TinyChecker: Transparent Protection of VMs against Hypervisor Failures with Nested Virtualization

Cheng Tan,† ‡, Yubin Xia† ‡, Haibo Chen†, Binyu Zang‡
†*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*
‡*School of Computer Science, Fudan University*

***Abstract*—The increasing amount of resources in a single machine constantly increases the level of server consolidation for virtualization. However, along with the improvement of server efficiency, the dependability of the virtualization layer is not being progressed towards the right direction; instead, the hypervisor level is more vulnerable to diverse failures due to the increasing complexity and scale of the hypervisor layer. This makes tens to hundreds of production VMs in a machine easily risk a single point of failure.**

This paper tries to mitigate this problem by proposing a technique called TinyChecker, which uses a tiny nested hypervisor to transparently protect guest VMs against failures in the hypervisor layer. TinyChecker is a very small software layer designated for transparent failure detection and recovery, whose reliability can be guaranteed by its small size and possible further formal verification. TinyChecker records all the communication context between VM and hypervisor, protects the critical VM data, detects and recovers the hypervisors among failures. TinyChecker is currently still in an early stage, we report our design consideration and initial evaluation results.

***Keywords*-Hypervisor failure, Failure detection, Fault tolerance**

I. Introduction

Virtualization has been a common technique for current multi-tenant cloud. By running multiple VMs atop a virtual machine monitor (VMM, or hypervisor), the efficiency of a server machine can be maximized by multiplexing resources among the hosted VMs and smoothing peak time resource demands. With the amounting of resources in a single machine tightly following Moore's Law, the number of VMs being consolidated to a single machine is constantly increasing and it is no surprise to see tens to hundreds of VMs running atop a single machine.

The growing level of server consolidation also creates increasing demand of high dependability of the virtualization layer. Unfortunately, with more and more functionalities being integrated into the virtualization layer, the scale and complexity of the virtualization layer also increase dramatically since its initial design. Table I shows the increase of code size among different major versions. The constantly adding complexity also decreases the reliability of the virtualization, which has been reflected in frequent bug/crash reports in major hypervisor vendors' mailing list. With the increasing number of VMs in a single machine, a tiny bug or exploit of security vulnerability may crash the whole virtualization layer, rendering hundreds of VMs out of services.

Table I
Xen code's expansion

version	VMM	Dom0 Kernel	Tools
Xen v2.0	45K	4136K	26K
Xen v3.0	121K	4807K	143K
Xen v4.0	270K	7560K	647K

While it is impossible to completely survive failures in the virtualization layer, prior research has resulted in a number of mitigation approaches. One viable approach would be periodically checkpointing the running VMs [11] and restoring the checkpointed VMs to another VMM. However, the cost can be quite high for hundreds of VMs running atop a single hypervisor and the states between two checkpoints will be lost. The replicated state machine approach [10] is also quite heavyweight and cannot trivially handle multiprocessor VMs. Practically monitoring the status of hypervisor and live migrating the VMs to other hypervisors are also heavyweight in requiring in creating network bursts and it is not always safe for migration under a buggy VM. ReHype [7], instead, is designed to survive hypervisor failures in place. However, the detection and recovery modules is collocated with the hypervisor, which can be easily contaminated or even corrupted by the failures in the hypervisor.

In this paper, we propose TinyChecker, which provides transparent detection and recovery of VMs against hypervisor failures. Unlike previous approaches, TinyChecker provides in-place recovery and is completely isolated from the questioning hypervisor. The key technique of our approach is leveraging a special-purposed hypervisor using nested virtualization [12] for failure detection and recovery. Because of the small size, TinyChecker can be more reliable than commodity hypervisor. It is also possible to use formal verification assure TinyChecker's correctness.

By interposing the control and data exchange between the hypervisor and the guest VMs, TinyChecker can transparently detect hypervisor's failure and recover the system without losing the on-going work in per-exit level. TinyChecker also replicates critical VM states from the hypervisor to defend against wild writes upon failures. When a failure is detected, TinyChecker will reboot the hypervisor while preserving the states and data of the VMs in the memory.

We have done a preliminary implementation of TinyChecker, which is a tiny nested hypervisor with the failure checking module. A set of fault injection experiments shows that TinyChecker can successfully survive hypervisor failures. An initial performance evaluation show that the overhead incurred by TinyChecker is very small.

