Evaluating the Applicability of Robustness Testing in Virtualized Environments

Charles F. Gonçalves*†, Nuno Antunes*, Marco Vieira*

*CISUC, Department of Informatics Engineering
University of Coimbra
Polo II, 3030-290 Coimbra – Portugal
charles@dei.uc.pt, nmsa@dei.uc.pt, mvieira@dei.uc.pt

†Information Governance Secretary
Federal Center for Technological Education of Minas Gerais
Av. Amazonas, 5253 - Belo Horizonte - MG, 30421-169 – Brazil
charles@cefetmg.br

***Abstract*—Virtualization provides many benefits as server consolidation and cost reduction, but it also introduces new challenges like security and isolation. Thus, trust is still one of the roadblocks in their adoption in critical systems. Virtualized systems are governed by a hypervisor and resources are shared amongst virtual machines. Paravirtualization improves the performance of the costly I/O operations, by providing an hypercall interface to the guests' kernel. Hypercalls must be robust and secure, as their abuse leads to harmful effects. This paper presents an assessment of the applicability of robustness testing to the Xen hypercall interface. For this, we devised a testing campaign by mutating valid hypercall invocations with invalid values. The campaign was then executed from a compromised machine inserted in a representative virtualization environment. The results revealed the compromised machine being crashed frequently, in some cases without notification, and also lead into inconsistent states. Results also show the inadequacy of the approach: new failure mode scales are necessary, as well as new mechanisms for failure detection.**

***Index Terms*—robustness testing, trust, virtualization**

I. Introduction

Virtualization is one of the key technologies behind Cloud computing, which is one of the trendiest concepts of the last decade. In fact, nowadays it seems that every major software systems company is moving towards cloud [1], providing diverse service models to their customers. The simplest of such models is the Infrastructure as a Service (*IaaS*), in which providers offer machines (typically virtual) and other resources to consumers [2]. Consumers have complete control over the machines and may install and run the necessary software.

Virtualization [3] allows the creation of virtual instances of physical resources [4], [5]. A hypervisor governs the virtualization infrastructure, sharing the physical resources amongst Virtual Machines (VMs) which are entitled to a contracted amount of each resource. Each instance can customize the respective virtualized resources as a dedicated instance, unaware of what else is running on the same infrastructure. This allows to reduce costs through server consolidation and to enhance the flexibility of physical infrastructures. Together with the possibility for multiple tenants (users) to use the same infrastructure simultaneously, this enabled higher degrees of scalability to organizations and also different business models in which resources are *paid per use*.

Paravirtualization enables the optimization of the performance of some virtual machine operations, namely the costly input and output operations [3]. For that, an hypercall interface is provided to be used by the kernel of a paravirtualized guests. As hypercalls are used to execute sensitive operations, their abuse can lead to harmful effects. Thus, the hypercall interface has to be robust and secure.

Robustness is *"the degree to which a system or component can function correctly in the presence of invalid inputs"* [6] and its assessment using robustness testing, have been carried out on diverse systems and components [7]–[9]. Despite those efforts, recent research shows that there is still a need for more software robustness research [10]. Considering that only 23% of organizations completely trust public clouds [11] we have a fertile ground to develop a robustness evaluation that would bring more trust to those virtualized environments.

This work presents an experimental evaluation of the **applicability of robustness testing techniques for the Xen hypercall interface** assessment. For this, we devised a testing campaign through the systematic mutation of correct requests with unexpected and invalid values. This campaign was submitted from what we deem as the "compromised machine" in a setup that included `TPCx-V` [12], a well known benchmark for virtualized environments. The goal was to understand the impact of the tests on the different components of a complex system that emulates a cloud environment. Also, we want to shed light on the failure detectors needs for such environments, understanding if widely used failure modes are applicable, and if the tests can be automated in practical ways.

The results observed included the compromised machine crashing frequently, in some cases without notification, which is a robustness issue. In some tests, the machine was led into inconsistent states. The ineffectiveness of the approach was clear: failure mode scales that consider the multiple perspectives in the cloud are lacking, and new failure detectors are necessary to understand the impact of tests.

