A Case Study on State-Based Robustness Testing of an Operating System for the Avionic Domain

Domenico Cotroneo[1], Domenico Di Leo[1], Roberto Natella[1], Roberto Pietrantuono[1],

[1]Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II,
Via Claudio 21, 80125, Naples, Italy
{cotroneo, domenico.dileo, roberto.natella, roberto.pietrantuono}@unina.it

Abstract. This paper investigates the impact of state on robustness testing, by enhancing the traditional approach with the inclusion of the OS state in test cases definition. We evaluate the relevance of OS state and the effects of the proposed strategy through an experimental campaign on the file system of a Linux-based OS, to be adopted by Finmeccanica for safety-critical systems in the avionic domain. Results show that the OS state plays an important role in testing those corner cases not covered by traditional robustness testing.

Keywords: Robustness Testing, Operating Systems, Safety-Critical Systems, DO-178B, FIN.X-RTOS

1 Introduction

The importance of high robustness in safety-critical systems is well recognized [1][2][3]. The Operating System (OS) is the foundation of any software system, and an OS failure may affect the system as a whole; thus, assessing its robustness is one of the most important tasks to perform during verification of critical software systems. Robustness testing techniques are conceived to assess the system's ability to not fail in presence of invalid inputs and unforeseen conditions.

Due to the kind of bugs for which robustness testing is conceived, and to the complexity and size of OS code, performing an effective robustness testing campaign is challenging. Robustness bugs are characterized by rare and subtle activation conditions, which are hard to find during functional testing [1][2][3]. Unfortunately, in OSs there are so many factors potentially involved in bug activation (such as I/O events and task scheduling), that it is difficult to include all of them when generating robustness test cases. As a result, many OS defects found in the field are related to boundary conditions and error handling, as shown in [3]. This difficulty is further exacerbated by the OS architecture, whose subsystems are tightly coupled, making it hard to isolate the reproduction of rare conditions for each of them. Considering these issues, it is clear that achieving high test coverage in OSs becomes a really tricky task.

In the literature, much effort has been devoted to assess robustness of OSs through the injection of invalid inputs via APIs (i.e., system calls/driver interfaces) with the goal of assessing their robustness. From results of these studies, an important emerging aspect is that, to improve the effectiveness of robustness testing, test cases should consider one more variable, other than exceptional inputs; that is, the current *state* of the OS. Indeed, the OS state can significantly affect its execution, as well as the test case outcomes. Executing a given robustness test case in different states

increases the probability to explore those parts of the code most rarely reached, i.e., it increases the final coverage: states representing "unusual" conditions combined with exceptional inputs can produce very rare execution patterns.

Starting from these studies, this paper investigates the impact of OS state, or part thereof, on robustness testing. We introduce a robustness testing strategy that accounts for the state of the *file system* (in terms of resource usage, concurrent operations, and other aspects). First, a model of the file system is presented that considers both entities a file system is composed of, and resources it uses and that contribute to determine its state. Then, the impact of the state on the achieved coverage is assessed through experiments performed on an industrial case study. Finmeccanica is in the process of developing a certifiable Linux-based OS, namely *FIN.X-RTOS*, compliant to the recommendations of the DO-178B standard [4], that is the reference standard in the avionic domain. In this process, evidences should be provided that the OS underwent a thorough robustness assessment campaign in terms of coverage. Results show that testing the OS by accounting for its state improves the final coverage, and hence the confidence in OS robustness, allowing to reach those corner cases not covered by traditional robustness testing.

After a related work section, the approach is described in Section 3. Section 4 shows obtained results in terms of coverage and via examples of achieved corner cases in the OS code; Section 5 concludes the paper.

2 Related Work

Past work approached robustness testing of operating systems from different points of view; they differ with respect to the OS interface under test (system calls or device driver interface), the assumed fault model, and the failure modes that were analyzed.

BALLISTA [1][2] was the first approach for evaluating and benchmarking the robustness of commercial OSs with respect to the POSIX system call interface [5]. BALLISTA adopts a *data-type based* fault model, that is, it defines a subset of invalid values for every data type encompassed by the POSIX standard. Examples of invalid inputs for three data types are provided in Table 1. Test cases are generated by all the combinations of invalid values of the system call's data types: a test case consists of a small program that invokes the target system call using a combination of input values.

Table 1. **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

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. BALLISTA found several invalid inputs not gracefully handled (Restarts and Aborts), and some Catastrophic failures related to illegal pointer values, numeric overflows, and end-of-file overruns [1].