The rest of this paper is organized as follows. The next section discusses about existing solutions, challenges and our fault model. Section 3 describes TinyChecker and the mechanisms to address the challenges. In section 4, the paper explains how to handle the failures in detail. The performance evaluation will be presented in section 5. Finally, we introduce the related work in section 6 and summarize our conclusions in section 7.

II. Hypervisor fault tolerance

Hypervisor failures which will cause hypervisor or VM corruption can be triggered by many software vulnerabilities or bugs. As the result of hypervisor failures, the whole platform will be out of service. Even worse, hypervisor erroneous actions may crash the management data in hypervisor or even corrupt the memory in guest VMs.

A. Hypervisor Microreboot

In order to tolerant the failures, virtualized system should have more efficient and effective abilities to correct the corrupted states. In order to overcome the shortcomings of naive rebooting, microreboot [8], [9] have been introduced to accelerate reboot procedure and preserve the work in progress. In a similar way, hypervisor microreboot [6] has been proposed to avoid rebooting all guest VMs. Generally, hypervisor microreboot involves three steps:1) Failure detection; 2) Hypervisor recovery and 3) State merge. Through these steps, the system can preserve the state of VM in memory and restore them when a new hypervisor is ready.

Failures can be classified as crash, hang, memory corruption and silent failure. Crash is an explicitly indicated failure which can be detected by existing hypervisor exception handlers. Hang is a failure where hypervisor is out of services. Typical method uses watchdog which sends periodic interrupts to check the availability of hypervisor. Memory corruption is the most dangerous failure, which may crash the critical data both in hypervisor and guest VMs. Some mechanisms, such as redundant data, can solve this problem to some extent. However, these solutions cause non-trivial performance overhead and it cannot guarantee recovering to the most recent state. Silent failure is a fault which the hypervisor has run some erroneous code, but no panic or exception is triggered. It is more difficult to detect than the previous failures.

Hypervisor recovery is the procedure to correct the mass states after a failure happens. Corrupted state can be recovered by simply reboot the hypervisor, but normal reboot leads to lose all the current states.

B. Challenges

Traditional failure detection techniques will miss some of the failures which will not trigger crash or hang explicitly. These failures include wrongly updating CPU states, corrupting VM memory and critical data in hypervisor. How to detect the failures is a key challenge for hypervisor fault tolerance. TinyChecker uses a new mechanism called context-aware checking which will be described in section 3B to deal with this challenge. The basic idea of context-aware checking is to let TinyChecker know the context of the hypervisor's actions in order to detect its failures.

Another challenge is how to verify the correctness of the preserved data which may have been crashed in the hypervisor failure. If simply reusing all the data without any assurance, the new hypervisor instance may fail another time. In order to avoid the malicious modification, TinyChecker uses on-demand checkpoint mechanism to solve this problem. The basic idea of on-demand checkpoint is to create a checkpoint for the old version when a suspicious update happens.

C. Fault Model

In our fault model, all the failures are assumed to be caused by software bugs. Software bugs can run arbitrary code and modify arbitrary memory space, because everything may happen in an error condition. We currently do not consider hardware failures.

The outcome of the failures can be manifested as four types which are introduced in section 2A. Software bugs can cause all the failures by jumping to impossible branches and updating the memory incorrectly. In order to prevent the affection of hypervisor software failures, hardware virtualization is used to support the isolation between hypervisors and TinyChecker. In this way, hypervisor failures cannot propagate to the TinyChecker space. On the other hand, prior experiences shows that a smaller code size usually represents more reliable software[4][5]. Because TinyChecker has small code size, it will be more likely to have strong reliability than commodity hypervisor. Furthermore, compiler-based techniques can be used to strength the reliability of TinyChecker and formal verification method can be used to confirm the correctness of TinyChecker in the future.

III. TinyChecker

A. Overview

TinyChecker is a tiny nested hypervisor which runs underneath the commodity hypervisor. It can provide transparent detection and recovery of guest VMs against hypervisor failures. There are three parts in the TinyChecker: access recorder, memory protector and failure detector.

Access recorder responds for recording the entire communication context between VMs and hypervisor into a table called request table. Communication represents vmexit and vmentry which are the transitions between VMX non-root operations and VMX root operations. Request table contains three columns which are domain_id, vmexit reason and time of the vmexit. Request table will support the failure checking by providing vmexit context.