The paper is structured as follows. Section II introduces the key concepts to understand the paper. Section III describes

978-1-5386-8489-4/18/$31.00 ©2018 IEEE

the experimental study performed, while Section IV presents and discusses the obtained results. Section V systematizes the lessons learned and finally, Section VI concludes the paper.

II. Background and Related Work

This section explains the core concepts involved on robustness testing, describe the real world context that we want to emulate, its relevance and the motivation behind this work.

A. Robustness Testing

Any trusted system must be extensively tested prior to production. Employing the needed human effort to properly test any system is usually unpractical or even prohibitively expensive. Nevertheless, the bias presented on the human mind can limit the ability of creating corner cases tests which is fundamental for a dependable system. That way, test automation is the common approach for complete test assessment.

Robustness testing is a technique that aims to automatically test the system robustness. It provides means to evaluating how a system behaves on the presence of external faults testing it without referencing its source code using only the system external interface. The intuition behind it is that by using the software in ways unforeseen by the designers, it will uncover problems in which both developers and testers did not think about. This has the potential to trigger "corner cases" which unveil design and implementation mistakes.

To find robustness problems in a system a combination of valid and invalid inputs are executed and the outcome is analyzed. Invalid inputs can be generated by applying mutation rules [9], [13], [14] according to the data type of each parameter. For instance, if the input parameter is a pointer data structure, replace by an invalid pointer or a valid one to a invalid address/object.

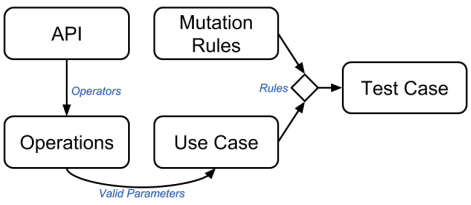

Fig. 1. Test Definition Method

In summary, an usual robustness testing approach follows a procedure that can be generally described by the process in Fig.1. Using the information gathered in the system's *API* description or documentation is possible to determine which *operations* are provided. A series of *use cases* are defined establishing the correct operations invocations using valid parameters (according with the domain of the parameter) - a proper operation use. For each *use case* adequate mutation rules will be applied on the parameters generating a *test case* - an unexpected or invalid operator usage.

Typically the failure mode used or adapted when applying robustness testing is the CRASH scale criteria [7]. CRASH is a an acronym for the possible failures: *Catastrophic* (the whole system crashes), Restart (the application become unresponsive and has to be restarted), Abort (the application terminates abnormally), Silent (invalid operation is performed without error), and Hindering (an inadequate error code is returned).

Robustness testing were applied in different domain like POSIX Operating System [7], [13], [15], self-adaptive systems [16], operating system drivers interface [8], microkernels [14], web-services [9] to cite only a few.

B. Target Environment

The real world context that we want to study is the shared infrastructure usage present on public cloud computing providers represented by the *IaaS* model. In this context a user (personal or corporate) shares the infrastructure with others tenants (users) and the overall security of such environments can be roughly bounded by the vulnerabilities and threats present on the less secure tenant on the physical host.

Usually, such environments are composed by the shared hardware, the hypervisor and its privileged domain and multiple virtual machines that we will refer to as tenants. Fig. 3 illustrate those components. We call a *Compromised Tenant (CT)* any virtual machine that is exploitable or has any malicious code running on it. Any tenant running on paravirtualization can easily access the hypercall interface through a privileged program crafted to that end and explore any vulnerability [17].

Our environment setup includes a deployment of the `TPCx-V` benchmark [12] that was devised to measure how a virtualized server runs database workloads. It exercises different components of such system, notably the hypervisor, and models many properties of cloud servers, such as multiple VMs running at different load demand levels, and large fluctuations in the load level of each VM [18].

C. Hypercall Interface Assessment

Previous analysis and practical testing has been done using hypercalls. In a practical experiment about hypercalls [17] a list of hypercall vulnerabilities found in **vulnerability report databases** are evaluated and analysed. Not all of them affected Xen, but it was highlighted that most of the discovered vulnerabilities affect the hypervisor, probably because its extensive interface. With this information it was developed a tool, called hInjector [4] that uses CT to inject attacks directly into the hypervisor using hypercalls. hInjector was developed taking into account the vulnerabilities already discovered, meaning that only some structures were supported by hInjector (e.g. *trap_info*, *physdev_get_free_pirq*, *gnttab_set_version*, *xen_memory_exchange*).