OS robustness testing evolved in *dependability benchmarks* in the framework of the DBench European project [6][7]. A dependability benchmark has been proposed to assess OS robustness in terms of OS failures, reaction time (i.e., mean time to respond to a system call in presence of faults) and restart time (i.e., mean time to restart the OS after a test). Valid inputs are intercepted and replaced with invalid ones, by using a data-type based fault model, as well as by *fuzzing* (i.e., random values) and *bit-flips* (i.e., a correct input is corrupted by inverting one bit). In dependability benchmarking, the *workload* has a key role: it is used for exercising the system in order to assess its reaction. To obtain realistic measures, the workload should be representative of the expected usage profile (e.g., database or mail server [6][7]). In [8], a stress test campaign on the Linux kernel assessed the influence of the workload on kernel performance and memory consumption over long time periods. In this work, we further investigate the influence of the external environment, and propose a state model for generating tests that includes the OS workload.

Robustness testing has been adopted for assessing OSs with respect to its interfaces to device drivers, since drivers are usually provided by third party developers and are buggier than other OS components [9]. The fault models mentioned above were adopted also in this case [10], and have been compared in terms of their ability to expose robustness bugs [11]. In [12], a fault injection approach is proposed that mutates the device driver code (by artificially inserting bugs) instead of injecting invalid values at the OS interface. These studies found that OSs are more prone to failures in case of device driver faults than application faults, 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 or disk faults), by corrupting code and data [13][14][15][16]. Similarly to system call testing, these approaches either rely on a representative workload for exercising the system, or neglect the system state.

The influence of OS state gained attention in recent work on testing device drivers [17][18]. In [17], the concept of *call blocks* is introduced to model repeating subsequences of OS function calls made by device drivers, since they issue recurring sequences of function calls (e.g., when reading a large amount of data from a device): therefore, robustness testing is more efficient when it is focused on call blocks instead of injecting invalid inputs at random time. Sarbu et *al*. [18] proposed a state model for device driver testing, 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. Case studies on Microsoft Windows OSs found that the test space can be reduced using the state model. Prabhakaran et *al.* [15] proposed an approach for testing journaling file systems, which injects disk faults at specific states of file system transactions. These studies showed that the OS state has an important role in testing such complex systems; however, they model a specific part of OS state (e.g., device drivers or journaling) and do not consider the overall state of the OS components, such as the file system and process scheduling.

3 Testing approach

Since OS components can be very complex and their state has a significant influence on the OS correct behavior, it is necessary to take the states of the Component Under test (CUT) into account, and assess its robustness as the state changes. According to this view, a hypothetical test plan is expressed through two dimensions: the exceptional inputs and the states. Inputs are selected as usual (e.g., boundary values) while the state varies in $S = \{s_1, s_2 \ldots s_n\}$. In order to apply this strategy, we need to test the CUT with both a *test driver* and a *state setter*. The former injects invalid inputs to its interface, whereas the latter is responsible for producing the state transition or keeping the component in a given state s_k (see Figure 1).

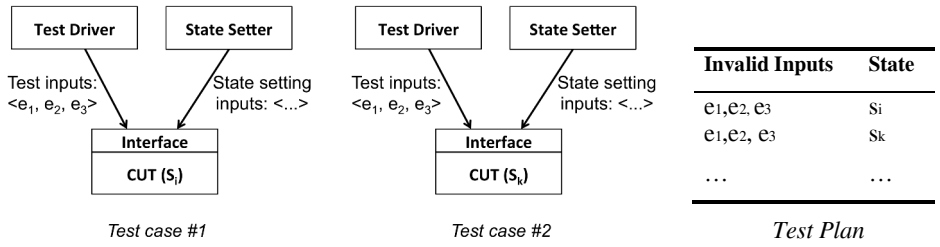

Figure 1 Robustness testing conducted with the CUT in two different states s_i and s_k

In complex components the state representation (i.e., the state model) plays a key role. It can be considered at several levels of abstraction, hence determining the number of potential states the *state setter* should cope with. This aspect is relevant for our approach, since it can affect the efficiency and the feasibility of robustness testing. Thus the state model should satisfy these requirements: *i)* it should be easy to set and control by the tester, *ii)* it should represent the state at a level of abstraction high enough to keep the number of test cases reasonably small and *iii)* it should include those configurations that are the most influential on the component behavior. Thus, with this regard, the model that we define expresses the state of an OS component without detailing its internals, since they are not always easy to understand and to manage, and would inflate the number of states.

