Separation Kernel Robustness Testing

The XtratuM Case Study

S.Grixti, N.Sammut
Faculty of Information and Communications Technology
University of Malta
Msida, Malta
stephen.grixti@alumni.um.edu.mt
nicholas.sammut@um.edu.mt

M.Hernek
Flight Software System Section
European Space Agency
Noordwijk, Netherlands
maria.hernek@esa.int

E.Carrascosa, M.Masmano, A.Crespo
Instituto de Automática e Informática Industrial
Universidad Politécnica de Valencia,
Valencia, Spain
{ecarrascosa, mmasmano, acrespo}@ai2.upv.es

Abstract— With time and space partitioned architectures becoming increasingly appealing to the European space sector, the dependability of separation kernel technology is a key factor to its applicability in European Space Agency projects. This paper explores the potential of the data type fault model, which injects faults through the Application Program Interface, in separation kernel robustness testing. This fault injection methodology has been tailored to investigate its relevance in uncovering vulnerabilities within separation kernels and potentially contributing towards fault removal campaigns within this domain. This is demonstrated through a robustness testing case study of the XtratuM separation kernel for SPARC LEON3 processors. The robustness campaign exposed a number of vulnerabilities in XtratuM, exhibiting the potential benefits of using such a methodology for the robustness assessment of separation kernels.

Keywords: fault injection; robustness testing; separation kernel; data type fault model; XtratuM.

I. Introduction

Worldwide telecommunications, weather prediction, navigation and remote sensing are only a few space applications on which today's society relies daily. Such a dependency is projected onto increasingly demanding spacecraft functional requirements, usually leading to more complex software implementations. A recent NASA study, which uses the number of uncommented source lines as a complexity metric, claims a 200-fold increase in complexity over 20 years, from Apollo to the initial ISS missions [1]. The extensive complexity of integrating varied criticality software has been pushing the space industry to using federated system architectures, where each subsystem has its own computing node in the avionics suite. However, with an undesirable increase in power consumption, mass and volume, the European Space Agency (ESA) has been recently considering moving to Integrated Modular Avionics (IMA) architectures, which have been extensively used across the aviation industry since the 1990s [2].

The main aim of IMA is allowing different criticality software to share the same node while conserving the inherently available advantages of federated systems, namely fault containment and separation of concerns. This is realised through Time and Space Partitioning (TSP), which partitions the software in the temporal and spatial domains [3]. This separation is achieved through adding a layer of software referred to as a separation kernel, which enforces the segregation and guarantees all applications are bound by the predefined temporal and spatial rules. This subdivides the software into logical containers or partitions, and allocates temporal and spatial resources to meet the requirements of all applications. The two main pillars of TSP may be summarized as follows:

- Temporal Partitioning: At a particular point in time a software partition has the sole control over the onboard computer (OBC), during which it is not interfered by other applications. This is achieved through creating a cyclic schedule during which every application has an execution slot.

- Spatial Partitioning: Every software partition is guaranteed to be free from unintended modification of data in the spatial resources assigned to it. This includes assigning memory areas to software applications and making sure other resources such as I/O are free from competition during the partition's execution slot.

As opposed to federated architectures, IMA requires fewer hardware components, potentially reducing cost and improving reliability. Furthermore, integrated architectures favor the integration of Commercial Off-The-Shelf (COTS) software in order to reduce development and maintenance costs. [4] IMA is thus a very promising technology within the Space sector. However, with various real-time embedded Space systems requiring high assurance levels, the applicability of such architectures is only a prospect if applications can be guaranteed fault containment through robust partitioning mechanisms [5]. It is thus implied that the dependability of a TSP system is tightly coupled to the robustness of the separation kernel and its reliability in preventing propagation of faults between applications of different criticality.

While there exist various means to improving separation kernel dependability, such as designing for fault prevention and tolerance, testing campaigns aimed at fault

removal play a reasonable role during development [6]. One subset of such dynamic verification techniques is robustness testing, which injects faults through exercising the code with invalid test inputs. Such testing campaigns are aimed at verifying fault tolerance and uncovering robustness vulnerabilities.

This paper explores the applicability of the data type fault model in separation kernel robustness testing. This approach, which injects faults through the API, has been tailored to study its relevance in uncovering vulnerabilities within separation kernels and potentially contributing towards fault removal campaigns. Having introduced TSP, Section II outlines the pillars of separation kernel dependability. Section III provides details of the implementation while Section IV exhibits the potential of such a methodology through the XtratuM Robustness testing case study. Discussion and a number of possible improvements are then provided in Section V.