III. Experimental Study

The main goal of this study is to evaluate the applicability of robustness testing in virtualized environments. We extend the work previously done in [4], [17] by widening the hypercalls support, applying robustness test techniques in the search for any abnormal behavior that can impact any security or

dependability mechanism on the underlying hypervisor. The key motivation is to analyze the possible impact of robustness testing over the hypercall API in a complex environment like the one observed in a deployment of the `TPCx-V`.

We defined an experiment that tries to answer the following questions:

RQ1 – Are the interface faults generated by mutating the input parameters effective?

RQ2 – Is it possible to create automated mechanisms to perform tests in a realistic manner?

RQ3 – What are the failure detectors requirements and capabilities for such environments?

RQ4 – Are the usual failure modes (e.g. CRASH scale) applicable in this domain?

A. Experimental Procedure

The first step needed to answer all those questions is to have a mechanism that enables the injection of tests into the hypervisor. For that, we adapted the *hInjector* tool presented in [4] to support our research. The *hInjector* was designed to evaluate *Intrusion Detection System* (IDS) under virtualized environments, it enables the injection of attacks directed to the Xen hypervisor using hypercalls. We performed adjustments to instrument each hypercall to provide useful information for our assessment like informing return codes, dynamically select which hypercall to test and other instrumentation that does not affect the overall performance neither are too intrusive.

A second step was to acquire basic knowledge of the hypercall interface, such as the set of hypercalls available, which operations they provide and what are the expected outcome of each one. Knowing the characteristics about the hypercalls allows the creation of realistic *use cases*, which will be used as a starting point to create the *test cases*. This step proved challenging since the hypercall API is quite extensive and have a limited documentation. To better understand the API, the source code was analyzed and as an outcome we detailed the parameters domain for those hypercalls (listed on Table I). The number of tests is defined by the number of mutation rules applied for each parameter. The runs is the number of execution of tests during the exploratory study. Note that, an Hypercall can have many operations and the suffix `_op` does not relate to this connection between hypercall and operations.

The next step was to define mutation rules, such as null and empty values, pointers for valid/invalid memory locations, maximum/minimum valid values in/outside the domain of the variable, values that cause data type overflow etc. This step follows best practices done on previous work (e.g. [13], [9]). The list of mutation rules used can be seen in Table II.

The information gathered of the hypervisor interface also allows us to define the mutation rules that are applied to those *use cases*. Each *test case* contains only one mutation and only in one parameter. The result is a mixed set of *use cases* and *test cases* that can be used to assess the interface.

Clarifying what is a *use case* in the context of this work, it is a set of valid values (according with the domain of the parameter) that will be used as input to a hypercall. A *test case* is a set of values, based on a *use case*, where one of its values is changed according to a mutation rule.

TABLE I
HYPERCALLS AND OPERATIONS COVERED (XEN 4.4)

	Xen 4.4	Covered	%
Hypercalls	39	26	66.6
Operations	285	95	33.3

Hypercall	N°	Tests	Runs
HYPERVISOR_set_trap_table	0	335	805
HYPERVISOR_mmu_update	1	395	790
HYPERVISOR_set_gdt	2	150	180
HYPERVISOR_stack_switch	3	150	449
HYPERVISOR_set_callbacks	4	300	758
HYPERVISOR_fpu_taskswitch	5	90	90
HYPERVISOR_sched_op_compat	6	530	608
HYPERVISOR_set_debugreg	8	165	198
HYPERVISOR_get_debugreg	9	90	108
HYPERVISOR_update_descriptor	10	150	180
HYPERVISOR_memory_op	12	4690	10153
HYPERVISOR_update_va_mapping	14	240	240
HYPERVISOR_set_timer_op	15	75	375
HYPERVISOR_console_io	18	220	264
HYPERVISOR_grant_table_op	20	5355	8975
HYPERVISOR_vm_assist	21	150	181
HYPERVISOR_update_va_mapping_od	22	315	379
HYPERVISOR_iret	23	5	10
HYPERVISOR_set_segment_base	25	165	239
HYPERVISOR_xsm_op	27	90	126
HYPERVISOR_nmi_op	28	150	181
HYPERVISOR_sched_op	29	1240	2656
HYPERVISOR_callback_op	30	30	39
HYPERVISOR_event_channel_op	32	3500	4200
HYPERVISOR_physdev_op	33	8335	19243
HYPERVISOR_hvm_op	34	825	990