3.1 Modeling the File System

In this work, we experiment the described strategy by applying it to the File System (FS) component. We choose the FS because it is a critical and bug-prone component [8][19] (its failure can corrupt persistent data or lead to unrecoverable conditions). Furthermore, the behavior of the FS is influenced by its internal state and the other components with which it interacts (e.g., virtual memory manger, scheduler). Following the previous requirements, we conceived a model for the FS (Figure 2). Moreover, the model is easily adoptable across different FS[1] implementations; as a

[1] In this work, the term "File System" refers to the OS component for managing files. The term "filesystem" refers to the contents on the storage, e.g., the structure of tree.

consequence, the proposed model does not take specific "internal design" of a FS into account (e.g., *inode* that are adopted in some UNIX file system, but not in others).

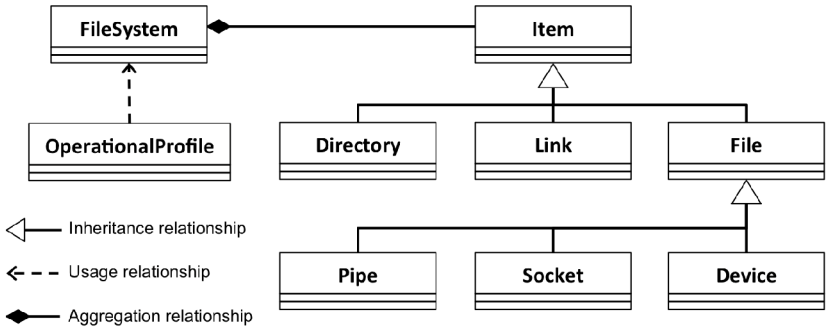

Figure 2 File System model

The model is a UML representation of the FS, with three main classes: `Item`, `FileSystem` and `OperationalProfile`. `FileSystem` represents the contents of data on the disk as a whole. It includes the state attributes that are not specific of a file. The class attributes are reported in Table 2.

Table 2 `FileSystem` attributes

Attributes	Description	Type
Partition Type	Typology of the partition	Primary, Logical
Partition size	Size of the partition on which is installed the FS	Byte
Partition allocated	The Current size of the allocated partition	Byte
Max file size	The maximum dimension of a file on the FS	Byte
Block size	The dimension of a block	Byte
FS implementation	The type of file system	NTFS, ext2, ext3
# of files allocated	The number of files in the FS	Integer
# of directories	The number of directories in the FS	Integer
FS layout	The tree that represents the FS	Balanced, Unbalanced
# of items allocated	The current number of items allocated in the FS	Integer

The choice of attribute values defines the test cases. Attributes like `Partition Allocated` can assume values from a minimum (e.g.,1MB) to the maximum allowable (e.g., 2TB). Therefore, the number of test cases, just for one parameter, grows rapidly. However, test cases in which the values of `Partition Allocated` varies with very small increments (e.g., from 1MB to 2MB) can be of little interest (e.g., 1MB or 2Mb both are values for a small partition). Thus, it is necessary to define criteria to keep the number of test cases reasonably low and cover a reasonable set of test scenarios. Hereafter, we illustrate potential choices for those attributes that the tester can set except for the attributes assigned by OS (e.g., `Max file size`).

The attributes `Block size` and `Partition size` are typically set when the file system is formatted for the first time. In a hypothetical test campaign, these values could assume minimum, maximum and intermediate values. The attribute `Partition allocated` can be expressed as a percentage of `Partition size`,

therefore the tester can set scenarios in which the file system is totally full, partially full or empty.

The attribute `FS layout` deals with the tree representing the directory hierarchy on the FS. In particular, it can assume the values: *balanced*, i.e., trees in which the number of sub-directories is almost the same on each directory, and *unbalanced*, i.e., trees in which the number of sub-directories significantly differs. In order to generate balanced and unbalanced trees, we introduce $P(\{d_{k+1}d_j\})$, i.e., the probability that a new directory, d_{k+1}, is a child of a directory, d_j, already present in the tree. This probability allows, to some extent, to control the structure of the hierarchy, once `Number of Directory allocated` is fixed. For $P(\{d_{k+1}d_j\})$, we provide the following formulas for generating balanced and unbalanced trees, although other choices are possible (e.g., to use a well-known statistical distribution):

$$P_{unbalanced}(\{d_{k+1}d_j\}) = depth(d_j) \Big/ \sum_{1}^{k} depth(d_i) \quad (1)$$

$$P_{balanced}(\{d_{k+1}d_j\}) = \frac{1}{depth(d_j)} \Big/ \sum_{1}^{k} \frac{1}{depth(d_i)} \quad (2)$$

$$ParentDirectory = d : \max\{P(\{d_{k+1}d_1\})...P(\{d_{k+1}d_k\})\} \quad (3)$$

where k is the number of current directories in the tree, and N the number of directories to be created; k is increased until $k=N$. In (1), new directories are more likely to form an unbalanced tree, since the higher the depth of a node is, the higher the probability to have children. In (2), new directories are more likely to group at the same depth. The parent directory (3) is the one with the highest value of $P(\{d_{k+1}d_j\})$.