II. Separation Kernel Dependability

Ensuring that onboard real-time embedded systems hosted on a single OBC are provided with a dependable infrastructure involves various components within the separation kernel. The following are a few of the important constituents that help guarantee fault containment between applications of varying criticality.

- Intricate resource allocation: Resource sharing in a safe and reliable manner can be guaranteed if allocated assets, such as I/O registers, devices, memory and OBC slot-times, are defined in detail for each partition.
- Robust temporal and spatial isolation: The separation kernel runs in supervisor processor mode and has absolute control on the status of each partition. In the event that a partition process attempts to overshoot its timeslot, or write data to a location that is not assigned to it, it is to be flagged by the separation kernel. Being a TSP infringement, the process is halted and the event noted. The separation kernel can also halt or reset a partition if it is repetitively violating TSP boundaries.
- Robust Inter-Partition-Communication (IPC): While functional dependency between partitions is usually kept to a minimum, transfer of data between applications is often necessary. This is done through IPC channels strictly defined by the separation kernel so as to limit propagation of faults between partitions. ault monitor and handling mechanism: This mechanism is responsible of detecting and handling irregular events occurring within partitions or the kernel itself. The main objective is to discover the errors as early as possible so that offending processes or partitions are dealt with and the faults contained [7].

Since such mechanisms are critical to IMA dependability, analyzing their robustness will be of particular interest. Using the data type model, faults may be introduced into each of these constituents through hypercalls implementing related services.

III. Robustness Testing Framework

Robustness is a related concept to the notion of dependability. It is defined as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [8]. While a contributor to system dependability, robustness is more concerned with cases where external components behave unexpectedly and feed invalid inputs to the system. This may happen either because of a fault within external components or simply because of an abnormal combination or sequence of events resulting in an irregular system call. An example of the latter would be a system composed of several fault-free components yet a robustness failure still occurs if the correct output from one component is invalid to the next component [9].

Robustness testing thus, involves test data that necessitates faults and is hence applicable to verifying the fault tolerance of the software package. In this case, the testing methodology is usually referred to as *fault injection*. [6] The flowchart in Fig. 1 outlines the methodology employed in executing a fault injection campaign for separation kernels. This is comprised of three major phases:

- Preparation
- Test Generation and Execution
- Log Analysis

A. Preparation Phase

The preparatory phase, which deals with outlining the testing scope and defining the fault model and test suits accordingly, is of particular importance and thus requires considerable effort. Having defined the scope as a fault removal campaign for separation kernels, the next step is devising the system fault model. This is exercised through software fault injection and raises three main questions [10]:

- What faults to inject?
- Where to inject the faults?
- When to inject the faults?

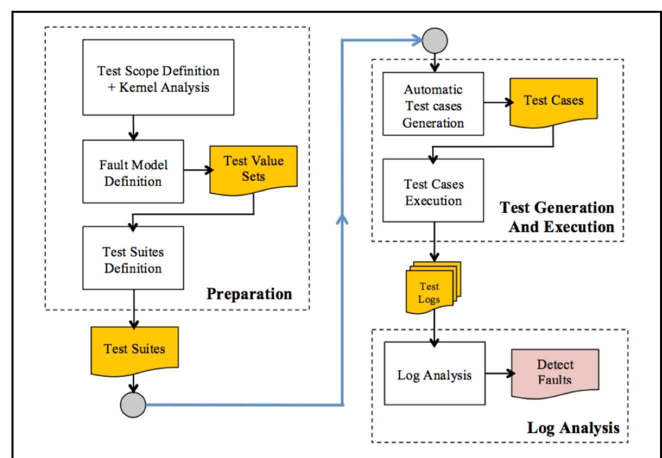

Figure 1. Robustness testing top-level methodolgy

It is thus clear that as a prerequisite to defining a fault model, a good understanding of the system architecture under test is an asset [11]. This helps identify mechanisms whose robustness is instrumental for system dependability and consequently significant in defining a test campaign that ensures proper coverage.

Based on its usefulness in both research and industrial projects, the data type fault model has been selected. In particular, the Ballista project [12], and the RTEMS robustness testing campaign by *Critical Software SA* [13], have shown that this fault model offers potential in provoking numerous and diverse vulnerabilities. The data type fault model is a black-box methodology that defines test cases from the data types of the parameters passed to kernel system calls. This is done by defining a set of test values for each data type, which is likely to contain exceptional values for functions [9]. This “dictionary” of interesting values is composed of values suggested in testing literature [14] and values that uncovered issues in previous tests. Since the greater majority of device drivers are coded in ANSI C [10], C-style data types are usually considered. As an example, possible test values for a signed integer data type would be: -1, 0, 1, MIN_INT and MAX_INT. For multi-parameter calls, test cases are created by using a combination of values across all the parameters of the function. A combination of valid and invalid test data helps unmask any vulnerable parameters and trace back to the parameter responsible for the failure.