Tying all the described steps we end up with a *test case* generation process that can be represented as in Fig. 2. For each parameter of every hypercall we apply every possible mutation to generate all possible *test cases*. This process generated more than 28 thousand tests which in time spawn more than 50 thousand runs. On the testing execution process, for each test a Loadable Kernel Module (LKM) module would be generated, then injected on the CT Kernel to test the hypercall and the outcome will be assessed based on facts gathered over the environment.

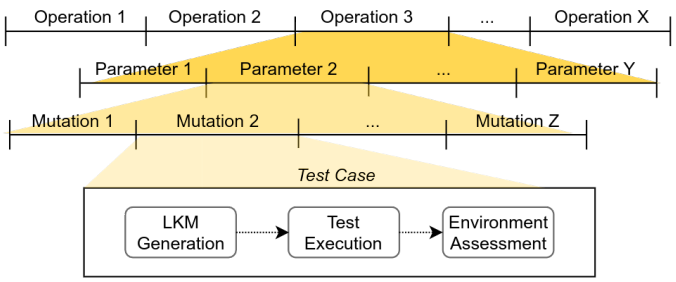

Fig. 2. *Test Case* derivation process and its life cycle

TABLE II
MUTATION RULES APPLIED ON THE API PARAMETERS

DataType	Rules
Pointer*	Invalid pointer to an allowed memory zone Invalid pointer to a forbidden zone of memory NULL pointer
Struct	Fill with dynamically allocated struct Replace by another struct type (of the same size) Replace by smaller struct NULL pointer
Signed Integer	Replace by NULL value Replace by 0 Replace by 1 Replace by -1 Add one Subtract one Replace by type maximum value Replace by type maximum value plus one Replace by type maximum value minus one Replace by type minimum value Replace by type minimum value plus one Replace by type minimum value minus one Replace by domain maximum value Replace by domain maximum value plus one Replace by domain maximum value minus one Replace by domain minimum value Replace by domain minimum value plus one Replace by domain minimum value minus one
Unsigned Integer	Replace by NULL value Add one Subtract one Replace by type maximum value Replace by type maximum value plus one Replace by type maximum value minus one Replace by type minimum value Replace by type minimum value plus one Replace by type minimum value minus one Replace by domain maximum value Replace by domain maximum value plus one Replace by domain maximum value minus one Replace by domain minimum value Replace by domain minimum value plus one Replace by domain minimum value minus one
String (Char*)	Replace by NULL value Replace by empty string Replace by a predefined character Replace by a predefined string Replace by nonprintable character Replace by string with nonprintable characters Remove null character ('\0') Add nonprintable characters to string Replace by alphanumeric string Add characters to overflow max size

B. Experimental Setup

A virtualization environment that is representative of a real world scenario (as those observed in Cloud Computing) is quite complex. A possible instantiation that is of our interest, as detailed on section II-B, is the one composed by many users sharing the same infrastructure and one of those is trying to explore the vulnerabilities present on the underlying hypervisor. We can observe this possible scenario on Figure 3.

Our experimental setup is composed of a Xen hypervisor (4.4.1), a privileged domain, named dom0 in Xen's terminology, and 14 virtual machines with diverse characteristics that represent different tenants sharing the same infrastructure.

As explained in section II-B, we adopted the *TPCx-V* benchmark workload and setup to tailor our environment

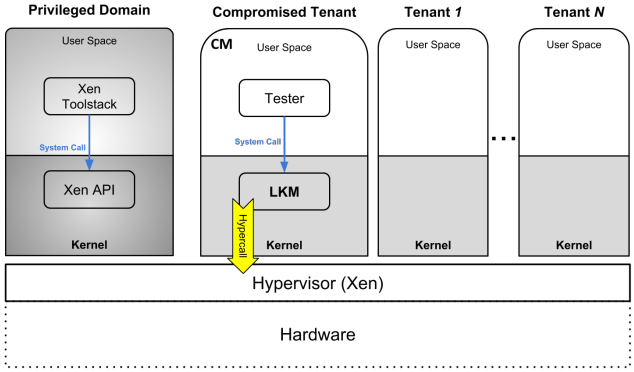

Fig. 3. Testing Environment and its components relations

providing that way a more realistic execution scenario. The CT in our setup is a dedicated virtual machine where all tests are executed. This deployment is supported by a Dell PowerEdge R710 with 24 Cores, 96 GB Ram and 12 TB disk.