As for the `FileSystem` class, it is possible to conceive several criteria for assigning values to the attributes. For instance, the attribute `Name` can assume alphabetical and numerical characters with equal probability or the length should not overpass a given value. The attributes `Permission` and `Owner` can be assigned in such a way that a given percentage of files are executable by the owner only, another percentage is readable by all users and so on. The attribute `Size` can be fixed for all files, generated according to a statistical distribution.

The `Item` class represents the entity which a `FileSystem` is made of. For this class, we define typical attributes that are available in every OS. Such attributes are: `name` of the item, `permission` (e.g., readable, writeable, executable), `owner` (root, nobody, user) and `size`. The classes that inherit from `Item` represent the different types of file in a UNIX file system. Files are randomly generated to populate the directory tree mentioned above; the location and type of file can be determined according to statistical distributions.

The FS, like other OS subcomponents, uses resources such as cache, locks and buffers. We refer to these resources as *auxiliary resources*, that is, resources that serve for managing an Item of a FS. For instance, if a thread performs I/O operations it is likely to stimulate auxiliary resources: indeed, buffers are instantiated; locks to control the access to them are used, and so forth. These resources are part of the internal state of the FS, although they are not included in our model, since (i) they cannot be easily controlled by the tester, and (ii) they are dependent on the FS internals. Moreover, most of these resources are instantiated at run-time, and they are

not part of the filesystem on the disk. The presence of these resources, however, cannot be neglected because they may influence the state of the FS and potentially change test outcomes. Therefore, in order to include both the behavior of the auxiliary resources in our model and the manner in which the FS is exercised, we introduce the `OperationalProfile` class. It expresses *the degree of usage* of the auxiliary resources and more generally, the way the FS is stimulated. This class does not directly model the auxiliary resource, but it allows to know the way in which the FS is invoked while performing a test. Thus the tester, *indirectly*, is aware of the mechanisms that are stimulated, e.g., if there are threads invoking I/O operations it is likely that caching and mutex mechanisms are invoked. The `OperationalProfile` attributes are reported in Table 3.

Table 3 `OperationalProfile` attributes

Attributes	Description	Type
Number of tasks invoking FS ops.	Number of tasks that invokes I/O operations (like read, write, open).	Integer
Average number of ops/s	Average number of operations made by a task	Integer
Ratio of read/write ops.	Ratio of read/write operations made by a task	Float

The `OperationalProfile` attributes are related to the performance of the File System and the hardware, which can limit the rate of FS operations that can be served by the system within a reasonable latency. Therefore, the selection of these attributes should be preceded by a capacity test aiming at assessing the maximum operation rate allowed by the system. A capacity test consists in gradually increasing the operations rate, given a fixed number of concurrent tasks (e.g., 2, 4 or 16), until the I/O bandwidth is saturated, i.e., the amount of transferred data per second reaches its peak [14]. After that the I/O bandwidth is known, the tests can select a discrete set of usage levels (e.g., 10% and 90% of I/O bandwidth) and the ratio between read and write operations (e.g., 2 read operations per 1 write operation).

4 Experimentation

In this section, we present an experiment aimed at analyzing the effects of the state on robustness testing, by comparing the proposed approach with a “stateless” approach and with stress testing. The proposed approach has been applied on the FIN.X Real-Time Operating System (RTOS) developed by Finmeccanica. It is a Linux-based OS aimed at industrial applications in the avionic domain. The original Linux kernel has been enhanced by providing hard real-time and scalability on multi-core architectures and removing unessential parts. FIN.X-RTOS is accompanied with documentation and evidences recommended by the DO-178B safety standard [4]. At time of writing the requirements of the standards at level D have been fulfilled (software functions that may cause "a minor failure condition"), and FIN.X-RTOS is currently on the assessment process for the more stringent requirements of level C (software functions that may cause "a major failure condition"), which encompass robustness testing and full statement coverage.

4.1 Experimental Setup

The proposed approach has been applied to the ext3 file system available in FIN.X-RTOS. We selected a set of system calls to test, described in Table 4. The system calls are commonly used by applications and exercise different parts of the FS code.