B. Test Generation and Execution Phase

The methodology involves the use of an IMA testbed with dummy partitions defined by the separation kernel under test. One of these partitions will contain fault placeholders, which for the scope of this paper will be referred to as the *test partition*. Using the data type model, these fault placeholders take the form of separation kernel hypercalls with a test dataset comprised of valid and invalid parameters. Such fault placeholders are generated through two XML files that define kernel-specific test information; a technique that has been previously proposed for the *Xception* Toolset by *Critical Software SA* [13]. The *API Header* XML file lists all hypercalls and parameter data types for the separation kernel in consideration, while the *Data Type* XML lists test values associated with each data type. The excerpts shown in Fig. 2 and Fig. 3 are examples of a three-parameter hypercall header and the test value set for an unsigned integer type, respectively. Both examples are reproduced from the XtratuM case study, which is discussed in the next section.

```
<Function Name="XM_reset_partition" ReturnType="xm_s32_t" IsPointer="NO">
  <ParametersList>
    <Parameter Name="partitionId" Type="xm_s32_t" IsPointer="NO"/>
    <Parameter Name="resetMode" Type="xm_u32_t" IsPointer="NO"/>
    <Parameter Name="status" Type="xm_u32_t" IsPointer="NO" />
  </ParametersList>
</Function>
```

Figure 2. Example of an API header (XtratuM case study)

```
<DataType Name="xm_u32_t">
  <BasicType>unsigned int</BasicType>
  <TestValues>
      <Value>0</Value>
      <Value>1</Value>
      <Value>2</Value>
      <Value>16</Value>
      <Value>4294967295</Value>
  </TestValues>
</DataType >
```

Figure 3. Example of a data type test value set (XtratuM case study)

Fig. 4 shows the flow logic to generate the test partition containing the fault placeholders. The following steps outline the fault injection process for a particular hypercall.

1. The hypercall to be used as a fault placeholder is provided to the toolset. This may be provided automatically as part of a test campaign or selected by the user as required.
2. Through the use of the two kernel-specific XML files, the toolset generates all the possible combinations considering the test data over each parameter.
3. A source file containing one test hypercall is generated and compiled with the test partition sources. This generates the test partition executable.
4. The test partition is ‘packed’ with the rest of the partitions and the TSP system is run on the target-system simulator for a selected number of cyclic schedules. The test call is invoked at least once per major frame.
5. In each case the return code to the test hypercall together with partition and separation kernel health specifics are logged for analysis at a later stage.
6. Steps 2 to 5 are repeated until all test combinations have been run and results logged. The test setup runs on a UNIX-based operating system and a number of shell scripts ensure a whole testing campaign, comprising multiple hypercall test suits and the accompanying logs, can be completed automatically with no intervention required from the test administrator.

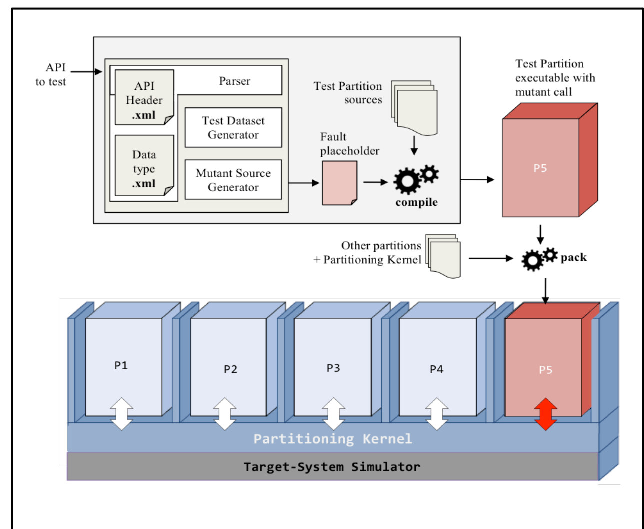

Figure 4. Methodology to generating the test partition

The flowchart in Fig. 5 elaborates on steps 1 to 3, and provides a more detailed understanding of the toolset generating the source file containing the fault placeholders for a particular hypercall.

