extends fuzzing with erroneous inputs generated by a grammar that describes the format of inputs like a Backus-Naur form. These erroneous inputs (random and boundary values) are syntactically correct, and able to test more thoroughly the target program.

In order to improve the efficiency of robustness testing, subsequent studies investigated the *data-type based* error injection approach, which focuses on *invalid inputs* that tend to be more problematic than other to be handled. Such an approach was proposed in [11], [28] (the *BALLISTA* tool) for evaluating the robustness of commercial OSs, with respect to their ability to handle invalid inputs from faulty user-space programs to the *system call interface* [21]. Each robustness test consists of a system call invocation with a combination of both valid and invalid parameters. For each group of system calls and each data type, these studies define a set of invalid input values (e.g., closed or read-only files, and NULL or wrong pointers to memory areas). Examples of invalid inputs, using a data-type based approach on three data types, are provided in Table I. The test outcome is determined by recording the error code returned by the system call (e.g., to identify whether the error code reflects or not the invalid input, or an error code is not returned at all), and by monitoring system processes using a watchdog process (e.g., a failure occurs if a process unexpectedly terminates during the experiment, or it is stalled). Test outcomes are classified by severity according to the CRASH scale: a Catastrophic failure occurs when the failure affects more than one task or the OS itself; Restart or Abort failures occur when the task launched by BALLISTA is killed by the OS or stalled; Silent or Hindering failures occur when the system call does not return an error code, or returns a wrong error code. These studies found severe robustness vulnerabilities in several commercial OSs, which were due mainly to illegal pointer values, numeric overflows, and end-of-file overruns [28].

Robustness testing of OSs has also been focused on device drivers, since they are usually provided by third party developers and represent a major cause of OS failures [6], [15]. The robustness of the *Driver Programming Interface*, DPI, of OSs has been targeted in [1] and [12], in which invalid values are generated by faulty device drivers when they invoke a function of the OS kernel: in [1], invalid values are introduced using a data-type based approach, while in [12], the code of device drivers is mutated (by artificially inserting bugs) to cause a faulty behavior. Johansson et al. [24], and Winter et al. [47] later, compared the bit-flipping, fuzzing, and data-type based approaches with respect to their effectiveness in detecting vulnerabilities in the DPI of Windows CE, and the efforts required to setup and execute experiments. They found that bit-flipping is the approach most effective at finding vulnerabilities, but it incurs a high execution cost due to the large number of experiments, thus providing a low efficiency, while the other approaches are more efficient but incur in a higher implementation cost (e.g., in the case of the data-type based approach, the user has to define exceptional values for each data type). From all these studies, OSs result to be more vulnerable to device drivers than to applications, since developers tend to omit checks in the device driver interface to improve performance, and because they trust device drivers more than applications. Other works assessed the robustness of OSs with respect to hardware faults (e.g., CPU and disk faults), by corrupting OS memory image [18], [5], and with respect to synchronization faults in kernel code [9], [39].

All these approaches neglect the system state, or they rely on a representative workload to exercise the system and bring it to relevant states before a robustness test. A relevant example is represented by *dependability benchmarks* [26], [25], [27], which have been proposed for comparing the robustness of different OSs. These dependability benchmarks evaluate robustness while the target OS is under different working conditions (i.e., state): they define realistic scenarios in which the OS is part of a database server system or mail server system, and the system is exercised using a representative set of user requests. System call inputs generated by user-space applications (e.g., the DBMS or the mail server processes) are intercepted and replaced with invalid ones, by using respectively data-type based values, random values, and bit-flips (i.e., a correct input is corrupted by inverting one bit).

The influence of OS state has been investigated in recent work on testing device drivers [23], [43]. In [23], the concept of *call blocks* is introduced, based on the observation that device drivers issue recurring sequences of function calls (e.g., when reading a large amount of data from a device). Therefore, Johansson et al. [23] improved the efficiency of robustness testing by focusing on call blocks (i.e., repeating subsequences of OS function calls), instead of injecting invalid inputs at random time. Call blocks are identified by the tester through a manual analysis before performing robustness tests. Sarbu et al. [43] proposed a state model for testing device drivers of Microsoft Windows OSs, using a vector of boolean variables. Each variable represents an operation supported by the device driver: at a given time t, the i-th variable is true if the driver is performing the i-th operation. They found that the use of a state model can reduce the number of tests. Prabhakaran et al. [40] proposed an approach for testing journaling file systems, which injects disk faults at specific states of file

TABLE I
EXAMPLES OF INVALID INPUT VALUES FOR THE THREE DATA TYPES OF THE *write(int filedes, const void *buffer, size_t nbytes)* SYSTEM CALL.

File descriptor (filedes)	Memory buffer (buffer)	Size (nbytes)
FD_CLOSED	BUF_SMALL_1	SIZE_1
FD_OPEN_READ	BUF_MED_PAGESIZE	SIZE_16
FD_OPEN_WRITE	BUF_LARGE_512MB	SIZE_PAGE
FD_DELETED	BUF_XLARGE_1GB	SIZE_PAGEx16
FD_NOEXIST	BUF_HUGE_2GB	SIZE_PAGEx16plus1
...	...	...

system transactions. In [8], we conducted a preliminary study on the impact of the OS state on OS failures and on the code coverage achieved by robustness tests at system calls.

All these studies showed that the OS state has an important role in testing such complex systems; however, they *require knowledge about OS internals*, and *a manual analysis to define state models* in which to inject faults. The objective of this paper is to overcome this limitation, by automatically inferring state models to use in robustness testing.

III. ROBUSTNESS TESTING APPROACH

Our approach for robustness testing has been designed to perform the injection of faults in the OS, by taking into account the state of the OS during the injection. The state can affect how an event impacts on the OS and the ability of the OS to robustly handle its occurrence. We assume that the tester has little knowledge about the inner workings of the OS, since OSs are typically provided by a third-party and it would be unfeasible to manually analyze them to understand in-depth their complex behavior. Therefore, we identify the states of the OS through a black-box approach: we analyze the sequence of events produced by the OS (observed at its interfaces) during its execution, based on the idea that the history of events at OS interfaces reflect the state of the OS. Test cases should run at distinct sequences of events in order to cover different states of the OS and to efficiently perform robustness testing (i.e., avoiding that several tests impact the same state, wasting testing efforts and time). We identify OS states using *behavioral model mining* techniques. These techniques have been adopted for automating several software engineering tasks, including specification mining [3], automated debugging [10], and reverse engineering [46]. To the best of our knowledge, this is the first work that tailors these techniques for robustness testing of OSs.