C. Robustness Testing Campaign

Our experimental campaign covers 66.6% of the API as we can see in Table I. The method used to choose which hypercall to access was based on the trade-off between: I) the information available by each hypercall and the effort to understand the parameters values and domain; and II) hypercall usage over a run of the *TPCx-V* benchmark. To determine which hypercall was used over the complete run of the *TPCx-V* we profiled the Xen code to gather that information. Only 13 hypercall were used during the *TPCx-V* run, from those we only do not cover four (`HYPERVISOR_multicall`, `HYPERVISOR_xen_version`, `HYPERVISOR_vcpu_op`, and `HYPERVISOR_mmuext_op`). Still, not every hypercall has the same relevance in the virtualization aspect, for instance, the `HYPERVISOR_xen_version` are commonly used to provide information about the current Xen's version running to guide compatibility operations to the guest OS. Nevertheless, we also know that the robustness problems and security issues are not exclusivity of relevant operations and we intend to address those excluded operations in future works.

Basically the testing process (*Tester*) is a program that iterates over a list of *test cases*. For each one the *Tester* performs the test injection loading the pre-compiled LKM module into the CT's Kernel and collects relevant information to future analysis. Information about the hypervisor and other tenants are collected on the privileged domain using the Xen Toolstack. After each test the data collected on CT and the privileged domain are evaluated. The number of tests executed for each hypercall can be seen in Table I.

IV. RESULTS AND DISCUSSION

The first testing approach tried to assess *how the hypervisor would respond to a test load containing all test generated* executed sequentially. For that we needed to perform a series of consecutive tests under the CT. But executing all tests

at once was not possible since the Xen Hypervisor has a series of security mechanisms that kill any VM that behave inappropriately. These mechanism is triggered by some tests what interrupts the serial execution of all tests.

The second effort was to determine *which was the subset of tests that did not trigger those Xen's security mechanisms*. We performed a series of individual tests on the CT accounting which one "*crashed*" the VM. After this test we accounted that 12.1% of the overall tests caused the CT to be killed. Evaluating the results we noted that not every crash was caused, explicitly, by a Xen's security mechanisms. Sometimes the CT was silently crashed, and we started distinguishing the CT failure between *GUEST_CRASH*, where the CT was crashed without any explicit information from the hypervisor or *GUEST_KILLED* where there are traces of the Xen's security mechanisms.

The kind of failure observed in a *GUEST_CRASH* can be classified as a *Silent* failure based on the CRASH scale. According to discussions in the xen-devel mailing list, a silent crash should never happen, thus when it happens, a clear case of lack of robustness is present. But using only the information collected by our methodology does not suffice to determine if all those failures are a truly silent crashes. Further information should be collected using adequate serial consoles that were not included in our experiments.

To validate the subset of tests that did not crash the CT we re-executed all of them. In this new execution new tests led to crash failures. To better understand the test cases' predictability we the split the tests by hypercall. The only hypercall that showed an unpredictable behavior was the `HYPERVISOR_grant_table_op`, which has 5355 tests and in the first execution produced 2617 crashes. Subsequent executions were performed looking for a convergent subset of test that did led to crash failures. For each new execution, the previous cases that led to crashes were removed to create a set of tests that would not crash the CT. The convergent series of crashes during execution was: 2617, 4, 3, 2, 1, 1, 1, 1, 0, 0, 0.

After defining which subset lead to CT abrupt interruption we performed a further analysis on the test results to define which observed behaviour could be extracted from the experiment logs. Some tests failed during its execution leading to earlier error. Those tests were not even fully loaded into the CT Kernel. Most of them (32) by raising a *SIGSEGV* during the LKM execution. The other two led to an invalid module format that could indicate a problem during the LKM generation.