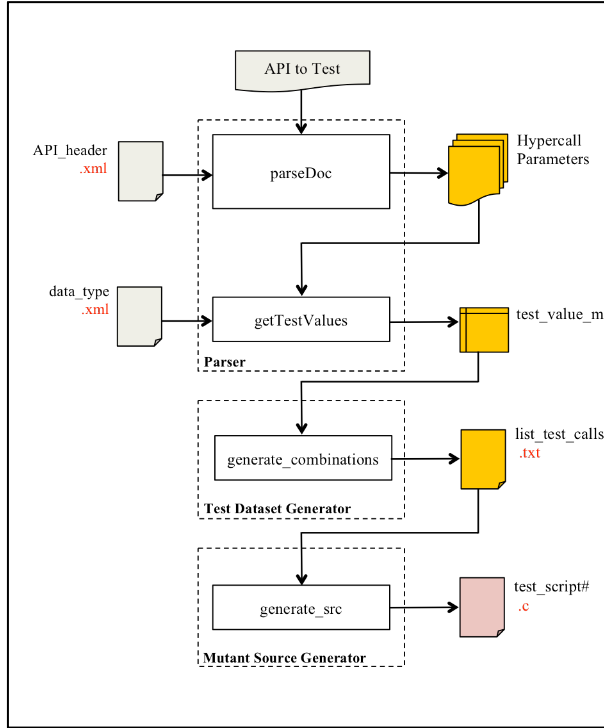

Figure 5. Generation of the mutant source

Each source file contains a fault placeholder in the form of one hypercall invoked with a test dataset. The toolset can be subdivided into three main sections:

- XML Parser: The front-end function parses the hypercall selected for the test suite and, using the preconfigured *API Header* XML file, outputs the hypercalls input parameters together with their data types. The next stage of the parser then generates the *test_value_matrix* according to the *Data Type* XML file. This matrix contains the test values associated with each input parameter for the respective hypercall.

- Test Dataset Generator: This toolset generates a list of datasets with all possible test value combinations found in the *test_value_matrix*. The total number of combinations is equivalent to

$$\text{combinations}_{\text{total}} = \prod_{X=0}^{N} n_{PX} \qquad (1)$$

Where n_{PX} is the number of test values for input parameter *PX* and *N* is the total number of parameter inputs.

- Mutant Source Generator: This stage incorporates each test dataset within a single source file, which is then compiled to form part of the *test partition* executable.

C. Log Analysis Phase

This phase emphasizes the importance of proper product analysis during the preparation phase of the test campaign. A compromise is to be found between logging too much, leading to a lengthier analysis phase, or logging too little and potentially missing out on identifying certain robustness weaknesses. During each test execution, the following are monitored and logged:

- Return Codes, which classify the immediate response to the test call and help categorize unexpected responses.
- Exception handlers, which identify any non-nominal partition or kernel behaviour.
- Partition and separation kernel statuses, which pinpoint *Halt* or *Reset* events on both partition and kernel levels.
- Operations undertaken by the Fault Monitoring and Handling mechanism of the separation kernel. This helps track any actions leading to fault containment.

Since robustness evaluation relies on failures identified by the health monitor of the kernel under test, a good understanding of the latter is an asset to appropriately configure logging. As identified through the kernel reference architecture [15], logging such data associated with each test call is sufficient to classify the outcome of each dataset. However, since tests are independent of hypercall logic, the means to defining a test case pass/fail is also generic. In fact, the Ballista project categorizes test results according to the CRASH (Catastrophic, Restart, Abort, Silent, Hindering) severity scale [9] and applies the following properties for every module under test:

- **C**atastrophic - A test should never crash the system: violation of this property implies a corruption of the kernel's state and is hence considered a *catastrophic* failure.
- **R**estart - A test should never hang: this is considered as a *restart* failure because a restart of the task that stopped responding is required for system recovery.
- **A**bort - A test should never crash the testing task: *abort* failures cause an irregular task termination causing a core dump.
- **S**ilent - A test should always report exceptional situations: when a reportable exception is not indicated, this is considered a *silent* failure.
- **H**indering - A test should never report incorrect error codes: a *hindering* failure occurs when an incorrect error code is reported. As with the previous case, additional test data analysis is required to identify this kind of fault.

Catastrophic, Restart and *Abort* failures are nominally flagged through the kernel fault handling mechanism and will also be apparent in partition or kernel health statuses. Furthermore, a test case that fails to return is potentially indicative of a *catastrophic*, *restart* or *aborted* failure that has not been reported by the fault monitor. Conversely, identification of *Silent* and *Hindering* failures is usually only practicable through manually crosschecking returned codes against reference documentation. The creation of an oracle that can predict if a test case is to generate an exception or not is usually considered impractical [12]. While classified into five distinct categories, it is to be mentioned that all of these failures come at different levels of criticality. As an example, a fault occurring within a partition-control module is more significant and has potentially more impact than a fault within a module that seeks through fault monitor events.