The SABRINE approach automates collection and analysis of OS events, to identify interesting sequences of events in which to inject faults. The process consists of 5 phases (Fig. 1):

1) **Behavioral Data Collection**: Before performing robustness testing, the system is executed and profiled under fault-free conditions. This phase collects data about the OS behavior, in terms of interactions between OS components at run-time.
2) **Pattern Identification**: Behavioral data is preprocessed, by dividing them in *sequences*. Each sequence is a set of interactions that occur during the execution of an individual system call or interrupt request. Identical sequences are grouped together, and represent a *pattern*.
3) **Pattern Clustering**: Patterns that are similar (even if they are not identical) are further grouped together, using a clustering algorithm. This phase is important for obtaining a compact and efficient set of robustness test cases.
4) **State Model and Test Suite Generation**: For each cluster obtained from the previous phase, a *behavioral model* is generated. A behavioral model consists of a set of states interconnected by events. Robustness test cases are generated for each model, in which faults are injected in specific states of the model.
5) **Test Execution**: Each test executes the system under the same working condition of the first phase. During execution, behavioral data is collected and analyzed at run-time, and a fault is injected when the OS reaches a given state of the behavioral model.

We first define some basic concepts in Section III-A, then we discuss in detail each phase in the following sections.

A. Definitions

In our approach, we distinguish between the different *components* that form an OS. A component is a subsystem of the OS that is responsible for managing a resource or for providing a set of services, such as memory management, I/O management, and process scheduling. Each component provides an *interface* to other components, that is, a set of functions that are invoked to request a service. Applications can require a service to the OS by performing a *system call*, which in turn triggers one or more components that interact in order to implement the OS service (Figure 2). Additionally, component services can be invoked by *interrupt requests* coming from the hardware, and by *kernel tasks*, i.e., processes that execute in kernel space and that can directly interact with OS components. Several system calls, interrupt requests and kernel tasks can be executed in parallel (by alternating on the same CPU, or by running concurrently on different CPUs). The applications that run on top of the OS and exercise it are referred to as the *workload*.

We test OS robustness against *service failures* of a component. A service can fail, for instance, due to the exhaustion of a resource, or due to a hardware fault in a device or a defect in an OS component. In case of a failure, the function that has been invoked typically returns an *error code* to notify that a service cannot be provided. A service failure may cause

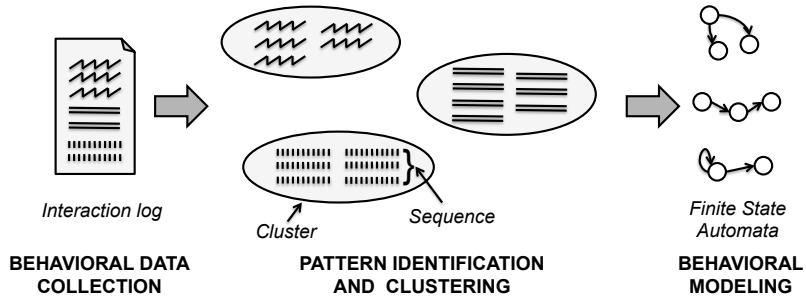

Fig. 1. Overview of the SABRINE approach.

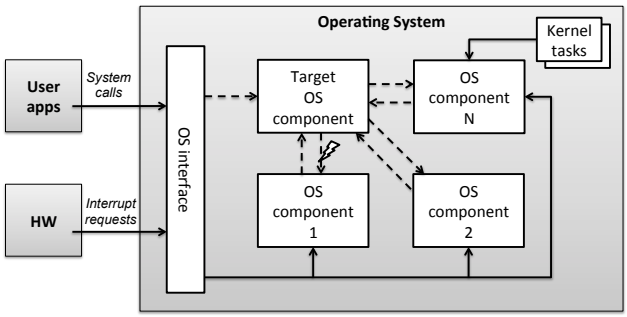

Fig. 2. System overview.

a *non-robust behavior*, such as an OS crash, when it is not correctly handled by the OS code that invokes the service. In such a case, the OS is considered *vulnerable* to that service failure, and a *robustness vulnerability* has been found, which may require to fix the OS in order to make it robust against the service failure (e.g., by retrying the failed operation, or switching to a degraded mode of service). To test robustness, we force a service failure (also referred to as *fault*) while the system is exercised with a workload, that is, by forcing the called function (representing the service) to return an error code, and analyzing the system reaction to the service failure.

In particular, given that the same service can be requested by several OS components, we focus on service invocations performed by one specific *target component* at a time. For instance, the target component can be represented by a new component under development, such as a device driver to support new hardware, or a new filesystem component. To identify the states of the target OS component, we log *interactions* at its interfaces with other OS components (dashed arrows in Figure 2). The target component may be invoked by another component (*input interaction*), or the target component may invoke another component (*output interaction*). An interaction with a function that can fail (e.g., a function for resource allocation), and in which a failure can be injected, is referred to as *injectable interaction* (see Figure 3). We both consider the case in which an injectable interaction is *direct*, that is, the injectable function is invoked by the target component, and the case in which the injectable interaction is *indirect*, in which the injectable function is invoked by another component on behalf of the target component (i.e., the function is invoked to provide a service to the target component). We include indirect interactions in our robustness tests since the fault may propagate to the target component and trigger its robustness vulnerabilities.

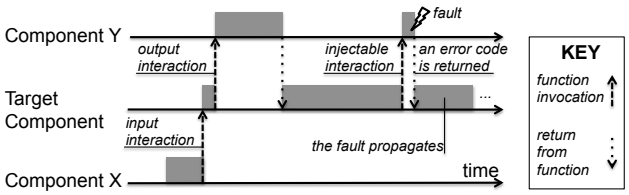

Fig. 3. Interactions among OS components.

B. Behavioral Data Collection

In this phase, the OS is executed using a workload, without injecting faults. The workload is selected by developers and testers before performing robustness testing. In a similar way to performance benchmarks, the workload reflects the context in which the OS will be adopted (e.g., web applications, DBMSs, ...), and it affects the way the OS is exercised during the tests (e.g., a DBMS-oriented workload stresses storage-related services) and its behavior under unexpected events. During the workload execution, we monitor component interactions (Figure 2), and derive state models for the target component on the basis of its interactions with other components. As discussed later, state models are based on *input*, *output*, and *injectable interactions* that involve the target component.