TABLE III
TESTS RESULTS BREAKDOWN BY *State*

State	*#*
ERROR	34
GUEST_CRASH	6642
GUEST_HANG	2
GUEST_KILLED	248
INJECTION_HANG	20
RESTART	26
SUCCESS	24521
UNKNOWN_ERROR	195

The following behaviours were observed (Table III summarizes the results). Note that silent failures, where the system performed some invalid operation that could not be externally identified, could not be identified due to the lack of visibility to the internal behaviour of the CT:

- **ERROR**: Error on LKM module loading (error code)
- **GUEST_CRASH**: CT crashes without any notification (on Dom0)
- **GUEST_HANG**: CT became unresponsive
- **GUEST_KILLED**: CT is deliberately killed by the hypervisor
- **INJECTION_HANG**: The CT is up, the Tester process is running but did not return in the period accessed
- **RESTART**: The CT is running but it has restarted
- **SUCCESS**: The Tester process injected the test and returned appropriately, the CT is running [1]
- **UNKNOWN_ERROR**: A timeout when waiting for a test response (none of the previous state was identified)

Despite the fact that there is not a consistent return status throughout all operations, there is subset that returns a type named `neg_errnoval`[2] that allows us to identify robust operations. But this is only possible for 50.7% of all tests. (see Table IV). Which let us with the huge effort to perform *ad hoc* analysis for all other operations, justifying that way the need for failure detectors tailored for such environments.

With regard to detection of lack of robustness based on successful operations (return code zero) we can only apply this method for the same percentage of the tests which represent 11.2% over the overall tests (22.1% over the subset which return a meaningful code). Even though, for such operations further investigation should be performed to confirm a lack of robustness.

TABLE IV
BREAKDOWN BY EXIT CODES

#	Nº	Code	Description
97	-95	EOPNOTSUPP	Operation not supported on transport endpoint
4590	-38	ENOSYS	Function not implemented
133	-28	ENOSPC	No space left on device
5231	-22	EINVAL	Invalid argument
414	-19	ENODEV	No such device
21	-17	EEXIST	File exists
24	-16	EBUSY	Device or resource busy
2621	-14	EFAULT	Bad address
4131	-3	ESRCH	No such process
355	-2	ENOENT	No such file or directory
3070	-1	EPERM	Operation not permitted
5881	0	-	Normal Operation

V. LESSONS LEARNED AND OPEN CHALLENGES

The results from our experimental campaign showed that further research is necessary for this particular domain, and the robustness testing approach revealed rather ineffective, showing the need for new and adequate methods.

[1]Note that this class may include cases were some feedback should have been given. The *Silent* class on [13]

[2]This type defines a tuple a integer and a code, which are used to identify robust operations on Table IV

Concerning the **RQ1**, we found that the traditional mutation method to create invalid inputs, based on rules over the input data types, seems to have a limit application on those virtualized infrastructures. Few inputs parameter has meanings related with its type. e.g in `set_timer_op` the parameter has contextual meaning (time is well modeled by an integer). Many parameters relate with other VM's references, mutating them lead only to a 'not existing domain' warning/error. Also, some operations are memory specific, for instance, `HYPERVISOR_set_segment_base`, and the domain for that are also context dependent and the effects of possible robustness failures would only be noted by following memory operations;

We believe that realist injection mechanisms are possible and can be made automatically (**RQ2**). But for that, tests should be context aware to include the environment pieces that could suffer the effects, for instance, possible domain ids and specific addresses should be taken into account. Some operations are state and context related and need multiple operations, a single test may not reveal problems. The CT can be affected by the effects of the tests on the hypervisor, that way, mechanisms to detect and overcome possible impacts of that should be devised.

In the target environment of this study, the ability to correctly detect the presence of failures depends of an holistic view needed to connect all evidences to any fault/failure. The failure detector should be operator aware since: i) there is not a unified way to access exit status, usually they are operation specific. e.g always zero, specific register value, multiple return status contexts (status code or register values); ii) some operations destroy or reboot guests and should be blacklisted. At last, some virtualization operations provided by the hypercall API mix a set of hardware (direct memory, and interruptions) and software operations (multiplexing memory and IO operations) and should be handled in a different manner for an effective failure detection. Those are failure detectors requirements that we identified and which answer the **RQ3**.