Having provided a good overview of the robustness testing toolset that has been developed, its potential is now demonstrated through a case study. The next chapter details the specifics behind the robustness testing of the XtratuM separation kernel and how the developed toolset was instrumental in uncovering a number of robustness failures.

IV. Case Study: XtratuM for LEON3

This section provides an overview of how this methodology was used in a robustness testing campaign for XtratuM (XM) for SPARC LEON3. For the scope of this study, rather than dummy partitions, ESA's EagleEye TSP was used as a testbed. EagleEye TSP is an ESA reference spacecraft mission representative of a typical earth observation satellite. Its main purpose is the validation of functional and real-time properties of new technologies [15], and is thus considered very appropriate for the scope of this project. This platform consists of a LEON3 central node with a memory management unit, simulated using TSIM from *Aeroflex Gaisler*. It runs XM as a separation kernel defining the OBSW into five partitions over a cyclic major frame of 250ms. Fig. 6 outlines the test setup utilizing the EagleEye TSP spacecraft.

Before going into the details of the test campaign, an overall understanding of the separation kernel is deemed necessary.

A. Components of the XtratuM Separation Kernel

Considered as a bare-metal hypervisor, XM is a layer of software that provides one or more virtual execution environments for partitions in highly critical systems. The internal XM architecture includes components [4] such as

- Memory Management (Spatial Separation)
- Scheduling (Temporal Separation)
- Interrupt management
- Clock / Timer management
- Inter-Partition Communication (IPC)
- Health Monitor (HM)
- Tracing facilities

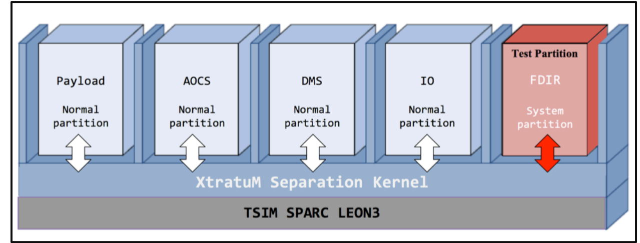

Figure 6. XM test setup using the EagleEye TSP spacecraft

XM defines two levels of partitions: normal and system. The latter have added privileges, namely managing and monitoring the state of the system and other partitions. Such services are provided through hypercalls that can only succeed if invoked from a system partition. As shown in Fig. 6, the FDIR is the only system partition within the EagleEye TSP spacecraft. The added privileges make it an ideal candidate for a test partition. Within each of the partitions created by XM then resides an operating system (OS) that locally handles partition-scope tasks. Examples of such OSes supported by XM are the RTOS RTEMS for multi-threaded *C* applications and the XtratuM Abstraction Layer (XAL) as a single threaded *C* runtime. XM also supports the Open Ravenscar Kernel (ORK) for running partitions implemented in the Ada programming language. [7]

B. Data Types and Test Data Selection

The data types used in XM interfaces are compiler and machine cross development independent [7]. Table I lists all XM data types together with the declarations in C-language

As referred to in the previous chapter, the test calls are generated through a class of test values that are data type bound. Such values are appropriately selected to exercise the error handling and robustness capabilities of the kernel code. These are typically boundary or "magic values" in the data type range [13]. Test datasets are key to the reliability and confidence in the robustness testing results and are to be chosen only after a clear understanding of the testing scope.

TABLE I. XtratuM Data Types

XM Basic Types	XM Extended Types	Size (bits)	ANSI C Types
xm_u8_t	-	8	unsigned char
xm_s8_t	-		signed char
xm_u16_t	-	16	unsigned short
xm_s16_t	-		signed short
xm_u32_t	xmWord_t xmAddress_t xmIoAddress_t xmSize_t xmId_t	32	unsigned int
xm_s32_t	xmSSize_t		signed int
xm_u64_t	-	64	unsigned long long
xm_s64_t	xmTime_t		signed long long

TABLE II. DATA TYPE TEST-VALUE-SET EXAMPLE

XM Data type	Data Type Range	Test Data	Description
xm_s32_t	$-2^{31} \rightarrow 2^{31}-1$	-2147483648	MIN_S32
		-16	*
		-1	
		0	ZERO
		1	*
		2	
		16	
		2147483647	MAX_S32

*valid / invalid input depending on hypercall

Table II provides an example of test values corresponding to the xm_s32_t type, which is a standard C-type signed integer. It is to be noted that these test values are not supposed to represent all the values that may be interesting to test with the given data types [13]. They were chosen to provide a reasonable range of non-nominal input conditions while keeping the test campaign practically manageable. For the scope of this case study, the selected test values used are the ones suggested in testing literature [17], [14], and values that uncovered issues in previous tests. The Ballista project [12] has been a prime source in this regard.