Component interactions are monitored through *static* (i.e., hard-wired in the kernel source code) or *dynamic probes* (i.e., inserted at run-time) located at the interfaces of components. A probe consists of a small piece of code (e.g., a breakpoint) that is inserted in a given code location, and that triggers a *handler routine* when executed. In turn, the handler collects information and restores kernel execution. For tracing input interactions, we probe the interface of the target component, storing information (as described in the next section) about the component that invokes the target. For tracing output and injectable interactions, we probe the interfaces of components that are invoked by the target component, storing information

about the invoked component. Probing at component interfaces represents a practical solution for most of modern commodity OSs, since they often provide tools that allow to insert static and/or dynamic probes in kernel code and monitor its execution with a low overhead, such as DTrace for Oracle Solaris [33], SystemTap for Linux [22], and DebugView for Microsoft Windows [42]. Data from probes are transferred to an external computer through a *serial port*, which is typically adopted for debugging purposes since the serial port driver has minimal interactions with other OS components, thus limiting interferences on OS execution due to monitoring.

When collecting behavioral data, we take into account that component interactions may vary between different workload executions due to random factors: for instance, some interactions may appear in a different order or not appear at all, depending on the timing of I/O events and process scheduling. As a consequence, such random factors can affect the definition of robustness test cases, since some OS states can be missed during an individual workload execution. For this reason, we repeat the execution of the workload several times during this phase: by doing so, we are able to include sets of interactions even when they do not occur at every execution, and to generate robustness test cases that also cover them, leaving uncovered only very rare interactions.

C. Pattern Identification

The output of the previous phase consists of an *interaction log*, in which interactions among components appear in sequential order (i.e., ordered by their timestamp). The log is divided into *sequences*, where each sequence is a set of events that occur during the execution of an individual system call, interrupt request, or a kernel task. Two executions of the same system call represent two distinct sequences, but they can produce different sets of interactions, depending on the state of the system. By dividing the log into sequences, we discriminate subsets of interactions that repeat identically (in this phase) or are similar (in the next phase), in which it is likely that the target has assumed the same states. Identical sequences are grouped together, forming a *pattern*.

To extract sequences, we log the following information for each interaction between the target and other components:

- **Operation ID**: A string identifier of the operation (system call, interrupt request, or kernel task) that is being serviced at the time of the interaction.
- **Execution ID**: An integer that identifies a specific execution of a system call, interrupt request or kernel task. Each time that an operation starts, a new execution ID is generated, and all the interactions produced during this operation will be identified by this value. If a system call is started while the same system call is already executing (e.g., by a different process on a different CPU), the interactions of the new execution have the new execution ID, while the interactions generated by the other one are still denoted by the previous execution ID.
- **Trace ID**: An integer that identifies a specific execution of the whole workload. Since the workload can be executed several times, and more than one workload execution can appear in the same interaction log, the interactions of each workload execution are denoted by a specific trace IDs.

Moreover, each logged interaction contains the following information:

- **Called function**: In the case of input interactions, it is the name of the function of the target invoked by another component. For output and injectable interactions, it is the name of the function invoked by the target in another component.
- **Call point**: The code location in which the function is invoked (e.g., the address in the executable code of the instruction that invokes the function).

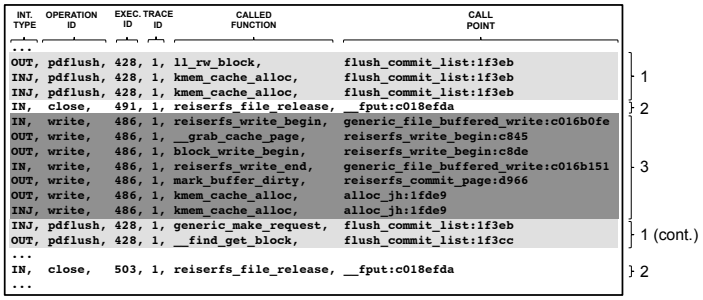

Fig. 4. Example of interaction log and pattern identification.

Figure 4 shows an extract of the interaction log from the case study that we will consider in this work. The first sequence in the example (highlighted in light gray) is identified by the triple $\langle pdflush, 428, 1\rangle$, in which there are two output interactions (denoted by "OUT") and two injectable interactions (denoted by "INJ"), and all of them are invocations made by the *flush_commit_list* function of the target component (a filesystem). This sequence is interleaved with two other ones, identified by $\langle close, 491, 1\rangle$ (white background) and $\langle write, 486, 1\rangle$ (dark gray background) respectively. This interleaving occurred since *pdflush* (a kernel task) has been suspended while executing *kmem_cache_alloc*, which performs memory allocation and can preempt a task when this operation takes a long time (e.g., an I/O operation is required in order to free memory). The other two sequences are generated by the workload invoking the *close* and *write* system calls, which in turn trigger input interactions with the target component (denoted by "IN"). When the third sequence performs its second memory allocation, a workload process is preempted in favor of *pdflush*, which continues the first sequence. The same sequence of interactions can repeat identically in the log, with a different identifier: this is the case of the sequence identified by $\langle close, 503, 1\rangle$ (a sequence containing only one interaction), which is identical to the sequence identified by $\langle close, 491, 1\rangle$. These sequences represent two instances of the same pattern (number 2), and only one instance per pattern is considered in the subsequent phases.

D. Pattern Clustering

The execution of the OS typically leads to patterns that are not identical, but differ for a few interactions, or there is a small variation in the order of the interactions. In other words, several patterns tend to be very "similar". Small variations in the sequences are unavoidable, and are due to non-deterministic factors that can affect OS execution. Figure 5 shows two similar patterns related to the *write* system call (for the sake of readability, only the called function is showed for each interaction). The patterns p_1 and p_2 exhibit almost the same number and sequence of interactions, aside from three interactions (gray background) which appear only in p_2. These interactions, in this specific case, represent the allocation of additional memory when metadata are written to the disk.