Table 4 System calls tested

System Call	Description
`access`	check user's permissions for a file
`dup2`	duplicate a file descriptor
`lseek`	reposition read/write file offset
`mkfifo`	make a FIFO special file (a named pipe)
`mmap`	map files or devices into memory
`open`	open and possibly create a file or device
`read`	read from a file descriptor
`unlink`	delete a name and possibly the file it refers to
`write`	write to a file descriptor

To apply the proposed strategy, we selected, without loss of generality, two well-known tools for supporting testing execution, namely *Ballista* and *Filebench*[2]. With regard to Figure 1, *Ballista* plays the role of *test driver*, while *FileBench* is the *state setter*. The *Ballista* tool is currently distributed with the Linux Test Project tool suite. We ported the original version to FIN.X-RTOS. *FileBench* is a tool for FS benchmarking: the user can customize a workload by configuring I/O access patterns in terms of number of threads, access type and so on. In our test campaign, we choose a realistic scenario in which the partition of filesystem is partially full (75% of Partiton size) and there are tasks invoking FS operations, e.g., read and write. Leveraging on the model introduced in section 3, we create a logical partition with a balanced tree and the number of directories is 10 each one populated with 100 small files. No other items have been considered. Table 5 summarizes the values that we selected for the FileSystem entity's attributes. Table 6 shows the values selected for the `File` entities; all the files, apart from `Name`, have the same values. Table 7 specifies the attributes of `OperationalProfile`, which are typical values for FS benchmarking [6][8].

Table 5 `FileSystem` values

Attribute	Value
Partition type	Logical
Partition size	2GB
Partition allocated	1,5GB
Block size	4096
File system implementation	ext3
Number of files allocated	1000
Number of directories allocated	10
Number of items allocated	1010

[2] http://www.ece.cmu.edu/~koopman/ballista/ - http://www.fsl.cs.sunysb.edu/~vass/filebench/

Table 6 `File` values

Attribute	Value
Name	Numeric string with length equals to five
Permission	Readable, Writeable, Executable
Owner	Root
Size	1500Kb

Table 7 `OperationalProfile` values

Attributes	Values
Number of tasks invoking FS operations	16
Average number of operations per second	10
Ratio of read/write operations	1

Those instances of `File`, `FileSystem`, and `OperationalProfile` reproduce stressful conditions in which to test the FS. By stressing the FS with read and write operations on a full allocated partition, we aim at creating exceptional conditions: in fact, with this setting, it is more likely to experiment conditions in which disk blocks are not available, seek operations have to traverse several directories, and so on.

4.1.1 Definition of Test Campaigns

We carry out three experimental campaigns:

1. **Stateless robustness testing.** Ballista injects faults to the selected system calls (Table 4). The faultload to apply to the parameters of the system call belongs to the default *Ballista* configuration. An example of such a faultload is represented in Table 1. This test campaign lasts 15 minutes.
2. **Stress testing.** *FileBench* invokes the system calls `read` and `write` on the files previously allocated for 1 hour. The operations produced by *FileBench* reflect the attributes of `OperationalProfile` (Table 7). *Ballista* is not executed.
3. **Stateful robustness testing.** *FileBench* and *Ballista* work at the same time. *Ballista* and *FileBench* use the same configuration (faultload and operations executed) of the previous campaigns. The entire test campaign lasts 1 hour.

The experimental duration for the first test campaign is the time that *Ballista* spends to execute all the test cases. The second campaign lasts the time necessary for *Ballista* to execute all the tests while *FileBench* is running. The time for the third test campaign is set to 1 hour in order to compare the results between the second and third campaign over the same duration time.

4.2 Results

We first analyze the outcomes of robustness tests, which are classified according to the CRASH scale (see Section 2). Table 8 provides the summary produced by *Ballista*

in the default configuration (i.e., all potential test cases are generated). We did not observe any *Catastrophic* failure, and only a small number of *Restart* and *Abort* failures occurred. This result was expected, since the OS is a mature and well-tested system, and is consistent with past results on POSIX OSs [1], in which only a small number of corner cases led to *Catastrophic* failures (e.g., an OS crash). The relevance of *Restart* and *Abort* failures is a controversial subject, since OS developers tend to consider them as a "robust" behavior of the OS [1]. According to this point of view, we do not consider *Restarts* as severe failures: several OSs (e.g., QNX, Minix) intentionally deal with a misbehaving task by killing it in some specific cases (e.g., manipulation of an invalid memory address, or lack of privileges for performing an operation), in order to avoid further error propagation within the system. Similarly, *Abort* failures can represent an expected (and desirable) behavior of the OS, such as in the case of the `read()` and `write()` system calls that can bring a task in a "waiting for I/O" state. For these reasons, a "Restart" or "Abort" outcome cannot be considered as a "failure" without a detailed analysis of the expected behavior. It should be noted that stateful robustness testing differs from stateless robustness testing with respect to the number of Restart outcomes, mostly due to failed memory and disk allocations. Although we cannot conclude that these outcomes represent OS failures, this result points out that OS state can affect test outcomes and the assessment of OS robustness.