It is to be noticed that the example in Table II includes test values that are neither boundary nor “magic values” for the particular data type. Marked with an asterisk in Table II, such values can potentially be valid for certain parameters in the hypercalls under test. The objective is to avoid fault masking while testing. In hypercalls with more than one input parameter, masking can occur if parameter validity checks are done on one parameter and not the others. Consider the cases in Fig. 7, where *hypercall_1* incorporates validity checks only on the first parameter. As with *Case 1*, since the first parameter is invalid, the kernel returns an error code, which is considered as robust behavior. In the event *Case 2* results in a non-robust behavior (e.g. unexpected termination), the invalid first parameter in *Case 1* is said to mask a second-parameter robustness failure. [9]

Case 1: *hypercall_1*(< *invalid* >, < ■ >) → Robust

Case 2: *hypercall_1*(< *valid* >, < *invalid* >) → Non-robust

Figure 7. Example of fault masking

TABLE III. XTRATUM TEST CAMPAIGN

Hypercall Category [16]	Total Hypercalls [16]	Hypercalls tested	No. of Tests	Raised Issues
System Management	3	2	8	3
Partition Management	10	6	236	0
Time Management	2	2	34	3
Plan Management	2	1	2	0
Inter-Partition Communication	10	8	598	0
Memory Management	2	1	991	0
Health Monitor Management	5	3	64	0
Trace Management	5	4	428	0
Interrupt Management	5	4	172	0
Miscellaneous	5	3	41	3
Sparc V8 Specific	12	5	88	0
Total	***61***	***39***	***2662***	***9***

C. Test Campaign

The rationale of this test campaign is to demonstrate the potential of the data type fault model in detecting robustness vulnerabilities in the hypercall categories tested. Table III shows that various calls in all categories have been tested, amounting to 64 per cent of total XM hypercalls. As shown in Fig. 8, just below 50 per cent of untested calls are hypercalls with no parameters. While, as suggested by the Ballista project [12], the data type fault model may be extended to apply to such hypercalls, this was not considered for the scope of this exercise.

During this test campaign a total of 2662 tests were defined. As listed in Table III, the execution of such test cases has raised 9 notable issues, some of which share

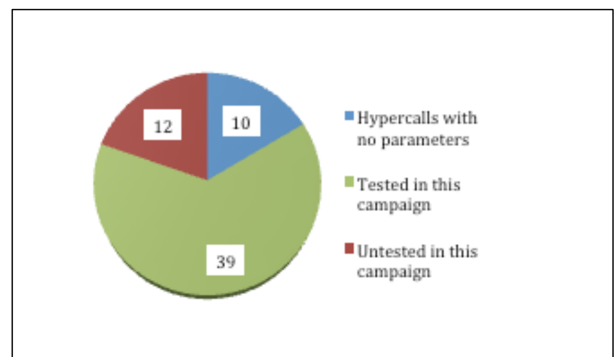

Figure 8. XtratuM test campaign distribution

common robustness vulnerabilities. The three issues raised under the *System Management* hypercall category where related to unexpected kernel operation invoked through the *XM_reset_system* hypercall. *XM_reset_system (xm_u32_t mode)* is a system partition service that resets XM in either of two modes: *XM_COLD_RESET(0)* or *XM_WARM_RESET(1)*. XM fails to correctly check the *mode* parameter and an unexpected system reset is invoked for invalid modes. This service has now been revised by the XM development team to return an *XM_INVALID_PARAM* for invalid modes.

- Test calls *XM_reset_system(2) and XM_reset_system(16)* resulted in a kernel cold reset; an operation, which should have returned the invalid parameter return code *XM_INVALID_PARAM*. According to reference documentation, a kernel cold reset may be only be invoked through hypercall *XM_reset_system (0)*.
- Test call *XM_reset_system(4294967295)* resulted in a kernel warm reset; an operation, which should have returned the invalid parameter return code *XM_INVALID_PARAM*. According to reference documentation, a kernel warm reset may be only be invoked through hypercall *XM_reset_system(1)*.

All issues raised under the *Timer Management* category were related to the *XM_set_timer*. *XM_set_timer(xm_u32_t clockId, xmTime_t absTime, xmTime_t interval)* is a standard service to arm a timer on either of the two clocks provided by XM.