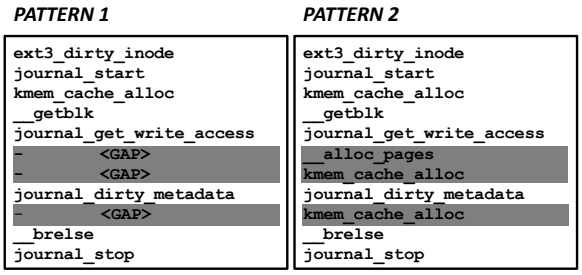

Fig. 5. Example of similar patterns.

However, generating one behavioral model for each individual pattern would lead to an excessive number of models and, as discussed later, to superfluous robustness test cases. Therefore, before generating behavioral models, we group together similar patterns, thus obtaining *clusters* of patterns. Each cluster represents a specific "mode of operation" of the target component, where the patterns in a given cluster only differ with respect to a few interactions. To perform clustering, we first measure the similarity among all pairs of patterns using a *similarity function*, and then we cluster patterns that are similar with a *clustering algorithm*.

A similarity function is a quantitative way to express the similarity between two sequences, and it is used in several applications, such as the processing of biological sequences. In our case, we compare sequences of interactions, in which each interaction (i.e., a pair $\langle$ *called function*, *call point* $\rangle$) represents an element of the sequence. Two main approaches exists in the literature for evaluating similarity, which respectively (i) only consider which elements appear in each sequence, and evaluate the number of elements that appear in both sequences (*set-based similarity functions*), and (ii) consider the ordering of elements while comparing common elements between the sequences (*sequence-based similarity functions*) [19]. In our approach, we measure the similarity between patterns with a sequence-based function: two sequences of interactions with different orderings may reflect different states of the system, and should be regarded as dissimilar.

Sequence-based functions are based on "alignment" algorithms, in which the elements of the sequences are placed side by side in order to maximize the number of matches, and minimizing the number of gaps and mismatches[1]. With the Smith-Waterman algorithm [13], we compute an alignment score for each pair of patterns p and q according to the following dynamic programming formulation:

$$F_{0,j} = -j * g \quad , \quad F_{i,0} = -i * g$$

$$F_{i,j} = max \begin{cases} F_{i-1,j-1} + sim(p_i, q_j) \\ F_{i-1,j} - g \\ F_{i,j-1} - g \\ 0 \end{cases}$$

$$sim(p_i, q_j) = \begin{cases} m & \text{if } p_i = q_j \\ -n & otherwise \end{cases}$$

In this set of equations, p_i and q_j are the i-th and j-th element of patterns p and q, respectively, with $i \in [1, ..., N]$ and $j \in [1, ..., M]$, and N and M are the lengths of patterns p and q. F is the scoring matrix, where the value $F_{i,j}$ is the score of the best alignment between the initial segment $p_{1...i}$ of p up to p_i and the initial segment $q_{1...j}$ of q up to q_j, which is calculated recursively from $F_{i-1,j}$, $F_{i,j-1}$, and $F_{i-1,j-1}$ [13]. The constants g and n are the score penalty for gaps and mismatches, while m is a score reward for matches. Common choices are $g = 1$, $n = 2$, and $m = g + n$ [19]. The highest value of F, that is $F_{N,M}$, represents the score of the best possible alignment. For instance, the patterns p_1 and p_2 showed (aligned) in Figure 5 have score $SW(p_1, p_2) = W_{match} * m + W_{mismatch} * n + W_{gap} * g = (8) * (+3) + (0) * (-2) + (3) * (-1) = 21$, where the Ws are the number of matches, mismatches and gaps, respectively. For each pair of patterns, we compute the SW score, and collect this score into a cell of a matrix, named Similarity Matrix (SM), which expresses quantitatively the degree of similarity among all pairs of patterns. The score of each pair is normalized using the length of the longest pattern in each pair, since patterns in our context have variable length, which would otherwise affect the evaluation of pattern similarity.

We group together similar patterns using a *spectral clustering* algorithm [45]. Spectral clustering groups a set of elements on the basis of their similarity matrix, and has recently emerged as an effective and computationally-efficient clustering approach. A spectral clustering algorithm interprets input elements as the nodes of a graph, and the similarity score of each pair of elements as the weight of the connection between two nodes. Then, elements are clustered into k groups, by performing k cuts in the graph, each group includes the nodes that are still connected after the cuts. The idea behind spectral clustering is that cutting "weak" connections splits the graph into partitions of elements that are "strongly connected" and thus very similar each other. The weights of cuts in the graphs are closely related to the *spectrum* of

[1]When the elements at a given position of a pair of patterns are the same, we say that there is a *match*; otherwise we say that there is a *mismatch*. A *gap*, instead, consists in introducing a special symbol to fill the vacuum due to the different lengths of the two sequences.

the graph, that is, the eigenvalues $\lambda_1, \ldots, \lambda_n$ of the graph laplacian matrix L derived from SM [7]. By processing L on the basis of its eigenvectors, spectral clustering obtains k cuts and, in turn, k clusters. To select the number k of clusters, we use the *eigengap* heuristic [45], which chooses k such that all $\lambda_1, ..., \lambda_k$ eigenvalues of L are very small and λ_{k+1} is relatively large. Intuitively, if the first k eigenvalues are very small, then the algorithm can split the graph into k parts without separating strongly-connected nodes.

E. Behavioral Modeling and Test Suite Generation

At this point, the initial interaction log, through pattern identification and clustering, has been turned into clusters of sequences. From each cluster, we infer a behavioral model in the form of a *finite state automata* (FSA). We based our robustness test generation approach on the kBehavior mining algorithm [32], [31]. This algorithm incrementally infers FSAs from execution traces, which in our case consist of sequences of component interactions. The algorithm starts with an empty automata (e.g., only one state with no transitions), examines the first pattern and generates an FSA whose transitions are labeled with an interaction (i.e., a pair $\langle called\ \ function, callpoint \rangle$). If the cluster contains more than one pattern, they are subsequently provided as input to the mining algorithm, one at at a time. Each time that a new pattern is provided, the algorithm augments the FSA with new transitions and states, in order to reflect both the new patterns and previous ones. This process is repeated for each cluster, leading to an FSA for each cluster (as showed in Fig. 1).