The duty of memory protector is to guarantee the integrity of the critical data. During the TinyChecker initialization stage, the critical area in memory will be set to readable but non-writable to hypervisor. The critical areas in hypervisor are the metadata for managing VMs, such as p2m table or running domain list. VM's memory is another critical areas which has also been protected by TinyChecker after its allocation. However, not all the pages in VM's address space will be protected. I/O buffer is one exception as it is the communication channel between the hypervisor and guest VMs.

During the VM's initialization, I/O buffer's addresses have been detected by listening on the special registers and DMA request. These pages are not protected by TinyChecker, because it is legal for hypervisor to fill these pages. In short, memory protector has protected the critical memory from hypervisor's arbitrary modification.

Failure detector is the core part of TinyChecker which is going to detect and confirm the hypervisor failure's occurrence. Information provided by access recorder and memory protector helps failure detector to verify the suspicious operations. Through two mechanisms called context-aware checking and on-demand checkpoint failure detector can detect failures happened in hypervisor and provide an consistent state after the failure.

Figure 1 shows the overview of TinyChecker.

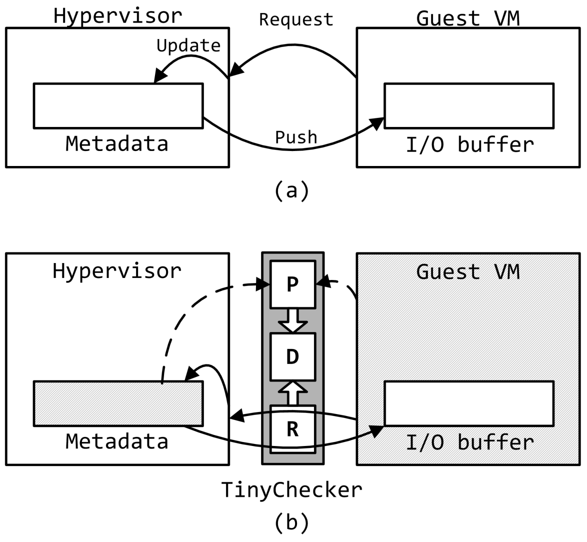

Figure 1. TinyChecker overview: P represents Memory Protector, R represents Access Recorder and D represents Failure Detector

Figure 1(a) describes the normal behavior of an vmexit and the hypervisor's actions. First, the VM traps into hypervisor. Through checking the vmexit reason, hypervisor serve the VM's request by updating its metadata or filling the I/O buffer.

Figure 1(b) shows the logical structure of TinyChecker. The shaded area has been protected by the memory protector, and all the modification in these areas will be detected. All the communication between guest VMs and the hypervisor will be interposed by access recorder.Failure detector receives the information from the other two parts of TinyChecker and makes the decision that whether a failure happens in this turn of hypervisor's execution.

B. Context-aware checking

Using current solutions, failures which do not trigger crash or hang will either cannot be detected or cause non-trivial overhead. In order to capture the failures accurately and efficiently, detailed information is needed to check suspicious operations.

TinyChecker introduces a checking mechanism called context-aware checking to detect and confirm the failure which is difficult to discover. The basic idea of context-aware checking is letting failure detector know what actions the hypervisor will take in the following period based on the vmexit reason in VMCS. VMCS is short for virtual machine control structure which controls the transitions of vmentry and vmexit. Hypervisor behaves differently when the VM exit with different reasons. In detail, when a VM causes a vmexit, the vmexit reason and its qualification will be saved in the corresponding part of VMCS. TinyChecker first parses the VMCS and record the reason of this vmexit. Different exit reasons determine different registers or different part of VMCS to be updated. Through analyzing the reason and qualification of the VM request, TinyChecker hashes the irrelevant data both in VMCS and general purpose registers. The hash will be saved for further checking. At the time of vmresume, TinyChecker will hash the current state again to check whether there is an illegal modification.

Compared with CPU states in VMCS and registers, memory updates are more complicated. Concerning the update to the metadata in hypervisor or the VM memory, TinyChecker uses a context signature table to verify the correctness of this memory update operation. Context signature is the hash of specific triplet which includes vmexit reason, the binary code block involving this operation and the memory area which this operation wants to update. In the triplet, vmexit reason helps TinyChecker to verify the control flow of the hypervisor. The recorded binary code block is a certain length of memory which have been pointed by pc register at that time. It helps to identify the memory access operation and integrity of the code in hypervisor. Different vmexit and code will modify different memory areas, the memory area in triplet guarantees that operations update the right part of memory. The main process of context-aware checking is showed in figure 2.