- Test call *XM_set_timer(0, 1, 1)* has resulted in a system fatal error leading to an XM halt. When invoking *XM_set_timer* with small intervals, such as 1μs, the next execution time is always expired by the time it is checked and the timer handler is invoked again. This leads to a recursive loop resulting in a stack overflow. A minimum interval accepted by *XM_set_timer* has now been defined by the XM development team, and *XM_set_timer* will now return *XM_INVALID_PARAM* for interval values under 50μs.
- Test call *XM_set_timer(1, 1, 1)* results in a timer trap which crashes the TSIM simulator. As with the previous case, the unexpected trap is most likely the result of a race condition cause when *XM_set_timer* is invoked with small intervals.
- Test calls *XM_set_timer(0, 1, LLONG_MIN)* and *XM_set_timer(1, 1, LLONG_MIN)* incorrectly returned a successful operation code when invoked with a negative interval. XM fails to correctly check the *interval* parameter and does not detect an invalid negative interval. This service has now been modified by the XM development team to return an *XM_INVALID_PARAM* for invalid intervals.

The issues raised under the *Miscellaneous* hypercall category were related to *XM_multicall* hypercall. *XM_multicall(void *startAddr, void *endAddr)* offers the capability to pack several hypercalls in a buffer and then execute them as a batch. Test calls with invalid pointers to *startAddr* and *endAddr* parameters did not return an expected invalid parameter return code *XM_INVALID_PARAM*. The kernel instead attempted to execute the hypercall leading to unhandled data access exceptions. Additionally, it has been determined that such a service may lead to breaking the temporal isolation. A partition may require multiple time consuming services to process all hypercalls in the buffer, preventing nominal context switching as required by the scheduling plan. This service has been temporarily removed by the XM development team.

All of these issues have been identified by the XM Health Monitor and may be thus categorized as *Catastrophic, Restart or Abort* failures within the *CRASH* failure scale. Other *Silent* or *Hindering* faults would be potentially identified through manually crosschecking returned codes, which is being suggested as future work.

V. Discussion and Possible Improvements

While the XM robustness campaign covers a significant part of the separation kernel's functionality, this exercise was intended to be more of a demonstration of the applicability of the data type fault model to the separation kernel testing rather than a comprehensive robustness test of XM itself. Through this experimental project, the potential of such a methodology is demonstrated and a number of possible improvements emerge. The following explains aspects of the robustness testing approach that might be improved for future considerations.

Probably the most prominent drawback when using the data type fault model is the fact that result analysis cannot be automated. The output of a particular test call is context-dependent, heavily affected by the state of the system when the test call is invoked. Hence an automated oracle that can differentiate between a successful and a failed test is only possible if it considers the state of the separation kernel at that moment. This is possible if a logic model of the whole system is available [13]. Such a model could capture the state of the system when the test call is invoked and output the expected behaviour and return code based on the rule stipulated in the product manual. Additionally such a model could be potentially used to generate more effective test datasets, increasing coverage and reliability in the robustness results. The probability is however, that such a commodity is not readily available, and is to be developed during the *Preparatory Phase* of the testing campaign. This requires extra effort resulting in longer timelines [13]. As a matter of fact, other literature suggests that the creation of an oracle in this regard is to be considered impractical [12].

This exercise did not consider test cases for hypercalls with no parameters. Such calls amount to 16 per cent of all XM hypercalls. The applicability of using such a methodology to parameter-less hypercalls is not immediately apparent. While such modules cannot be directly tested through parameter sets, they are still influenced by the system state. The Ballista project proposes the use of phantom parameters [18]. This technique makes use of a dummy module that sets the appropriate system state with a phantom parameter before

calling the module under test. While this concept extends the applicability of such a methodology, it is also beneficial when testing modules with parameters. Multiple references, such as [18], [19] and [20], mention that robustness results are different when the system under test is subjected to different states and different stress conditions. Phantom parameters could be used in this case to set the separation kernel into a particular stressful state before invoking the test calls.

Although the data type fault model is considered as a black box methodology, knowledge of the typical uses of the associated data type is very relevant when choosing test values. Without such consideration, testing is possible by choosing values generically, however this might come at the expense of reduced coverage [9]. For the scope of this project, each hypercall was treated as a black box and test value selection is solely done on data type association. As an example, with *XM_reset_partition(partitionId, resetMode, status)*, parameters *partitionId* (valid values: 0 → 4) and *resetMode* (valid values: 0 → 1) are both integer types and thus their test datasets are very similar. As with the Ballista project, a more complete test scenario would have involved a basic dataset that is applicable to all integer types, together with an additional set specially selected in the context of hypercall functionality. Furthermore, as suggested in [9], different invalid values often elicit different system responses from a given hypercall. With some knowledge of the typical uses of a data type, the test administrator should attempt to identify different ways in which values can be invalid.