Table 8 Results of robustness tests.

Function	# Tests	Stateless robustness testing		Stateful robustness testing	
		# Restart	# Abort	# Restart	# Abort
`access()`	3,986	0	4	1	4
`dup2()`	3,954	0	0	1	0
`lseek()`	3,977	0	0	0	0
`mkfifo()`	3,870	0	5	1	5
`mmap()`	4,003	0	0	0	0
`open()`	3,988	0	8	40	8
`read()`	3,924	0	253	1	253
`unlink()`	500	0	1	0	1
`write()`	3,989	0	68	4	68
Total	32,191	0	339	48	339

However, the stateful tests cover a scenario not considered by stateless tests, and therefore they represent an additional evidence of the robust behavior of the OS. As a result, we observed an increased coverage of kernel code after executing the stateful tests; this aspect is relevant since coverage is a measure of test confidence and a requirement for software in safety-critical systems (e.g., DO-178B at level C [4]).

We analyzed statement coverage of file system code, which is the target of our tests. The file system code is arranged in three directories: the code in the "fs/" directory is independent from the specific file system implementation (i.e., it is shared among several implementations such as ext3 and NTFS); the "ext3" directory provides the implementation of the ext3 file system; finally, the "jbd" directory provides a generic support for journaling file systems. Data about coverage was collected using GCOV. Table 9 compares the statement coverage with respect to the three considered scenarios. We observed differences in coverage between stateless (second column) and stateful robustness testing (fourth column), ranging between 0.49% and 15.11%. Part of the code is covered by the plain state setter (i.e., without

using Ballista); the remaining part is covered due to interactions between Ballista and the OS state (some examples are provided in the following). In particular, stateful testing exercised those parts of the file system that interact with other subsystems (e.g., interactions between "fs/buffer.c" and the memory management subsystem, and between "fs/fs-writeback.c" and disk device drivers). The coverage improvement is more significant for the journal-related code (i.e., the JBD component in “fs/jbd”). This effect can be attributed to the interactions between file system transactions and the state of I/O queues. For instance, a transaction commit can be delayed due to concurrent I/O operations, therefore affecting the management of data buffers within the kernel and the file system image on the disk. Although the improvement is less significant for the implementation-independent code, the proposed approach has been useful for improving test coverage with no human effort. This aspect is relevant since FIN.X-RTOS is mostly composed by third-party code re-used from the Linux kernel; covering this code can be very costly, due to the lack of knowledge of kernel internals and the inherent complexity of OS code (e.g., heuristics for memory management).

Table 9 Statement coverage.

Source file	Stateless robustness testing	Stress testing	Stateful robustness testing
fs/binfmt_elf.c	319/850 (37.53%)	331/850 (38.94%)	332/850 (39.06%)
fs/buffer.c	529/1320 (40.08%)	553/1320 (41.89%)	565/1320 (42.80%)
fs/dcache.c	371/880 (42.16%)	341/880 (38.75%)	387/880 (43.98%)
fs/exec.c	479/807 (59.36%)	392/807 (48.57%)	486/807 (60.22%)
fs/fs-writeback.c	146/273 (53.48%)	169/273 (61.90%)	174/273 (63.74%)
fs/inode.c	252/527 (47.82%)	307/527 (58.25%)	316/527 (59.96%)
fs/namei.c	918/1392 (65.95%)	626/1392 (44.97%)	925/1392 (66.45%)
fs/select.c	237/402 (58.96%)	237/402 (58.96%)	239/402 (59.45%)
fs/ext3/balloc.c	384/556 (69.06%)	385/556 (69.24%)	398/556 (71.58%)
fs/ext3/dir.c	140/219 (63.93%)	143/219 (65.30%)	144/219 (65.75%)
fs/ext3/ialloc.c	181/337 (53.71%)	186/337 (55.19%)	189/337 (56.08%)
fs/ext3/inode.c	719/1204 (59.72%)	729/1204 (60.55%)	737/1204 (61.21%)
fs/ext3/namei.c	607/1088 (55.79%)	654/1088 (60.11%)	781/1088 (71.78%)
fs/jbd/checkpoint.c	102/263 (38.78%)	141/263 (53.61%)	142/263 (53.99%)
fs/jbd/commit.c	300/362 (82.87%)	302/362 (83.43%)	318/362 (87.85%)
fs/jbd/revoke.c	108/228 (47.37%)	105/228 (46.05%)	116/228 (50.87%)
fs/jbd/transaction.c	489/697 (70.16%)	500/697 (71.74%)	545/697 (78.19%)