A set of robustness test cases is derived from each FSA. SABRINE identifies transitions of the FSA that represent an injectable interaction, i.e., an invocation of a function in which a failure can be injected. Then, SABRINE automatically generates a test case for each injectable interaction present in the FSA. More specifically, given a state S with an outgoing transition t such that t is an injectable interaction, a robustness test case is generated for the couple (S, t). The test consists in forcing a failure of the function when the system reaches the state S and the injectable function is invoked. The state S represents the *context* in which the function can be invoked and can fail. This approach allows SABRINE to cover each different context in which the injectable function is invoked, and to improve the efficiency of robustness tests.

Figure 6 provides an example of how robustness test cases are obtained from behavioral models. It depicts the FSA generated from the two patterns in Figure 5 (for brevity, only the called function is showed at each transition in the figure). States from 0 to 5, and states from 6 to 8 are connected by interactions that appears in both patterns, while states 5 and 6 are connected by two different sets of transitions. This occurs since the two patterns share most of their interactions, but one of them performs contains additional memory allocations, and the mining algorithm inserted new states and transitions corresponding to these interactions. Assuming to inject failures at the invocations of the *kmem_cache_alloc* memory allocation function, SABRINE generates 3 robustness test cases from the FSA. The example also points out the importance of clustering on the generation of test cases. The first invocation of *kmem_cache_alloc*, which appears in both patterns of Figure 5, is performed in the same context in both patterns. By using only one FSA for representing both patterns, the occurrences of the first *kmem_cache_alloc* invocation are collapsed into only one transition in the FSA, the one between states 2 and 3. In this way, we reduce the number of robustness test cases (only one test is performed for each transition with an injectable interaction), while still covering relevant states of the system, thus improving the efficiency of robustness testing. A similar reduction is obtained when the *kmem_cache_alloc* is performed in a loop: in such cases, since the same interactions are repeated several times, the mining algorithm translates these interactions into a loop in the FSA, and only one test case is generated for that injectable interaction.

F. Test Execution

Robustness test cases are translated in *test programs* that are then executed to inject service failures in the different states of the OS. In a similar way to the "Behavioral Data Collection" phase (Subsection III-B), the test program collects interaction sequences at run-time using kernel probes, and keeps track of the current state of the target component. If the test program, in the current state, observes the interaction specified in the FSA, it transits to the new state. If the observed interaction it is not the expected one, the target program transit to the initial state. When the system reaches the target state S, the target program injects a service failure during the injectable interaction. After the injection, the OS behavior evolves freely.

IV. Case study

To illustrate the use of the SABRINE approach, we consider an OS developed in the context of a pilot R&D project, in conjunction with the Finmeccanica s.p.a. industrial group. The goal of the project is to develop a reliable Linux-based Real-Time Operating System (RTOS), namely FIN.X-RTOS to adopt in software systems for avionic applications. In particular, in order to ease the certification of systems based on FIN.X-RTOS, the OS needs to be accompanied by evidences (e.g., test artifacts) showing the compliancy to the recommendations of the DO-178B safety standard [41]. The original Linux kernel has been enhanced in FIN.X-RTOS by providing hard real-time and scalability on multi-core architectures, and removing unnecessary parts. The requirements of the standard at level D (to be followed for software whose anomalous behavior would cause "a minor failure condition" for the aircraft) have been fulfilled. At the time of writing, FIN.X-RTOS is being tested with additional verification activities according to the requirements of level C (for software that may cause "a major failure condition" for the aircraft), which demand to test the robustness of the software against abnormal inputs and conditions. An example of requirement from the standard is to "*provoke transitions that are not allowed by the software requirements*" [41].

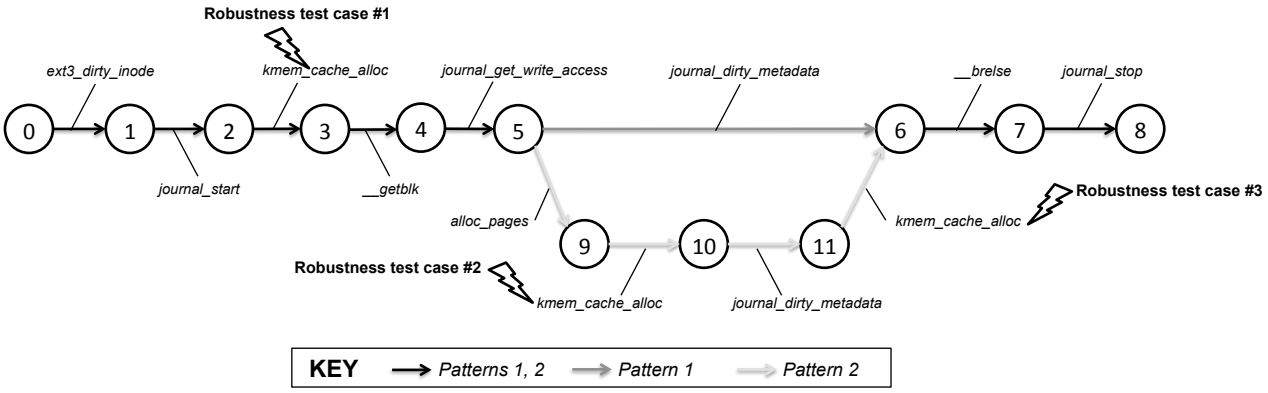

Fig. 6. Example of behavioral model.

We applied the SABRINE approach to assess the robustness of a set of I/O-related components against service failures in the memory allocator of the kernel. We selected memory allocation failures because most of OS components depend on this service, and it is a frequent cause of system failures [16]. Kernel developers also perceive memory allocation problems as a likely cause of OS failures: in fact, the Linux kernel includes a framework for injecting service failures, which encompasses memory allocation failures [36]. Both in our implementation of SABRINE and in the Linux injection framework, a failure is injected by forcing a memory allocation function (*kmem_cache_alloc*) to return a NULL pointer instead of a valid pointer to the newly allocated memory. While the Linux framework injects failures at a random time, the SABRINE approach selects the time in which to inject based on the state of the target component. We compare both these approaches in our experiments. It is important to note that the SABRINE approach is not limited to the *kmem_cache_alloc*, since several other injectors, both in the literature [17], [28] and among practitioners, also force the failure of a kernel function to test robustness. For instance, the Linux kernel fault injection framework also allows to emulate disk I/O errors by forcing the failure of functions for block I/O management [36].