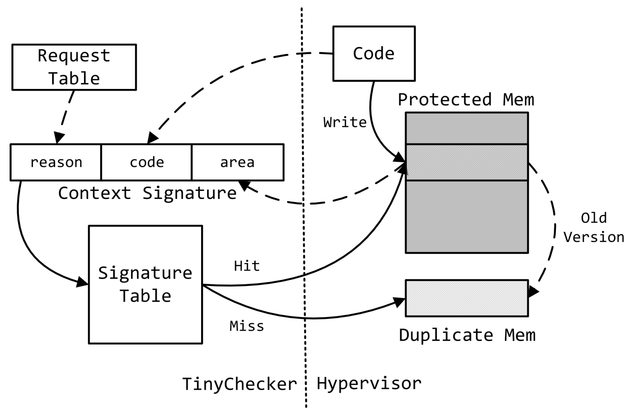

Figure 2. Context-aware Checking

TinyChecker calculates the hash of context signature of one write memory instruction, and search the signature table to see whether this operation is safe. If the hash hits, the operation should be executed faithfully, else it is considered to be a suspicious memory operation and an on-demand checkpoint will be made. By both state and memory context-aware checking, TinyChecker can detect all kinds of failures which modify the data it should not have to.

C. *On-demand checkpoint*

Periodic checkpoint is an important method for fault tolerance today. It can provide a consistent and correct state at the time the checkpoint is made. However, periodic checkpoint causes non-trivial performance loss and may not provide the latest state just before the failure happens. TinyChecker introduces a mechanism called on-demand checkpoint which will fulfill the requirements of both the little performance loss and the latest state of the system. The key idea of on-demand checkpoint is to make a redundant data structure only when some suspicious updates happens.

Because of the memory protection, TinyChecker can detect all the modification in the critical memory area. Further, through the context-aware checking, the operation will be identified as either safe or suspicious. TinyChecker will make an on-demand checkpoint if the operation is considered to be suspicious. One on-demand checkpoint will exist from the time it has been created to the time this operation is confirmed to be safe. If any failures have been detected in this period of time, TinyChecker will roll back its on-demand checkpoints. In this case, the critical part in the system will roll back to the state just before the failure has happened.

IV. Handle the failures

Hypervisor failures can be classified as four types: crash, hang, memory corruption and silent failure. The features of each failure are described in Table 2.

Table II
Failure classification

Failure type	Trigger Handler	Never Resume	Corrupt Memory	Corrupt State
Crash	Y	-	-	-
Hang	N	Y	-	-
Memory Corruption	N	N	Y	-
Silent Failure	N	N	N	Y

In the table above, ’Y’ represents this situation will happen in this failure; ’N’ represents this situation will not happen and ’-’ represents the corresponding situation may happen.

A. *Crash*

Crash is a failure which has been explicitly declared in the hypervisor. In Xen[1] a lot of BUG function have been written in the branch which should never reach. If any BUG function is triggered, it means a dangerous condition is satisfied and a failure has occurred in the hypervisor. Other crash situations will trigger panic and exception handlers which are easy to detect.

B. *Hang*

Hang is another failure whose outcome is the out of services. TinyChecker uses the request table with the interval of the vmexit service time to measure whether the hypervisor is out of services. Access recorder records every VM’s vmexit as triples (domid, reason, time) and delete this entry when successfully resume to VM. Because the vmexits must be in sequential order within one VM, the number of entries in the request table will not exceed the number of active domains. With these information, a straight forward detecting method is to check how long have the requests stayed in the table. If all the request stay for a long time and no requests are met in this period, the hypervisor is confirmed to be hanging.

A watchdog will periodically send a NMI signal to check the service time. If all the vmexit time intervals are larger than a certain threshold (e.g., 2s), the hypervisor is considered to meet some failures.

C. *Memory Corruption*

Memory corruption is the failure which hypervisor has wild wrote the critical memory. When hypervisor fails, it may run arbitrary code that may write arbitrary memory. Since the data cannot be reused for recovery, it is hard to tolerant the memory corruption failure. Nowadays solution involves periodic checkpoint, redundant data structures and redundant computations. However, these generic solutions lead non-trivial performance loss. TinyChecker use on-demand checkpoint to reduce the overhead of redundant data structures. In TinyChecker, the data will be duplicated only when needed.