In order to better understand the interactions between OS state and test cases, we analyzed more in depth part of the kernel code only covered by stateful robustness testing. Figure 3 shows an example of corner case in the kernel code not covered in stateless testing (the code is highlighted in bold font; part of the code was omitted; we kept some comments from developers). The `real_lookup()` routine is invoked when file metadata are not in the page cache, and the FS needs to access to the disk. It blocks the current task on a semaphore (using the `mutex_lock()` primitive) until a given directory can be accessed in mutual exclusion. It then checks if metadata have been added to the cache during this wait period. Usually, metadata are not found, and the routine performs an access to the disk. In stateful testing, a different behavior was observed, since the cache has been re-populated during the wait period (developers refer to this situation as "nasty case"), and additional operations are executed (e.g., to check that metadata are not expired due to a timeout in distributed file systems). This

code was only executed in stateful testing due to interactions with the cache that occur when concurrent I/O operations are taking place.

```
static struct dentry * real_lookup(struct dentry * parent,
        struct qstr * name, struct nameidata *nd) {
    /* --- OMISSIS (declarations) --- */
    mutex_lock(&dir->i_mutex);
    result = d_lookup(parent, name);
    if (!result) {
        /* --- OMISSIS (performs lookup) --- */
        mutex_unlock(&dir->i_mutex);
        return result;
    }
    /* Uhhuh! Nasty case: the cache was re-populated while
     we waited on the semaphore. Need to revalidate.*/
    mutex_unlock(&dir->i_mutex);
    if (result->d_op && result->d_op->d_revalidate) {
        result = do_revalidate(result, nd);
        if (!result)
            result = ERR_PTR(-ENOENT);
    }
    return result;
}
```

Figure 3 Example of kernel code covered due to interactions between the file system and caching (from `real_lookup()`, fs/namei.c:478).

Another example is provided in Figure 4, which is related to concurrency of kernel code. The `ll_rw_block()` routine performs several low-level accesses to the disk, and each access is controlled by a "buffer head" data structure. During the inspection of the list of buffer heads, one of them could have been locked by another concurrent task; this condition is detected by the `test_set_buffer_locked()` primitive, which may fail to lock the buffer head in some cases. Stateful testing covered this rare scenario, and it is worth being tested to verify that pending I/O is correctly managed.

```
void ll_rw_block(int rw, int nr, struct buffer_head *bhs[]) {
    int i;
    for (i = 0; i < nr; i++) {
        struct buffer_head *bh = bhs[i];
        if (rw == SWRITE)
            lock_buffer(bh);
        else if (test_set_buffer_locked(bh))
            continue;
        /* --- OMISSIS (performs I/O op.) --- */
}
```

Figure 4 Example of kernel code covered due to concurrent I/O requests (from `ll_rw_block()`, fs/buffer.c:2941).

Finally, we analyzed an example of kernel code interacting with memory management, which is provided in Figure 5. The `try_to_free_buffers()` routine is invoked by the file system when the cache for file system data (the "page cache") gets large and pages need to be freed for incoming data. It may occur that a file system transaction involves I/O buffers allocated over several pages, and these pages cannot be de-allocated until the transaction commits. Pages are then marked with "`mapping == NULL`" in order to be reclaimed later (the `drop_buffers()` routine

checks that I/O buffers in the page are not being used). As suggested by the comment in the code, this condition is unlikely to occur; the code has been executed in stateful testing since memory management has been put under stress.

```
int try_to_free_buffers(struct page *page) {
    /* --- OMISSIS (declarations) --- */
    BUG_ON(!PageLocked(page));
    if (PageWriteback(page))
        return 0;
    if (mapping == NULL) {      /* can this still happen? */
        ret = drop_buffers(page, &buffers_to_free);
        goto out;
    }
    /* --- OMISSIS (page writeback and deallocation) --- */
}
```

Figure 5 Example of kernel code covered due interactions between the file system and memory management (from `try_to_free_buffers()`, fs/buffer.c:3057).