The relationships between I/O-related components in the kernel are showed in Figure 7. In this architecture, I/O system calls (e.g., *write*s) first pass through the Virtual File System, which provides generic services for implementing file systems, and forwards a file operation to the specific filesystem that manages the file (e.g., EXT3, ReiserFS, ...). The file system can issue an I/O operation to the Block I/O Layer, which provides generic services such as scheduling of I/O requests and caching of disk data. In turn, the Block I/O Layer forwards requests to a device driver, which manages the disk device. All these components use the memory allocator for dynamically allocate memory, such as for storing file metadata and for temporary I/O buffers. The target components are represented by thick boxes in Figure 7, and include two widely-adopted file systems (EXT3 and ReiserFS) and a device driver (the SCSI subsystem). We adopt the *Apache HTTPD* web server to exercise the OS, using the *httperf* performance testing tool to generate web requests [38]. Experiments were executed in a virtual machine environment, and were fully automated using programs running on the host machine. A SystemTap program [22] collects behavioral data. FSA models are created with the kBehavior algorithm [32], [31], and automatically translated in test programs implemented in the SystemTap language. Behavioral data and error messages from the OS are collected using virtual serial port connections. In the case of an OS crash, we collect information including the type of exception (e.g., illegal memory access), the code location, the contents of the stack and of CPU registers. The virtual machine is automatically rebooted in case of a crash.

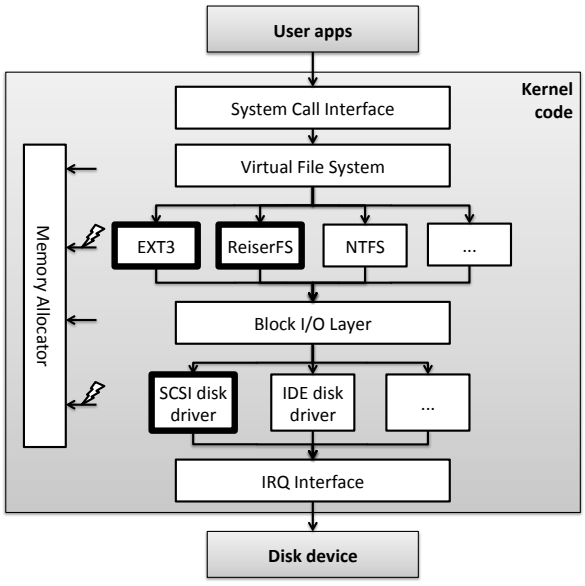

Fig. 7. Overview of I/O-related subsystems in FIN.X-RTOS.

V. Experimental results

This section presents the results obtained by SABRINE in our case study. Table II provides some basic facts about data collection and processing, and test generation. We collected an interaction log for each target component, by running the web

TABLE III
CLUSTERS FOR EXT3.

Cluster	Behavior	Context	# patterns
1	gets and sets the file metadata	*stat* syscall	6
2	retrieves and stores in memory the file index block, or updates it on the disk	*open*, *unlink* syscalls	5
3	copies file contents from disk to a cache, and modifies it	*write* syscall	8
4	copies a small amount of data from a file to a network socket	*sendfile* syscall	10
5	modifies the contents of a file already in the disk cache	*write* syscall	8
6	flushes a small amount of data from the cache to the disk	*pdflush* kernel task	19
7	flushes a large amount of data from the cache to the disk	*pdflush* kernel task	6
8	copies a large amount of data from a file to a network socket	*sendfile* syscall	12
9	updates file metadata to reflect that is has been memory-mapped	*mmap2* syscall	5

TABLE II
STATISTICS ON THE BEHAVIORAL DATA COLLECTION AND TEST CASES.

	EXT3	ReiserFS	SCSI
# interactions	34,784	97,341	27,311
*# sequences**	432	239	1,307
*# patterns**	79	57	10
# clusters	9	6	2
# test cases	49	28	10

* *involving the* kmem_cache_alloc *function.*

server workload 10 times for each target. For EXT3 and ReiserFS, interactions were performed in the context of file-related system calls, such as *open* and *write*, and of the *pdflush* kernel thread of the Block I/O Layer, which periodically flushed cached data to the disk. For SCSI, interactions were initiated by kernel threads of the Block I/O Layer, which requested data transfers, and by interrupts from the hardware. Several thousands of interactions appear in each log. These logs were divided in sequences, and identical sequences were grouped into patterns. Since we aimed at injecting service failures of the *kmem_cache_alloc* function, we focused our analysis only on sequences interacting with this function. For EXT3 and ReiserFS, a non-negligible number of patterns and clusters was generated, since memory allocations were performed in many different contexts during filesystem operations. Instead, even if SCSI produced the highest number of sequences with *kmem_cache_alloc*, it exhibited the lowest number of distinct patterns and of clusters: for this target, memory allocations performed always at the same code location (i.e., when allocating memory for storing a new data transfer command), leading to repetitive sequences. Consequently, only 2 clusters are enough to group the patterns of SCSI, while EXT3 and ReiserFS require 9 and 6 clusters, respectively.

We examined in depth the clusters, in order to understand the mode of operation represented by each pattern, and to assess whether clustered patterns are "semantically" similar. Table III provides a description for the clusters of EXT3; similar interpretations apply to ReiserFS and SCSI clusters, but we do not show them due to space constraints. Column "Behavior" provides a brief description of clusters, and column "Context" details the system calls or kernel task in which these behaviors were observed. Each cluster represents a distinct behavior of the file system. For instance, cluster 1 gathers the patterns representing "get" and "set" operations on file metadata (e.g., file permissions), which is the case of the *stat* system call; clusters 2 and 3 represent the typical behavior of *read* and *write* system calls. For each cluster of each target component, we derived an FSA, and a set of one or more test cases for each FSA (Subsection III-E).