With regard to **RQ4** we found out that the traditional failure modes commonly used in the literature are not very descriptive for virtualized environments. The multi-perspective notion of failure makes the existing failure modes very hard to be accurately applied in the depiction of the test results. There are several levels at which failures may occur, with very diverse impacts for the different stakeholders of the systems. The existence of many trust relations in a virtualized environment makes it difficult to understand who is affected by the each of the failure types;

VI. Conclusions

IaaS virtualization solutions that are used by multiple and independent tenants are nowadays the norm, and are being trusted to host their businesses. To be trustworthy, these solutions must be robust and secure.

We presented an experimental assessment of the applicability of robustness testing for the evaluation of Xen hypervisor. Robustness testing has long been used to assess the robustness of systems, looking for corner cases that the programmers did not consider.

The results show that further research is necessary in this particular domain, and the robustness testing approach revealed rather ineffective, showing the need for new and adequate approaches.

In particular, new and more effective ways to identify failures are necessary. Additionally, it is necessary to also research and propose new ways to characterize these failures, according to the subsystems affected, the stakeholders involved and the relations of trust among them.

References

[1] "User Stories - OpenStack Open Source Cloud Computing Software." [Online]. Available: https://www.openstack.org/user-stories/

[2] "Internet engineering task force (ietf)." http://www.ietf.org/.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS operating systems review*, vol. 37, 2003, pp. 164–177.

[4] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, S. Kounev, A. Avritzer, and M. Luft, "Evaluation of intrusion detection systems in virtualized environments using attack injection," in *International Symposium on Recent Advances in Intrusion Detection*, 2015.

[5] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, "Virtualization for high-performance computing," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 2, pp. 8–11, 2006.

[6] J. Radatz, A. Geraci, and F. Katki, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std*, vol. 610121990, 1990.

[7] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," *Proceedings of 16th IEEE Symposium on Reliable Distributed Systems*, pp. 72–79, 1997.

[8] A. Albinet, J. Arlat, and J.-C. Fabre, "Characterization of the impact of faulty drivers on the robustness of the Linux kernel," *International Conference on Dependable Systems and Networks*, pp. 867–876, 2004.

[9] M. Vieira, N. Laranjeiro, and H. Madeira, "Assessing Robustness of Web-Services Infrastructures," in *37th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN07)*, 2007, pp. 131–136.

[10] A. Shahrokni and R. Feldt, "A systematic review of software robustness," in *Information and Software Technology*, vol. 55, no. 1, 2013, pp. 1–17.

[11] "2017 State Of Cloud Adoption And Security." [Online]. Available: https://www.forbes.com/sites/louiscolumbus/2017/04/23/2017-state-of-cloud-adoption-and-security/

[12] A. Bond, D. Johnson, G. Kopczynski, and H. R. Taheri, "Profiling the performance of virtualized databases with the tpcx-v benchmark," in *7th TPC Technology Conference, TPCTC 2015*, 2015, pp. 156–172.

[13] P. Koopman and J. DeVale, "The exception handling effectiveness of POSIX operating systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 837–848, 2000.

[14] M. Rodríguez, F. Salles, J.-C. Fabre, and J. Arlat, "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid," in *Dependable Computing*. Springer, Berlin, Heidelberg, 1999, vol. 1667, pp. 143–160.

[15] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated Robustness Testing of Off-the-Shelf Software Components," *Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, 1998*, pp. 230–239, 1998.

[16] J. Cámara, R. de Lemos, N. Laranjeiro, R. Ventura, and M. Vieira, "Robustness evaluation of the rainbow framework for self-adaptation," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*. New York, New York, USA: ACM Press, 2014, pp. 376–383.

[17] A. Milenkoski, B. D. Payne, N. Antunes, M. Vieira, and S. Kounev, "Experience report: an analysis of hypercall handler vulnerabilities," in *IEEE 25th International Symposium on Software Reliability Engineering (ISSRE 2014)*. IEEE, 2014, pp. 100–111.

[18] A. Bond, D. Johnson, G. Kopczynski, and H. R. Taheri, "Architecture and performance characteristics of a postgresql implementation of the TPC-E and TPC-V workloads," in *5th TPC Technology Conference, TPCTC 2013*, 2013, pp. 77–92.