VI. Conclusion

The main aim of this work was to push the boundaries of separation kernel robustness testing methodologies through researching different methods, or applying known methods differently. This work has achieved this through employing the data type fault-model to the realm of separation kernel robustness testing.

As demonstrated through the XtratuM for SPARC LEON3 case study, where a significant number of previously unknown robustness vulnerabilities where identified, it is evident that the data type fault model offers potential in separation kernel robustness testing. The test campaign, which is to be considered as a study to help exhibit the potential of the testing methodology rather than a fully fledged robustness testing campaign of the separation kernel, covered over 64 per cent of total XM hypercalls. The 2662 executed tests uncovered 9 notable robustness vulnerabilities, which had gone undetected during system development fault removal campaigns. While results are promising, further work is required to quantify the efficiency and dependability of the methodology. A dry run by manually cross-checking return codes against reference documentation would be instrumental as future work in establishing a truth base to which robustness testing results may be compared.

Acknowledgment

This research was assisted by our colleagues at ESTEC-ESA, Martin Hiller and Jorge Lopez Trecastro. Their contribution supported the development of the fault injection methodology described in this work.

References

[1] NASA Office of Chief Engineer, "NASA Study on Flight Software Complexity," California, 2009.

[2] James Windsor and Kjeld Hjortnaes, "Time and Space Partitioning in Spacecraft Avionics," in *Third IEEE International Conference on Space Mission Challenges for Information Technology*, Noordwijk, The Netherlands.

[3] Martin Hiller, James Windsor, Knut Eckstein, Maria Hernek, and Kjeld Hjortnaes, "ESA Roadmap for IMA spin-in to spacecraft avionincs ," ESA SP, Volume: 701 SP, Noordwijk, The Netherlands, 2012.

[4] Raul Barbosa and Johan Karlsson , "Experiences from Verifying a Partitioning Kernel Using Fault Injection ," in *12th European Workshop on Dependable Computing, EWDC 2009* , Toulouse, 2009.

[5] John Rushby, "Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance," NASA Langley Research Center, Technical Report NASA/CR-1999-209347, 2000.

[6] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, 2004.

[7] Miguel Masmano, Alfons Crespo, and Javier Coronel, "XtratuM Hypervisor for LEON2/LEON3: Volume2: User Manual," xm-3-usermanual-022i, Spain, 2012.

[8] IEEE Computer Society, "IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)," 1990.

[9] N. Kropp, P. Koopman, and D. Siewiorek, "Automated Robustness Testing of Off-The-Shelf Software Components," in *28th Fault Tolerant Computing Symposium*, 1998.

[10] Andreas Johansson, "Robustness Evaluation of Operating Systems," 2008.

[11] Mei-Chen Hsueh, Timothy K. Tsai, and Iyer K. Ravishankar, "Fault Injection Techniques and Tools," University of Ilinois, Theme Feature 1997.

[12] Philip Koopman, Kobey DeVale, and John DeVale, "Interface Robustness Testing: Experience and Lessons Learnt from the Ballista Project," in *Dependability Benchmarking for Computer Systems*., 2008.

[13] Critical Software , "RTEMS 4.5.0 Evaluation Report," DL-RAMS02-01-05 , 2003.

[14] B. Marick, *The Craft of Software Testing*. NJ: Prentice-Hall, 1995.

[15] Victor Bos et al., "Time and Space Partitioning the EagleEye Reference Mission," Proceedings of DASIA 2013 Data Systems in Aerospace, 2013, pp. 70-76.

[16] Miguel Masmano, Javier O. Coronel, Alfons Crespo, and Patricia Balbastre, "XtratuM Hypervisor for LEON3 : Volume 4: Reference Manual," xm-3-reference-023i, Spain, 2012.

[17] SYSGO Embedding Innovations. (2015, April) PikeOS Hypervisor. [Online]. http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/

[18] John P. De Vale, Philip J. Koopman, and David J. Guttendorf, "The Ballista Software Robustness Testing Service," in *Testing Computer Software Conference*, 1999.

[19] Dominco Cotroneo, Domenico Di Leo, Roberto Natella, and Roberto Pietrantuono, "A Case Study on State-Based Robustness Testing of an Operating System for the Avionic Domain," Computer Safety, Reliability, and Security; Vol 6894 of the series Lecture Notes in Computer Science pp. 213-227.

[20] D. Cotroneo, D. Di Leo, F. Fucci and R. Natella, "SABRINE: State-based robustness testing of operating systems," Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, Silicon Valley, CA, 2013, pp. 125-135.