5 Conclusion and Future Work

This paper investigated the impact of OS state on robustness testing through an experiment on the File System of a Linux-based OS for critical applications. In order to include the OS state in the robustness test plan, we introduced a model of the File System by including a set of factors (such as file tree layout and concurrent I/O operations) that are most influential on the File System behavior, and that can be controlled by the tester. We performed an experiment using the proposed model, which highlighted the influence OS state on the test outcomes and on statement coverage. In particular, robustness tests were able to reach corner cases with complex interactions with other subsystems (such as scheduling, caching and memory management), which are not covered by traditional robustness testing. In turn, this approach comes in handy to achieve an increased confidence in OS robustness with low human effort, since both robustness test cases and OS states can be automatically generated once programmed by the tester.

Future work encompasses an experimental campaign with more robustness tests and OS states, in order to assess the full potential of robustness testing. Moreover, we plan to analyze test planning strategies in order to achieve the best trade-off between time and the code coverage or the explored states. Another direction is to extend the approach to other subsystems. For instance, a model similar to the FS could be introduced for the virtual memory manager, by including the amount and type of memory areas allocated by processes, physical free memory, swap usage and so on.

Acknowledgements. We would like to thank Mariana Esposito for her valuable contributions, and Francesco Rogo and MBDA Systems for their technical support with FIN.X-RTOS. This work has been funded by the FP7 European project CRITICAL-STEP (http://www.critical-step.eu) IAPP no. 230672, and by the Italian research project “Iniziativa Software”, which involves the Finmeccanica company and Italian universities (http://www.iniziativasoftware.it).

References

1. Koopman, P. and DeVale, J., "The exception handling effectiveness of POSIX operating systems," IEEE Trans. on Software Engineering, vol. 26, no. 9, 2002
2. Koopman, P. and Sung, J. and Dingman, C. and Siewiorek, D. and Marz, T., "Comparing operating systems using robustness benchmarks," SRDS 1997
3. Sullivan, M. and Chillarege, R., "Software Defects and their Impact on System Availability-A Study of Field Failures in Operating Systems," FTCS 1991
4. RTCA Inc., "Software considerations in airborne systems and equipment certification," RTCA DO-178B, EUROCAEED-12B, 1992
5. IEEE Standard for Information Technology-Portable Operating System Interface (POSIX). IEEE Std 1003.1b-1993, IEEE CS, 1994
6. Kanoun, K. and Crouzet, Y. and Kalakech, A. and Rugina, A.-E. and Rumeau, P., "Benchmarking the Dependability of Windows and Linux using PostMark™ Workloads," ISSRE 2005
7. Kalakech, A. and Kanoun, K. and Crouzet, Y. and Arlat, J., "Benchmarking The Dependability of Windows NT4, 2000 and XP," DSN 2004
8. Cotroneo, D. and Natella, R. and Pietrantuono, R. and Russo, S., "Software Aging Analysis of the Linux Operating System," ISSRE 2010
9. Chou, A. and Yang, J. and Chelf, B. and Hallem, S. and Engler, D., "An empirical study of operating systems errors," SOSP 2001
10. Albinet, A. and Arlat, J. and Fabre, J.C., "Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel," DSN 2004
11. Johansson, A. and Suri, N. and Murphy, B., "On the selection of error model(s) for OS robustness evaluation," DSN 2007
12. Duraes, J. and Madeira, H., "Multidimensional characterization of the impact of faulty drivers on the operating systems behavior," IEICE Trans. on Information and Systems, vol. 86, no. 12, 2003
13. Gu, W. and Kalbarczyk, Z. and Iyer, R.K. and Yang, Z., "Characterization of Linux kernel behavior under errors," DSN 2003
14. Skarin, D. and Barbosa, R. and Karlsson, J., " GOOFI-2: A tool for experimental dependability assessment," DSN 2010
15. Bairavasundaram, L.N. and Rungta, M. and Agrawa, N. and Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. and Swift, M.M., "Analyzing the effects of disk-pointer corruption," DSN 2008
16. Dreges, R. J. and Nanya, T. "Analysis of Inter-Module Error Propagation Paths in Monolithic Operating System Kernels," EDCC 2010
17. Johansson, A. and Suri, N. and Murphy, B., "On the impact of injection triggers for OS robustness evaluation," ISSRE 2007
18. Sarbu, C. and Johansson, A. and Suri, N. and Nagappan, N., "Profiling the operational behavior of OS device drivers" Empirical Soft Eng, vol. 15, no. 4, 2009
19. Prabhakaran, V. and Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H., "Model-based failure analysis of journaling file systems," DSN 2005