TinyChecker's memory protector begins its initialization procedures after hypervisor's initialization. First, the critical parts in hypervisor's memory will be detected and divided into several areas. The following areas should be protected:

1) Domain list
2) EPT page table for VM
3) IRQ descriptor table and IO-APIC entries
4) Shared page between hypervisor and VM

These areas are critical for recovery, because they should be reused to quick reboot the hypervisor. Second, the write ability to these areas will be deprived, but the read ability is retained. In this case, the updating actions in these areas will cause EPT violation and trap into TinyChecker. In the corresponding handler, TinyChecker will first do the context-aware checking to verify whether this operation is suspicious. If it is safe, which means hash hit in the context signature table, TinyChecker will help to update this memory. Yet, if the hash miss in context signature, the memory protector will duplicate the old version of this page as an on-demand checkpoint and then update the corresponding page. In such case, memory protector will guarantee the integrity of the memory states under the suspicious operations.

Last but not least, when the VM is created, TinyChecker will detect the I/O buffer in the VM. Memory protector will protect all the VM memory except the I/O buffer using the same mechanism as hypervisor metadata protection.

D. *Silent Failure*

Silent failure is the hardest failure to detect which bypass the crash handler, memory protector checking and do not trigger hypervisor hang. The outcome of the silent failure is that the VM fails after a vmexit. Since the VM's memory except I/O buffer and the control data structure in hypervisor have both been protected by memory protector, any failures in memory can be detected. Thus, the reasons of VM failure can be caused by either wrong VMCS state or wrong general purpose registers. In order to protect the integrity of these unchanged data, TinyChecker uses context-aware checking to prevent the reserved VMCS state and registers against being wrongly modified.

As described previous sections, TinyChecker always knows what reason cause this vmexit and what the hypervisor should do. When a vmexit happens, the access recorder not only records this request, but also hashes the VMCS and registers which should not be modified based on the vmexit reason and its qualification information in the VMCS. TinyChecker will check the hash of current VMCS and registers just before vmresume is executed. If there is a mismatch of current hash and saved hash, a silent failure has taken place.

V. Priliminary Evaluation

A. *Failure detection*

Because TinyChecker is still in its early stage and some functions are not fully completed, we only use register bit-flip as the fault injection method. Other injection measures such as memory injection and code injection will be our future work. In the experiment, the injection will triggered by a vmcall trapped into the injection handler. The injection handler will randomly flip one bit in the registers and resume to the hypervisor. In our experiment, all the injection errors can be detected by TinyChecker.

B. *Performance evaluation*

We have evaluated the performance overhead of TinyChecker and pure nested hypervisor by comparing them with vanilla Xen. The version of vanilla Xen is 4.0.0, and the version of Domain0 kernel is 2.6.31.13. The configuration of VM is 1 VCPU, 1G memroy and 4G disk running debian Linux.

The benchmarks include Kernel Build and dbench. Kernel build is to measure the slowdown for CPU-intensive workload. Dbench tests the slowdown of disk I/O-intensive workload. It requires frequent interactions between the hypervisor and a guest VM due to the frequent data exchanges. As evaluated in prior systems (e.g., Turtles [12] and CloudVisor [13]), this benchmark can be viewed as a worst-case benchmark. All the benchmarks have been tested over five times.

Compared with Xen, there are 1.27% and 2.76% slowdown for nested virtualization and TinyChecker respectively in kernel build. The main reason for the nested virtualization slowdown is that there are twice as many vmexits as in Xen. Among the source of overhead, hashing the VMCS contributes to the other performance loss in TinyChecker.

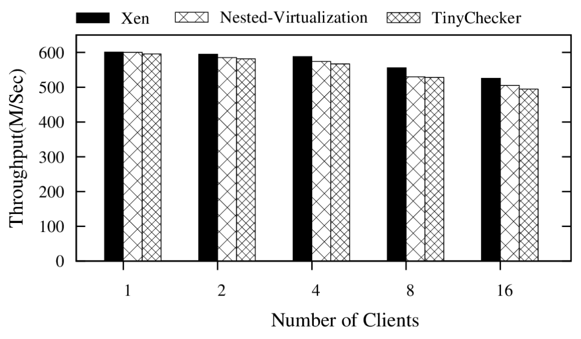

Figure 3. Performance slowdown of TinyChecker on dbench

As shown in figure 3, dbench experiences relative small performance slowdown. The TinyChecker throughput loss in 1, 2, 4, 8 and 16 clients are 0.94%, 2.29%, 3.63%,5.03% and 5.85%respectively.

VI. Related Work

Microreboot [8] enables rebooting fine-grained application components to recover from software failure. It provides the basic idea that if rebooting a part of system can solve the problem, the cost of recovery can be reduced.