In order to evaluate the efficiency of SABRINE, we executed the robustness test cases generated by the approach, and compared the results with the ones obtained using the standard fault injection framework included in the kernel [36]. In the standard injection framework, allocation failures are injected randomly: each time *kmem_cache_alloc* is invoked, it can fail with a fixed probability P; if a failure is not injected, the subsequent invocation becomes the next candidate injection. Moreover, the standard injector allows to inject service failures when the injectable function is invoked (directly or indirectly) by the target component (EXT3, ReiserFS, or SCSI) [36]. We performed 1,000 random injections for each target component; these experiments took a few days per target component to complete, therefore 1,000 injections can be considered a conservative estimate on the number of experiments that a developer would perform. We set $P = 10\%$, which is the value suggested by kernel developers in the documentation; this value avoids that too many injections take place only at the beginning of the experiment. As for SABRINE, we executed the number of tests reported in the last row of Table II. We classify the outcome of a test in:

- **Kernel Failure**: the OS is crashed, or its state is corrupted. To detect state corruptions in the OS, we enabled several consistency checks introduced by developers in the kernel code, including checks on stack overflows, stuck system calls, locks not released, and corruptions on key kernel data structures. This kind of failures is the most severe, since they affect the whole system.
- **Workload Failure**: the web server crashes, exits abnormally, does not reply to requests, or does not execute correctly the requests. These failures are detected through the logs of the web server and of the client.
- **FS Corruption**: after each test, we detect disk corruptions using filesystem check utilities.
- **No Impact**: neither the OS nor the workload show an erroneous behavior.

TABLE IV
STATISTICS ON FAILURE DISTRIBUTIONS.

Testing Technique	Target	Kernel Failures	Workload Failures	FS Corruptions
Random	EXT3	32.8%	0%	0%
	ReiserFS	9.6%	65.9%	0%
	SCSI	0%	0%	0%
SABRINE	EXT3	22.4%	16.3%	0%
	ReiserFS	20.0%	32.0%	0%
	SCSI	0%	0%	0%

Table IV provides the distribution of failures observed during the experiments. Both kernel failures and workload failures were observed; instead, no memory allocation failure caused filesystem corruptions, since the kernel tends to crash immediately or to fail gracefully in order to avoid data corruptions. The SCSI target component was robust to memory allocation failures, as it never caused failures: by inspecting its source code, we found that it keeps a pool of previously-allocated data structures (e.g., data transfer command structure) that supply memory when the kernel allocator fails, in order not to lose important disk writes. An important observation is that, in the case of random testing of EXT3, no workload failures occurred. Instead, we noticed that, when focusing error injection in particular states using SABRINE, workload failures were also observed for EXT3 (16.3%): in these cases, the web server failed since the injected error prevented the creation of a temporary file. This kind of failures could occur only under a specific state of the OS, and emphasizes the influence of the OS state on robustness testing.

Stack frame	*Kernel function*
0	`kmem_cache_alloc+0x22/0x110` ← **a failure occurs here**
1	`radix_tree_node_alloc+0x35/0xb0`
2	`radix_tree_insert+0x16e/0x1d0`
3	`add_to_page_cache+0x65/0x1d0`
4	`add_to_page_cache_lru+0x1b/0x40`
5	`mpage_readpages+0x70/0xe0`
6	`ext3_readpages+0x19/0x20` ← **affected EXT3 function**
7	`__do_page_cache_readahead+0x176/0x210`
8	`ondemand_readahead+0xbe/0x170`
9	`page_cache_async_readahead+0x66/0x90`
10	`generic_file_splice_read+0x4a9/0x630`
11	`do_splice_to+0x61/0x80`
12	`splice_direct_to_actor+0x8f/0x180`
13	`do_splice_direct+0x3b/0x60`
14	`do_sendfile+0x187/0x240`
15	`sys_sendfile64+0x77/0xa0`
16	`sysenter_past_esp+0x5f/0x91`

Fig. 8. Call stack of a robustness vulnerability.

It is important to note that Table IV does not provides an indication of the *efficiency* of robustness testing in terms of *unique* robustness vulnerabilities found, as many failures are due to the same vulnerability. By analyzing memory dumps, we concluded that OS crashes were caused by two robustness vulnerabilities in the kernel code. For instance, Figure 8 shows the case of a memory allocation in *radix_tree_node_alloc* that causes the corruption of data structures when the allocation fails and, in turn, the failure of the OS component calling the function. This vulnerability emerges when the file system is retrieving data from the disk to its cache in the main memory when a memory allocation fails, as in the case of cluster 3 in Table III. Table V provides the percentage of random tests able to reveal each robustness vulnerability: this percentage can be very low, as the case of *__get_blk* in ReiserFS (two cases out of 1,000 random tests trigger the vulnerability).

TABLE V
PERCENTAGE OF RANDOM INJECTION TESTS THAT TRIGGER EACH VULNERABILITY.

Vulnerability	EXT3	ReiserFS
__get_blk	29.0%	0.2%
radix_tree_node_alloc	3.8%	9.4%

TABLE VI
PROBABILITY TO REPRODUCE A ROBUSTNESS VULNERABILITY IN SABRINE.

Vulnerability	EXT3	ReiserFS
__get_blk	68.8%	100%
radix_tree_node_alloc	77.7%	100%

In our experiments, SABRINE was able to detect both the two vulnerabilities, with a high efficiency. For each vulnerability, SABRINE generated several test cases able to detect it, by injecting in states where the vulnerability could be triggered. The SABRINE approach identified the same vulnerabilities of random testing, but only a relatively small set of robustness test cases was required to find them (77 test cases in total). Moreover, a vulnerability can be easily reproduced once a test case of SABRINE can detect it. By repeating 10 times the execution of SABRINE test cases, almost every OS crashes repeated identically: Table VI provides the average probability of repeating an OS crash. Instead, it is difficult to reproduce failures using random injections, since the state of the system at the time of the injection plays an important role in triggering vulnerabilities, but it is neglected in random injections. The dramatic reduction of the number of test cases and the ability to easily reproduce OS failures increase significantly the efficiency of robustness testing.

VI. CONCLUSION

In this paper, we proposed and evaluated SABRINE, a state-aware robustness testing approach for OSs. SABRINE can efficiently and automatically generate robustness test cases for distinct states of the OS. The approach does not require the tester to know OS internals, as it infers behavioral models automatically. The overall approach was applied to a Linux-based RTOS used in the avionic domain. For this OS, we tested the robustness of three subsystems against memory allocation failures. Results clearly showed that the state of the OS plays an important role in robustness testing, and that robustness vulnerabilities can be detected with a small number of tests.

ACKNOWLEDGEMENTS

This work was supported by the projects *Embedded Systems in Critical Domains* (CUP B25B09000100007), *SVEVIA* (PON02_00485_3487758), *TENACE* (PRIN n.20103P34XC).