RootHammer [6] can fast rejuvenate a virtualized system by only rebooting the hypervisor without affecting VMs running on it. Before rejuvenation, RootHammer will freeze all VMs in memory which called on-memory suspend. Then, hypervisor is quick reloaded without losing current state. When the new instance of hypervisor has been initialized, it will merge the reserved state and begin to run. RootHammer has similar framework but different motivation with TinyChecker. It focuses on the performance of hypervisor microreboot and never considers hypervisor failures.

Otherworld [9] enable applications survival across kernel failures. When a critical error is encountered, Otherworld will microreboot a new kernel within a reserved memory space, in order to reserve all the previous memory. Then, all the application's information, which involving open file, signal handlers, shared memory IPC and so on, will be restored based on the preserved memory. Compared with Otherworld, TinyChecker has more clear recovery logic since the interfaces between VM and hypervisor is much simpler than interfaces between OS and process.

ReHype [7] introduces a mechanism for recovery from hypervisor failures by booting a new instance of the hypervisor while preserving the state of running VMs. ReHype can detect crash and hang failures happened in hypervisor and repair the corruption by rebooting hypervisor. After reboot, ReHype tries to resolve inconsistency in the system. ReHype does not have the ability to detect all kinds of failures and cannot recover from critical data corruption. Furthermore, its detection and recovery modules have tightly coupled with hypervisor which can be easily contaminated or corrupted. Compared with ReHype, TinyChecker provides protection for the critical data in memory and uses context-aware checking to detect all kinds of failures. Furthermore, using nested virtualization TinyChecker can be free from the hypervisor failures and nearly zero modification should be made to hypervisor.

CloudVisor [13] is a security monitor that leverages nested hypervisor to protect the privacy and integrity of customers' virtual machines in virtualized infrastructures. However, CloudVisor mainly focuses on the security and privacy of guest VMs instead of reliability.

VII. Conclusion

To protect VM against hypervisor failures, this paper introduced TinyChecker, which uses nested virtualization to enhance the system's reliability. TinyChecker used context-aware checking and on-demand checkpoint to detect failures and recover the corrupted states of the system. All the four types of failure defined in our paper may be detected and handled by TinyChecker. Through performance evaluation, we have shown that TinyChecker only incurs trivial performance overhead, even for a worst-case benchmark.

References

[1] 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*, Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, 2003

[2] Common vulnerabilities and exposures *http://cve.mitre.org/*

[3] Intel Virtualization Technology Specification for the IA-32 Intel Architecture *http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html*

[4] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. *Terra: A virtual machine-based paltform for trusted computing*. ACM SIGOPS Operating Systems Review, 37(5):206,2003

[5] L. Singaravelu, C. pu, H. Hrtig, and C. Helmuth. *Reducing TCB complexity for security-sensitive applications: Three case studies*. In proc. Eurosys, 2006

[6] K. Kourai and S. Chiba. *A Fast Rejuvenation Technique for Server Consolidation with Virtual Machine*. International Conference on Dependable Systems and Networks, Edinburgh, UK, 2007

[7] M. Le and Y. Tamir. *ReHype: Enabling VM survival across hypervisor failures*, 3rd ed. In Proc. 7th ACM VEE, pages 63C74,Mar. 2011

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. *Microreboot - A Technique for Cheap Recovery*. 6th Symposium on Operating Systems Design and Implementation, San Francisco, CA, pp. 31-44, 2004

[9] A. Depoutovitch and M. Stumm. *Otherworld - Giving Applications a Chance to Survive OS Kernel Crashes*. 5th ACM European Conference on Computer Systems, Paris, France, pp. 181-194, 2010

[10] T.C. Bressoud and F.B. Schneider. *Hypervisor-based fault tolerance*. ACM Transactions on Computer Systems (TOCS), pp.80-107, 1996

[11] G. Vallee, T. Naughton, H. Ong, and S.L. Scott. *Checkpoint/restart of Virtual Machines Based on Xen*. Proceedings of the High Availability and Performace Computing Workshop (HAPCW 2006), Santa Fe, New Mexico, USA, 2006

[12] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. HarEl, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. *The Turtles Project: Design And Implementation of Nested Virtualization*. In Proc. OSDI, 2010.

[13] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. *CloudVisor: Retroftting protection of virtual machines in multi-tenant cloud with nested virtualization*. In 23rd SOSP, October 2011