REFERENCES

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

[2] A. Ambrosio, F. Mattiello-Francisco, V. Santiago, W. Silva, and E. Martins. Designing Fault Injection Experiments Using State-Based Model to Test a Space Software. *Lecture Notes in Computer Science*, 4746:170, 2007.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1), Jan. 2002.

[4] J. Arlat, J. Fabre, and M. Rodríguez. Dependability of COTS microkernel-based systems. *Computers, IEEE Transactions on*, 51(2):138–163, 2002.

[5] L. Bairavasundaram, M. Rungta, N. Agrawa, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Swift. Analyzing the effects of disk-pointer corruption. In *Proc. IEEE Intl. Conf. Dependable Systems and Networks*, pages 502–511. IEEE, 2008.

[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. ACM Symp. on Operating Systems Principles*, 2001.

[7] F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

[8] D. Cotroneo, D. Di Leo, R. Natella, and R. Pietrantuono. A Case Study on State-Based Robustness Testing of an Operating System for the Avionic Domain. In *Computer Safety, Reliability, and Security*, pages 213–227. 2011.

[9] D. Cotroneo, R. Natella, and S. Russo. Assessment and Improvement of Hang Detection in the Linux Operating System. In *Proc. Intl. Symp. on Reliable Distributed Systems*, pages 288–294, 2009.

[10] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Proc. Intl. Conf. on Automated Software Engineering*, 2009.

[11] C. Dingman, J. Marshall, and D. Siewiorek. Measuring robustness of a fault tolerant aerospace system. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 522–527. IEEE, 1995.

[12] J. Durães, M. Vieira, and H. Madeira. Multidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior. *IEICE Trans. on Information and Systems*, 86(12):2563–2570, 2003.

[13] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis*. 11th edition, 2006.

[14] J. Fernandez, L. Mounier, and C. Pachon. A model-based approach for robustness testing. *Testing of Comm. Sys.*, pages 313–313, 2005.

[15] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Proc. USENIX Large Installation System Administration Conf.*, pages 101–111, 2006.

[16] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. Trivedi. A methodology for detection and estimation of software aging. In *Proc. Intl. Symp. on Software Reliability Engineering*, pages 283–292, 1998.

[17] A. Ghosh, M. Schmid, and V. Shah. Testing the Robustness of Windows NT Software. In *Proc. Intl. Symp. on Software Reliability Engineering*, pages 231–235, 1998.

[18] W. Gu, Z. Kalbarczyk, R. Iyer, Z. Yang, et al. Characterization of linux kernel behavior under errors. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 459–468, 2003.

[19] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Trans. on Soft. Eng. and Meth.*, 22(1), 2012.

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

[21] IEEE. IEEE Standard for Information Technology–Portable Operating System Interface (POSIX) Part 1. *IEEE Std 1003.1b-1993*, 1994.

[22] B. Jacob, P. Larson, B. Leitao, and S. da Silva. SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems. *IBM Redbook*, 2008.

[23] A. Johansson, N. Suri, and B. Murphy. On the impact of injection triggers for OS robustness evaluation. In *Proc. Intl. Symp. on Software Reliability Engineering*, pages 127–126, 2007.

[24] A. Johansson, N. Suri, and B. Murphy. On the selection of error model(s) for OS robustness evaluation. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 502–511, 2007.

[25] A. Kalakech, K. Kanoun, Y. Crouzet, and J. Arlat. Benchmarking the Dependability of Windows NT4, 2000 and XP. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 681–686, 2004.

[26] K. Kanoun, Y. Crouzet, A. Kalakech, A. Rugina, and P. Rumeau. Benchmarking the Dependability of Windows and Linux Using PostMark Workloads. In *Proc. Intl. Symp. on Fault-Tolerant Comp.*, pages 11–20, 2005.

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

[28] P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Trans. Soft. Eng.*, 26(9):837–848, 2000.

[29] I. Lee and R. Iyer. Software dependability in the Tandem GUARDIAN system. *IEEE Trans. Soft. Eng.*, 21(5):455–467, 1995.

[30] B. Lei, Z. Liu, C. Morisset, and X. Li. State based robustness testing for components. *Electronic Notes in Th. Comp. Sci.*, 260:173–188, 2010.

[31] L. Mariani, F. Pastore, and M. Pezzè. Dynamic analysis for diagnosing integration faults. *IEEE Trans. Soft. Eng.*, 37(4):486–508, 2011.

[32] L. Mariani and M. Pezzè. Dynamic detection of COTS component incompatibility. *Software, IEEE*, 24(5):76–85, 2007.

[33] R. McDougall, J. Mauro, and B. Gregg. *Solaris™ Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Prentice Hall PTR, 2006.

[34] B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[35] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical Report CSTR-95-1268, 1998.

[36] A. Mita. Fault injection capabilities infrastructure. http://www.kernel.org/doc/Documentation/fault-injection/fault-injection.txt.

[37] R. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira. Experimental Risk Assessment and Comparison using Software Fault Injection. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 512–521, 2007.

[38] D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[39] R. Natella and D. Cotroneo. Emulation of Transient Software Faults for Dependability Assessment: A Case Study. In *Proc. Eur. Dependable Computing Conf.*, pages 23–32, 2010.

[40] V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *Proc. Intl. Conf. on Dependable Systems and Networks*, pages 802–811, 2005.

[41] RTCA. DO-178B Software Considerations in Airborne Systems and Equipment Certification. *Requirements and Technical Concepts for Aviation*, 1992.

[42] M. Russinovich and A. Margosis. *Windows® Sysinternals Administrator's Reference*. Microsoft Press, 2011.

[43] C. Sârbu, A. Johansson, N. Suri, and N. Nagappan. Profiling the operational behavior of OS device drivers. *Empirical Software Engineering*, 15(4):380–422, 2010.

[44] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems. In *Proc. Intl. Symp. on Fault-Tolerant Comp.*, pages 2–9, 1991.

[45] U. von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[46] L. Wendehals and A. Orso. Recognizing behavioral patterns at runtime using finite automata. In *Proc. Intl. Wksp. on Dynamic Sys. Analysis*, pages 33–40, 2006.

[47] S. Winter, C. Sârbu, N. Suri, and B. Murphy. The impact of fault models on software robustness evaluations. In *Proc. Intl. Conf. on Software Engineering*, pages 51–60, 2011.